Int J Parallel Prog (2015) 43:86–129
DOI 10.1007/s10766-013-0277-2

Evaluation of Speculation in Out-of-Order Execution of Synchronous Dataflow Networks

Daniel Baudisch · Klaus Schneider

Received: 29 January 2013 / Accepted: 4 October 2013 / Published online: 22 October 2013

Abstract Dataflow process networks are a convenient formalism for implementing robust concurrent systems that has been successfully used for hardware and software systems in the past. However, the strictly stream-based execution limits the performance of dataflow process networks and requires to carefully balance the entire execution to avoid backpressure and idle nodes. Inspired by related techniques used in processor architectures, we already introduced in our previous work out-of-order execution of dataflow process networks. In this paper, we extend this improvement with speculation of input values for process nodes and allow otherwise idle processes to start computations with speculated input values. Clearly, outputs based on speculated inputs have to be held back until the speculation can be proved right, and have to be withdrawn in case the speculation was wrong. In contrast to related work, our approach has been implemented purely in software using standard hardware to address a broad field of multicore processors. Moreover, a software implementation allows us to dynamically adapt parameters to the needs of the application. This allows us to enforce a user-defined hit ratio of speculation that might even switch speculation off. After a detailed description of this approach and a discussion of possibilities of its implementation, we show its feasibility using a couple of benchmarks. In these benchmarks, the use of speculation achieved an average speedup of 1.2 compared to the non-speculative out-of-order execution.

Keywords Dataflow networks · Multithreaded software · Compiling · Out-of-order · Speculation

D. Baudisch (✉) · K. Schneider
Embedded Systems Group, Department of Computer Science,
University of Kaiserslautern, Kaiserslautern, Germany
e-mail: baudisch@cs.uni-kl.de

K. Schneider
e-mail: klaus.schneider@cs.uni-kl.de

Contents

1 Introduction

Dataflow process networks (DPNs) [13,28,32,38,50] are a quite simple, but nevertheless very powerful model of computation. DPNs consist of a finite number of processes which run in parallel without global coordination. Instead, the individual processes perform their computations independently of other processes and start as soon as the data values required for a computation step are available. In order to exchange data, the processes are connected by a set of fixed FIFO buffers. Each FIFO buffer has a unique source and a unique sink process that either writes values to or reads values from the FIFO buffer.

Since the model of computation is very general, it can be implemented in many ways. Its core has served as a basis for computer architectures [2,17,19,31,57], as well as for the design of programming languages and libraries [7,10,25,27,44,48,56]. It does not demand the use of special programming languages, and instead, allows one to use traditional sequential programming languages for the implementation of the process nodes. The main functionality that the libraries have to provide is the ability that several nodes can run in parallel with a communication over FIFO buffers.

However, a drawback of this generality is the impossibility to guarantee certain properties of the network. For example, *determinism*, i.e., whether the same inputs are mapped to the same outputs (independent of the firing schedule) is a desired property.

Similarly, *boundedness* of buffers, i.e., whether a DPN can be run with buffers of finite size, is also a very crucial property: while the size of FIFO buffers is unlimited in the general DPN model, it has to be finite for any practical implementation. One way to guarantee both properties is to impose certain restrictions so that the considered DPNs are deterministic and have decidable boundedness or liveness problems. Synchronous dataflow (SDFG) networks [8,9,13,32,36–38] and cyclo-static dataflow networks [11,20] are such restricted cases, which have become very successful, in particular, for the synthesis of signal processing systems [1,9,22,23,33,34,47,51]. They are a special kind of DPNs where in each firing step, each process node always consumes the same number of data values from the input streams and produces the same number of data values for the output streams.

A correct implementation of an SDFG network in a sequential language (without any additional library) is fairly simple: Process nodes are mapped to usual functions. With the help of the input and output dependencies of a node, a periodic schedule for the functions can be generated, which calls the functions one after the other. Thus, this class of DPNs can be implemented in a completely sequential manner. However, in order to exploit the power of state-of-the-art multicore processors, concurrent implementations are necessary to achieve the best performance.

On these concurrent systems, scheduling of functions can be organized in different ways, which may be roughly categorized into static (at compile-time) and dynamic (at runtime) variants [35]. While static variants may be appropriate for applications that run on real-time DSP multicore processors, where the final architecture and all running processes are known, the dynamic variant is the best choice for most applications for the following reasons: First, as the schedule depends on the architecture, the dynamic variant can adapt its schedule depending on a particular machine as well as for the inputs obtained ar runtime. Second, even with a fixed architecture, it is very hard to estimate the runtime of tasks in the context of other running processes and cache effects. Therefore, we focus on the dynamic variant in this paper which wraps the code for the individual nodes into tasks that are scheduled to a number of worker threads. Thereby, the number of worker threads should match with the number of available compute units to avoid additional overhead due to context switches.

We have shown in [6] how out-of-order execution, which is a well known technique used by many processor architectures, can be used in this context. In an unbalanced DPN, the values of some FIFO buffers may be produced faster than they were consumed. To ensure bounded memory, it is then necessary to execute the consumer node of this buffer several times in parallel, leading to a better utilization of resources. However, varying computational effort of the single executions of this node may produce outputs that arrive out-of-order, enabling other nodes to fire out-of-order as well. This out-of-order execution avoids idle time and can speedup the program execution. As data is now produced out-of-order for some nodes, the approach requires an additional effort to finally reorder the correct flow of output data. While static scheduling of DPNs follows the philosophy of statically scheduled processors (like VLIW processors) [16,21], our OOO-DPNs refer to dynamic scheduling similar to processors with out-of-order execution.

The main contribution of this paper is to extend the out-of-order execution of dataflow networks presented in [6] with a speculation mechanism to use idle execution

units for further calculations. The idea is thereby to use otherwise not utilized computational power to proceed with the execution of the program: whenever an execution unit is idle, it selects a task that is not runnable due to missing input values, speculates values for the missing inputs, and executes this task using the speculated values. The results are held back until the missing inputs become known. A comparison of actual inputs and speculated inputs decides later whether to accept the result obtained with speculated inputs or to neglect it. In the latter case, the task is then executed with the correct input values as in a normal execution without speculation. The target machines of our synthesis are standard commercial processors, e.g., based on the x86 and AMD64 instruction set, i.e., we do not need any particular hardware extension. We implemented our approach and evaluated it with the help of several case studies to show the performance gains and overheads.

The rest of the paper is structured as follows: Sect. 2 gives an introduction to dataflow process networks, out-of-order execution and speculation in microprocessors and related work. Section 3 introduces a method to improve the multithreaded execution of DPNs in several ways, i.e., load-balancing and additional flexibility in the provided parallelism. Subsequently, Sect. 4 presents an approach for data speculation in DPNs, which further improves the previously introduced out-of-order execution. Section 5 deals with weak memory models, which promise better performance and additionally challenge software developers. Section 6 demonstrates the feasibility of our speculative out-of-order execution with a couple of benchmarks. Section 7 gives discussions of different points considering our approach. Finally, Sect. 8 concludes the paper.

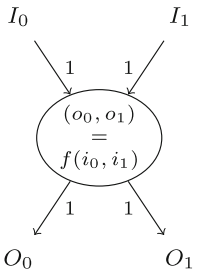
2 Preliminary

2.1 Synchronous Data-Flow

According to Kahn [28], a dataflow process network (DPN) consists of several nodes that are connected by unbounded FIFO buffers. The behavior of an individual node of the DPN is often described by a set of so-called *firing rules* [38]. Such a rule consists of a trigger condition and an action—thus, the model does not rely on a specific programming language. The trigger condition determines under which condition the corresponding action is executed and how many tokens from the input buffers are thereby consumed. Hence, the nodes of the DPN are triggered by the occurrence of input data values, and there is no global coordination like a clock.

A special variant of DPNs are synchronous dataflow graphs (SDFG) [32,36,37], where the number of tokens read from and written to a particular buffer is always constant. The left part of Fig. 1 shows an example of an SDFG node which has two inputs and two outputs. Each time the node fires, exactly one token is consumed from each input, and exactly one token is produced for each output. The corresponding pseudo-code to implement this behavior in a typical imperative programming language is shown on the right-hand side of Fig. 1.

Obviously, there is a straightforward approach to create (multithreaded) code from a SDFG: each node is translated to a separate thread [5], and all threads communicate by




```
thread T_N {
    loop {
        i_0 = I_0.pop()
        i_1 = I_1.pop()
        (o_0, o_1) = f_N(i_0, i_1)
        O_0.push(o_0)
        O_1.push(o_1)
    }
}
```

Fig. 1 *Left-hand side*: SDFG node with 2 inputs and 2 outputs. *Right-hand side*: Pseudo-code for the SDFG node on the left-hand side

simple FIFO queues with each other. To decouple the created threads, each thread runs an infinite loop which waits on input values and then fires one of the rules, i.e., runs code similar to the one given on the right-hand side of Fig. 1. Thereby, the scheduling is left to the operating system, which schedules the created threads according to the presence of input data and availability of resources.

Apparently, the communication overhead of this solution may be significant—depending on the computation time of a single firing of a node, threads may spend too much time for waiting on new inputs. For an efficient implementation, the static communication behavior of SDFGs can be exploited: Purely sequential software (without any buffer synchronization) can be obtained by creating a static schedule [12,32,36,37] consisting of a sequence of node firings that is produced such that it is always known that the required input values are available when a node is fired.

In case of a parallel target architecture, the compiler can also map the nodes to different processing elements. However, in this case, the nodes are no longer independent; instead they have to cooperate by explicit synchronization. Furthermore, in order to fully exploit the parallelism of the target architecture, a compiler has to have knowledge about the average runtime of each node, which is in general a very hard problem. In particular, this becomes very difficult in the context of caches, pipeline stalls, or concurrently running processes on the same machine. As already mentioned in the introduction, we do not want to fix a particular architecture when compiling the program. Hence, our approach creates programs that dynamically adapt to its target architecture and the available resources.

2.2 Out-of-Order Execution in Microprocessors

Out-of-order execution, originating in early work by Tomasulo [62], is a well-known technique used by many processor architectures. Its basic idea is to locally execute a sequential instruction stream of a usual von-Neumann architecture in dataflow order, thereby establishing more parallelism and better load balancing of available functional units.

To this end, the data dependencies between instructions are analyzed and tracked by a couple of data structures. The register set of the processor is extended by the information whether a register is up-to-date; if it is not up-to-date, an additional *tag field* determines the instruction that will finally overwrite the corresponding register.

The *reservation station* (RS) is a table containing all pending instructions. Each entry consists of the op-code, the target register, the operands and the tags in case that the operand is a register that will be overwritten by another instruction. Instructions are loaded by the instruction decoder into the RS. Concurrently, the decoder loads available operands or determines the tag field of the corresponding operand. As soon as all operands of an instruction become available, i.e., their tags are reset, the instruction is enabled and ready for scheduling. The *scheduler* is responsible for scheduling enabled instructions to the available functional units. When a functional unit finishes the processing of an instruction, it sends the instruction ID and its result to the RS, which is responsible to update its operand fields. Thus, the Tomasulo algorithm follows the dataflow order to drive the computation, instead of the original program order given by the order of instructions. The ability of processors to be interrupted by external signals, e.g., by wrong branch prediction, requires to update the registers and the memory in program. For this reason, a reorder buffer serves as an intermediate buffer for completed calculations, which are used in program order to update the registers and the memory. From an external view, this preserves an in-order behavior of the processor.

2.3 Speculation in Microprocessors

In principle, DPNs are intended for an in-order processing of their input streams similar to the in-order processing of an instruction stream by a microprocessor. Analogously to the out-of-order execution of modern microprocessors, we have shown that one can transfer the concept of out-of-order execution to DPNs [6]. Hence, it is natural to consider the question whether other ideas used in processor design can be transferred to DPNs as well. Speculation is such a technique that is motivated by the basic idea to usefully utilize idle computation power.

Historically, speculation in processors has its roots in pipelined processors [53]. Pipelined execution is state-of-the-art in the design of today's processors. Similar to a production line, the processing of an instruction is split into several steps. A pipelined CPU processes several instructions in parallel; thereby each instruction is in a different stage of the pipeline where a part of the entire execution is performed in parallel to the other stages. The efficiency of a pipelined processor strongly depends on how much its pipeline is utilized. Optimal utilization is usually given when each stage in a pipeline is busy. Conditional branches can prevent the processor of continuously executing an instruction stream, because the result of a branch instruction is usually determined in later stages of the pipeline, but needed to determine the instruction to be fetched next. Hence, without speculation, the processor would be forced to stop reading the next instructions until the value of the branch condition is known. To avoid this idle time, modern processors speculate about the value of the branch condition [39] and continue executing instructions from the predicted branch. If the prediction finally turns out to be correct, i.e., the speculatively executed instructions are then known to be correct, no time is lost due to stalls. Otherwise, one has to remove all speculatively executed instructions from the pipeline and to roll-back all changes that have been done by these instructions. To conclude, speculation in processors is widely used to speed up the execution, but this requires an additional effort for the implementation of the processors.

In this paper, we show how speculation about input values can also beneficial for the execution of DPNs if the nodes would otherwise be idle. Related work on this subject is presented in the next section.

2.4 Related Work

In the context of SDFG, several frameworks such as KAAPI [30] or StarSS [52, 61] exist, which simplify the programming of multithreaded implementations. Both provide libraries for the concurrent execution of nodes and their communication, which are based on a *task* model. In general, such a task is an atomic piece of work, i.e., communication with other tasks is done at the beginning and at the end of the task. These tasks are light-weight substitutes of threads, which can be scheduled to processing elements. The main advantage of tasks is a reduced overhead compared with context switches. Since a task does not communicate with other tasks during its execution, it can be completely executed before switching to another task, i.e., a task is not interrupted. In contrast, threads may be interrupted due to communication primitives or or due to a time-sharing scheduling policy.

In order to efficiently use these tasks, both frameworks support the scheduling of tasks by compiler directives, which are read by a front-end of a C compiler. Programmers must annotate the program with the help of these directives to tell the compiler the input and output dependencies between the nodes. They are used at runtime: with this information the scheduler may then dispatch any task without pending dependencies.

In contrast to our approach, both frameworks are only aware of the (static) dependencies between the nodes in the SDFG. We additionally maintain the (dynamic) dependencies between the individual executions of the nodes. This allows us to use out-of-order execution if data values of later executions are already available.

Using out-of-order execution in the context of synchronous dataflow has already been considered by Dennis and Misunas. In [18], they propose a custom-tailored hardware architecture to execute a specific light-weight format of SDFGs. This format restricts the size of its buffers to one token. Furthermore, all nodes are taken from a set of predefined mathematical instructions (`add`, `sub`, `mul`, `div`, ...), where each one has at most two inputs and two outputs. Since each node represents an atomic action, it can be mapped to an opcode of fixed size.

The hardware architecture proposed in [18] is inspired from general dataflow computers, and it basically consists of four parts: the memory, an arbitration network, a distribution network, and the operation units. The memory only stores so-called instructions cells, which describe the nodes of the SDFG and keep track of the presence of validity of tokens. With the help of this information, the hardware can mark instruction cells as soon as all their inputs are valid. Then, the arbitration network, which connects memory and operation units, handles the actual dispatching to resources. With the help of the distribution network, the results are transferred back to the corresponding instruction cells so that the next computations are triggered.

There is a lot of related work about speculative execution: [3,64] present approaches to parallelize sequential programs using a master/slave design. Both approaches use speculation as a primary component to create parallelism. In our case, we already have a parallel program running and want to use speculation to exploit more parallelism.

Therefore, our speculation has to be unobtrusively integrated into the existing out-of-order execution, i.e., speculations should have lower priority compared to the actual execution. Moreover, our approach is generally distributed and provides better scalability than master/slave approaches, where a master usually becomes the bottleneck when the number of slaves is increased.

All methods presented in [14,15,24,26,29,40–43,49,52,54] require hardware support to speculate, e.g., by adding features to a processor to efficiently handle speculated data or to recognize wrong speculations. Rangan et al. parallelize loops where loop-carried dependencies are handled using control flow speculation [63]. The approach differs to previously mentioned work in that it particularly considers loops, while others consider sequential programs in general. However, similar to previously mentioned related work, it requires special hardware, in particular versioned memory. Our approach runs completely in software and simply requires a shared memory SMP, e.g., an Intel x86 or AMD64 multicore architecture, which can be found in most desktop computers. Besides the requirement for special hardware, only [24,40,42,43] include data speculation in their approaches, which is also done in our approach.

Moshovos et al. [46] considers techniques to efficiently speculate values for speculative execution. This paper addresses the issue of achieving a high hit ratio. Of course, our work would also benefit from good speculations, which is however not in the focus of this paper.

As already stated, our approach is to add speculation to DPNs in a discrete way. In particular, the out-of-order execution should get priority. Speculation should only utilize idle compute units, i.e., speculation should fill in the gaps that would otherwise be obtained by idle processing elements. Furthermore, the mentioned previous work does not consider speculation as an option. Our speculation dynamically adapts to the needs such that it automatically selects tasks with high hit ratio. In case that the hit ratio falls below a user-defined threshold, the speculation turns itself off to save energy.

3 Out-of-Order Execution

Synchronous dataflow graphs has been successfully used in many application areas, in particular, in digital signal processing. Usually, the systems continuously read a inputs from their environment, calculate the outputs and update their states (with the help of some function T) and finally, write the output values to the environment. Figure 2 sketches this general scheme.

As Fig. 2 shows, the state of the system S does not allow a trivial parallel execution of several iterations. As the inputs are read in each step, they are responsible for the

```
SystemState_T S, S'
loop {
    I = read inputs;
    (O, S') = T(I, S);
    write outputs(O);
    S = S';
}
```

Fig. 2 Pseudo-code of an interactive system

dependencies between different iterations. In the following, we explain the idea of out-of-order execution and its implementation.

3.1 Motivation

In contrast to reactive systems, applications from the digital signal processing domain often work in so-called bursts: they supply many inputs at once, and the system may use them to compute many outputs. Hence, a parallel implementation may benefit from tracking the dependencies between tasks from *different* executions of T. Thereby, we can exploit parallelism between different iterations and not only within a single iteration. As a consequence, we obtain an out-of-order execution (with respect to iterations).

Assume a system is given that has to be executed as shown in Fig. 2. To execute the system on a parallel architecture, one could partition the system as described in [4] to get a DPN. Thereby, the previously created partitions yield the nodes of the DPN and dependencies between partitions are basically replaced by FIFO buffers. Section 2.1 already outlined the approach of [5], which translates each node of the DPN to a single thread. Communication can be done, e.g., using the Intel TBB concurrent bounded queue. However, the amount of parallelism is (1) fixed to the number of nodes, and (2) relies strongly on a balanced partitioning. While the former makes an application inflexible to the amount of parallelism provided by the targeted hardware architecture, the latter may result in an unbalanced workload.

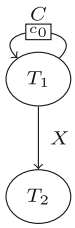
Figure 3 (a) shows an example DPN, (b) the timeline of the execution using the mentioned approach and (c) a timeline of the execution using an improved approach. The DPN in Fig. 3 consists of two nodes T_1 and T_2. To demonstrate the effect of unbalanced computational efforts, we assume that the computational effort of T_1 is half as that of T_2, i.e., the execution of a single iteration of T_1 takes only half the time required for a single iteration of T_2.

The issue of using the approach as described in [5] can be seen in the timeline depicted in Fig. 3 (b): Mapping each node to a single thread provides only two threads. A system with more cores will not be completely utilized, because one or more cores will be idle. Nodes with unbalanced computational effort will even lead to a worse utilization. A practical implementation of an application will use bounded buffers due to limited memory size. If two nodes communicate through a FIFO buffer and one node can be executed faster than the other node, then the faster node will have to wait sooner or later for the other node. In the given example, the producing node is faster and will start to stutter as soon as buffer X is filled. In particular, if buffer X is full, T_1 has to delay further computations until T_2 removes tokens from buffer X. The ability of full buffers to stall preceding nodes is called backpressure.

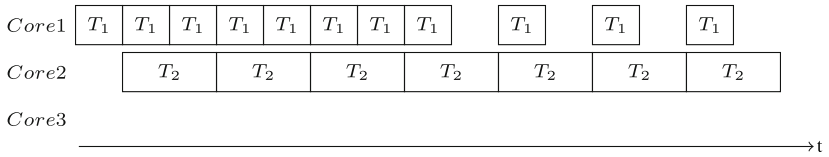
As can be seen, node T_2 has no self-dependency. This means that the computations of an iteration of this node do not depend on preceding iterations. It would be reasonable to execute several iterations of node T_2 concurrently as shown in the timeline given in Fig. 3 (c). This would eventually lead to a better utilization of all cores.

A solution to execute several instances of a node in parallel has already been considered by Stulova et al. [60]. They analyze a given SDFG with respect to the computational effort of nodes and their dependencies. Nodes with higher computational

(a) DPN consisting of two nodes with $O(T_1) \ll O(T_2)$:



(b) Timeline for mapping each node to a single thread with a FIFO buffer of size=4:



(c) Timeline for a task-based execution considering (in-)dependencies across iterations:

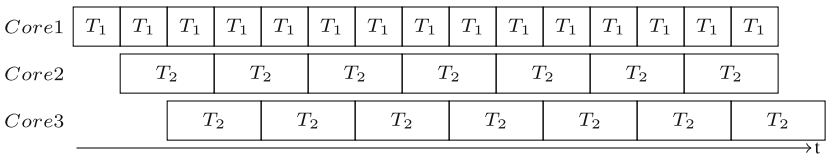


Fig. 3 Motivation for the task-based out-of-order execution of DPNs. The timelines in (**b**) and (**c**) show the execution of the DPN for different approaches. We assume that the computational effort of T_1 is half of the computational effort of T_2. For both approaches, we used bounded FIFO buffers with size 4. Mapping each node to a thread will lead to a fixed amount of parallelism. For that reason, the third core in timeline (**b**) is idle. The unbalanced computational effort leads to backpressure, i.e., node T_1 will start stuttering after filling the buffer to T_2. T_2 has no self-dependency, and therefore, several iterations of T_2 can be executed in parallel, leading to a better utilization of the given system

effort compared to other nodes of the SDFG are then duplicated to gain a more balanced workload. While this approach allows one to obtain more parallelism by node duplication and an improved workload, the final amount of parallelism is still statically determined. Moreover, that approach requires precise knowledge of the target architecture to determine the computational effort. However, real-world applications often contain control-flow, e.g., if-then-else statements, and therefore generally lead to a varying amount of computational effort. This makes static analysis and scheduling very difficult. For this reason, we target a dynamic approach for distributing work to multicore systems.

Lightweight parallel programming is allowed by libraries and APIs that use a task-based execution model, e.g., SmpSS [52], OpenMP[1] or Intel TBB.[2] The main idea is to decouple available parallelism in an application from the available hardware

[1] http://www.openmp.org/mp-documents/spec30.pdf.

[2] http://threadingbuildingblocks.org.

parallelism. Instead of assigning parts of work to threads, these parts are defined as tasks. The application creates fine-grained tasks to provide a large amount of parallelism and schedules executable tasks to a task queue. A set of so-called worker threads execute the given tasks. This provides several advantages compared to pure threaded execution: (1) The number of worker threads is chosen to be the number of available computational nodes, e.g., processors or cores. This avoids overhead of context switches and potential cache thrashing, which are introduced by time-sliced thread execution as done in most operating systems. (2) A worker is not restricted to execute only one task at a time. Depending on the actual hardware, a worker thread may choose to get multiple tasks at once from the task queue. This improves the relation of actual work to synchronization overhead at the cost of parallelism. Hence, repartitioning is not necessary for a task-based execution. To this end, we decided to use the advantages of task-based execution for the presented approach. Thereby, a task is defined as the execution of a single step of a node.

Assuming that we are able to concurrently execute tasks of the same node, the following situation may occur: Varying amount of computational effort of a node and concurrent execution of corresponding tasks can lead to out-of-order arrival of computed outputs. The approach of Stulova et al. [60] does not only statically duplicate nodes, but also outgoing buffers of these nodes. A static merging mechanism has to be implemented into each dependent node, i.e., a node that reads from duplicated buffers. However, this solution will not work for dynamic duplication, i.e., concurrent task execution, which is targeted here. Inspired by out-of-order execution in dynamic processors, we handled this issue by using a centralized buffer. The centralized buffer provides different functions: random access to that buffer allows reordering of values without a merging element. Moreover, computed output values are input values of pending tasks. Hence, out-of-order arrival of computed values may even trigger other tasks out-of-order, thereby allowing more parallelism. In the next section, we will give a detailed description of the required data structures and a pseudo-algorithm of our approach.

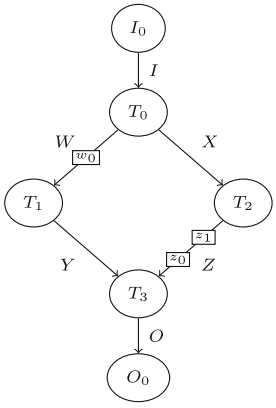
3.2 Data Structures and Algorithm

SDFG and Task Functions. Throughout this section, we use a running example, an SDFG consisting of four nodes T_0, T_1, T_2 and T_3 (see *SDFG* in Fig. 4). The code of the individual nodes is given in the *Task Functions* section in Fig. 4: when a node is triggered, the given task is executed in the final implementation, i.e., the given function is applied to the values belonging to the corresponding iteration.

Before we present our out-of-order execution, we first modify the given SDFG so that the communication with the environment is accomplished by additional nodes I_0 and O_0 (as shown in the *SDFG* section). These nodes are guaranteed to be executed in order (technically realized by a delayed self-dependency, i.e., the node has a feedback buffer to itself with one initial token) so that the behavior at the interface remains unchanged by internal out-of-order executions.

SDFG Description. As already explained above, our approach dynamically maintains a list of dependencies between task executions (which is defined by a tuple containing

SDFG:



Task Functions:

```
 1 : macro CBS'[i]. = CBS[i mod CBS_size].
 2 : macro CBS'_V[i]. = CBS'[i].values
 3 : function Fun_I(i){
 4 :     CBS'_V[i].I = ReadInputs()
 5 : }
 6 : function Fun_0(i){
 7 :     I = CBS'_V[i].I
 8 :     assert (I ≠ 0) // avoid division by zero
 9 :     CBS'_V[i+1].W = 1.0/I
10 :     CBS'_V[i].X = g_T0(I)
11 : }
   :
12 : function Fun_O(i){
13 :     WriteOutputs(CBS'_V[i].O)
14 : }
```

Fig. 4 *Left*: example SDFG. *Right*: the function of the nodes and how to access input and output buffers, i.e., data in the CBS

SDFG Description:

```
 1: structdef SystemState = {I, X, Y, Z, W, O}
 2: structdef TaskDesc = {NodeName : NodeID, iteration : int}
 3: structdef DependencyDesc = {NodeName : NodeID, numInitialTokens : int}
 4: structdef NodeDesc = {NodeName : NodeID, NodeFun : Function,
 5:                       Successors : list < DependencyDesc >}
 6: structdef CBSEntry = {iteration : int, RT : int,
 7:                       executed : bool [] , numInDeps : int [] ,
 8:                       values : SystemState}
 9: SDFG_Desc = {
10:     ( I_0, Fun_I, {(I_0, 1), (T_0, 0)}),
11:     ( T_0, Fun_0, {(T_1, 1), (T_2, 0)}),
12:     ( T_1, Fun_1, {(T_3, 0)}),
13:     ( T_2, Fun_2, {(T_3, 2)}),
14:     ( T_3, Fun_3, {(O_0, 0)}),
15:     (O_0, Fun_O, {(O_0, 1)})
16: } : list < NodeDesc >
17: NodeNames = {n|T ∈ SDFG_Desc ∧ n = T.NodeName}
18: function GetDescOfTask(n) {T|T ∈ SDFG_Desc ∧ T.NodeName = n}
19: function GetSucsOfTask(n) {GetDescOfTask(n).Successors}
20: function GetNumSucsOfTask(n) {List.length GetSucsOfTask(n)}
21: function GetPresOfTask(n) {{(k, t)|k ∈ NodeNames∧
22:                              (n, t) ∈ GetSucsOfTask(k)}}
23: function GetNumPresOfTask(n) {List.length GetPresOfTask(n)}
24: function GetMaxInitialToken(n)
25:     {max_{n∈NodeNames}(GetSucsOfTask(n).numInitialTokens)}
```

Fig. 5 Structures and data for the example SDFG: the description of the SDFG and functions to access members of the description

the SDFG node and the iteration). We use the data structures given in Fig. 5 to store these dependencies and the actual structure of the SDFG.

– *SystemState* contains a list of all buffers of the actual system.
– *TascDesc* identifies a specific task by the node and the iteration.

- $DependencyDesc$ describes a dependency from a task to another task that is identified by its corresponding node $NodeName$ and the distance of iterations, which is given by the number of initial tokens. The distance is explained as follows: if a node T is connected to another node T' by a buffer, then these nodes are dependent. If that buffer contains N initial tokens, the ith iteration of T will produce a value that is consumed by the $i + N$th iteration of T'.
- $NodeDesc$ contains a tuple describing all characteristics of a node that are important for the out-of-order execution. This includes the identifier of the node, a function to execute a task for the node and a list of outgoing dependencies.
- $CBSEntry$ describes a single entry of the central buffer station as described in the next paragraph. It contains the state of a single iteration of the DPN in the out-of-order execution, i.e., the state of tasks (pending or executed), the remaining dependencies to keep track of whether a task can be executed (member $numInDeps$), the remaining number of dependencies before an entry can be removed (member RT) and the system state (member $values$), which is addressed by the macro definitions given in lines 1f on the right hand side in Fig. 4.
- $SDFG\_Desc$ depends on the application and describes the nodes of the actually given SDFG. $NodeNames$ is a list of all nodes and is required by the functions for the out-of-order scheduling. The functions given in line 18 to line 24 are used to access members of the previously defined structures and for convenient access to parameters that are implicitly given, e.g., the number of outgoing dependencies of a node (see function $GetNumSucsOfTask$ in line 20).

Central Buffer Station. Due to the out-of-order execution of nodes of the SDFG, the nodes might not read and write the data in-order. Thus, we have to replace FIFO buffers by another data structure that gives nodes random access. As SDFGs allow the compiler to determine a static schedule, we can safely use buffers of fixed size and determine a lower bound for their size.

Inspired by the reservation station used by out-of-order processors, we store all buffers in a table—the central buffer station (CBS). This table includes the buffers for all input, output and state variables of the system. Similar to the reservation station of a microprocessor, it is organized as a ring buffer that provides access to at least *WS* iterations, where *WS* is the number of newest iterations that can be concurrently handled by the scheduler (called the window size of the scheduler). In contrast to the reservation station, the data in the CBS is not removed with a schedule, because each entry will require one schedule for each task. In addition, the CBS also serves as a reorder buffer to write outputs in-order to the environment. Hence, entries of the CBS are added and removed in-order. Since a buffer may have some initial tokens and a concurrent execution of *WS* iterations may write *WS* additional tokens into each buffer, the CBS must have a size of $\mathrm{CBS}_{\mathrm{size}} = N_{max} + WS$ entries where N_{max} is the maximum number of initial tokens. To allow out-of-order scheduling for all nodes, *WS* must be chosen to be greater than 1. The head entry (i.e., CBS entry indexed by $head$) and the tail entry (i.e., CBS entry indexed by $tail$) of the ring buffer address the oldest and the newest, respectively, iteration that are processed. The CBS for our running example after its initialization can be found in Fig. 6. The $speculationBuffer$ in the table is a member, which is required for speculative

Central Buffer Station (CBS) / task queue after initialization:

$head = 0,\ tail = n,\ TaskQueue = \{(I_0, 0), (T_1, 0)\}$

Entry	*iter–ation*	*RT*	*executed* $I_0 \ldots O_0$	*numInDeps* I_0	T_0	T_1	T_2	T_3	O_0	*values* I	X	Y	Z	W	O	*speculationBuffer* $I_0 \ldots$	O_0
0	0	6	$f \ldots f$	0	1	0	1	1	1				z_0	w_0		{} …	{}
1	1	7	$f \quad f$	1	1	1	1	1	2				z_1			{}	{}
2	2	7	$f \quad f$	1	1	1	1	2	2				z_1			{}	{}
⋮	⋮	⋮	⋮ ⋮	⋮	⋮	⋮	⋮	⋮	⋮							⋮	⋮
$\text{CBS}_{\text{size}} - 3$	$\text{CBS}_{\text{size}} - 3$	7	$f \quad f$	1	1	1	1	2	2							{}	{}
$\text{CBS}_{\text{size}} - 2$	$\text{CBS}_{\text{size}} - 2$	7	$f \quad f$	1	1	1	2	2	2							{}	{}
$\text{CBS}_{\text{size}} - 1$	$\text{CBS}_{\text{size}} - 1$	7	$f \ldots f$	2	2	1	2	2	3							{} …	{}

Fig. 6 CBS for example SDFG: the CBS after the initialization process and the task queue containing tasks that can be started. The *speculationBuffer* is only required for the speculative approach

execution. It will be added in Sect. 4 to the structure, and the reader can ignore it at this moment.

The read and write positions of a buffer in an iteration i are determined as follows: All nodes will always read the ith element, and therefore, the read position is $\text{ReadPos}(B) = i \bmod \text{CBS}_{\text{size}}$. The write position depends on the number of initial tokens for a buffer B: let B_{init} be the number of initial tokens for buffer B, then the write position for that buffer is defined as $\text{WritePos}(B) = (i + B_{\text{init}}) \bmod \text{CBS}_{\text{size}}$ (see pseudo-code of nodes in the right-hand side of Fig. 4).

In addition to the system's values and its state, the scheduler requires information about the tasks, e.g., whether a task is pending or has been executed, or how many dependencies are left to enable the node. In particular, for each task T, we add a counter $numInDeps.T$ and initialize it with the number of incoming dependencies which can be obtained from the SDFG description (see Fig. 5). To keep track of whether the head of the CBS can be removed, we add for each entry E of the CBS a counter $RT[E]$ for the remaining non-executed tasks. Some of the counter need an additional dependency, which is used to ensure correct execution. A detailed explanation is given in the following by the description of function *ReplaceHead*.

Scheduler. Figure 7 shows the pseudo-code for our worker thread, which is responsible for executing pending tasks. It selects a task from the *TaskQueue* (see line 3) and executes it using the task function of the corresponding node (see line 5). After a task has been executed, the corresponding *executed* flag is set. As explained in the previous paragraph, the *RT* counter is used to keep track of whether a head can be removed. Hence, after a task has been executed, the counter $RT[i \bmod \text{CBS}_{\text{size}}]$ is decremented (see line 8). When $RT[head]$ reaches 0, all tasks of the head have been executed, and the head can be removed (see line 9 and function *ReplaceHead* in Fig. 7). Moreover, the executed task provides input values for its depending tasks. Hence, the scheduler must decrement each counter of its successors (see line 10 and function *DecrDepsOfSucs*). As soon as a counter reaches the value 0, the corresponding task must be scheduled (lines 17f).

Function *ReplaceHead* is responsible to replace the head of the CBS, which is done after all tasks of the head have been executed. In practice, this means to reset the

Worker Thread (Scheduler):

```
 1 : thread Worker () {
 2 :     loop {
 3 :         T = TaskQueue.pop()  // blocking!
 4 :         LoadFence()  // required in weak-memory model
 5 :         GetDescOfTask(T.NodeName).NodeFun(T.iteration)
 6 :         StoreFence()  // required in weak-memory model
 7 :         CBS'[T.iteration].executed.(T.node) = t
 8 :         v = fetch_and_decr(CBS'[T.iteration].RT)
 9 :         if (v == 0)  then ReplaceHead()
10 :         DecrDepsOfSucs(T)
11 :     }
12 : }

13 : function DecrDepsOfSucs(TaskDesc T) {
14 :     forall S ∈ GetSucsOfTask(T.name)
15 :         i = T.iteration + S.numInitialTokens
16 :         v = fetch_and_decr(CBS'[i].numInDeps.S.NodeName)
17 :         if (v == 0)  then TaskQueue.push(S.NodeName, i)
18 : }

19 : function ReplaceHead() {
20 :     LoadFence()  // required in weak-memory model
21 :     CBS'[head].iteration = CBS'[tail].iteration + 1
22 :     StoreFence()  // required in weak-memory model
23 :     forall n∈NodeNames
24 :         CBS'[head].executed.n = f
25 :         CBS'[head].numInDeps.n = GetNumPresOfTask(n) + 1
26 :     CBS'[head].RT = 1 + List.length SDFG_Desc
27 :     head = (head + 1) mod CBS_size
28 :     tail = (tail + 1) mod CBS_size
29 :     StoreFence()  // required in weak-memory model
30 :     forall n∈NodeNames
31 :         i = tail − GetMaxInitialToken(n)
32 :         v = fetch_and_decr(CBS'[i].numInDeps.n)
33 :         if (v == 0)  then TaskQueue.push(n, i)
34 :     v = fetch_and_decr(CBS'[head].RT)
35 :     if (v == 0)  then ReplaceHead()
36 : }
```

Fig. 7 Pseudo-code of the out-of-order scheduler

counters to their initial values such that a new iteration can be processed. In particular, the counter $numInDeps.T$ is initialized for each task T with the number of incoming dependencies of task T plus 1 (line 25). The additional dependency is removed in a later step of function $ReplaceHead$. It is used to guarantee that the execution of a task is postponed until the CBS entries for writing computed values become available. Writes to a buffer may address an entry that does not correspond to the entry that is processed. The number of initial tokens in a buffer defines the relative offset, where data has to be placed, when a write access has to be handled. As a consequence, tasks can only be scheduled, when all addressed entries in the CBS are available. For instance, buffer W in the running example has one initial token (see left-hand side of Fig. 4). The ith iteration of node T_0 will write the $i+1$th token into buffer W, i.e., the value will be written to the $i+1$th entry of the CBS. Hence, the execution of task T_0 for iteration i has to be postponed until CBS entry $i+1$ becomes available. A single

additional dependency is sufficient, because entries in the CBS are added in order. All relative offsets of the addressed entries must be 0 or positive and at most the number of initial tokens of that buffer with the most initial tokens. Hence, whenever the entry addressed by the largest offset becomes available, then every preceding entry must be also available. In the following, we call the difference between the iteration of the targeted CBS entry and the iteration counter i of the executed task the *write distance*. The actual write distance depends on the node N and its outgoing dependencies. It is formally defined by function $GetMaxInitialTokens(N)$ given in lines 24f of Fig. 5, which is required at a later point of time in function $ReplaceHead$.

The counter $RT[head]$ is initialized with the number of tasks (line 26) plus 1. Analogous to the additional dependencies to avoid anticipated schedules, we have to avoid premature removal of CBS entries. In other words, we enforce the in-order removal of CBS entries (head first) by using a similar procedure. $RT[E]$ is incremented for all entries except for the head, i.e., after all tasks of an entry in the CBS have been executed, the corresponding entry E will remain in the CBS due to its additional dependency until that dependency is removed by the replacement of the preceding CBS entry (see line 34).

After updating the $head$ and $tail$ counter (lines 27f), the function removes additional dependencies from tasks that ensure a safe write distance as explained before. The reset of the CBS head must be completed before checking whether tasks must be scheduled to prevent race conditions: Otherwise, a task might be scheduled before the reset phase is finished. Since there is no condition that might prevent the execution of the task, it is also possible that this task is executed during the reset phase. At the end of the task's execution, the counters of the task's successors must be decremented, and the corresponding thread might access non-initialized counters. As a consequence, the reset phase would overwrite the counters with wrong values. A separation of these phases such that the reset of the head must be completed before any schedule is made, solves this problem.

Removal of additional dependencies to ensure correct execution is first done for the nodes, i.e., for $numInDeps.T$. A task T in iteration i will eventually write to CBS entry $i + GetMaxInitialTokens(T)$. As explained above, this means that only those tasks T can be scheduled for which $i + GetMaxInitialTokens(T) \leq tail$ holds. This is equivalent to $i \leq tail - GetMaxInitialTokens(T)$. If a CBS entry is replaced, then the new tail is determined by $tail_{new} = tail + 1$. As a consequence, the new condition for ensuring a safe write distance is then given by $i \leq tail_{new} - GetMaxInitialTokens(T)$. We assume that the additional dependency already has been removed for all tasks where $i \leq tail - GetMaxInitialTokens(T)$ holds. It follows that the condition must be changed for those tasks, where $tail - GetMaxInitialTokens(T) < i \leq tail_{new} - GetMaxInitialTokens(T)$ holds, i.e., $i = tail_{new} - GetMaxInitialTokens(T)$. Hence, the dependency counter of each node N is decremented at position $tail_{new} - GetMaxInitialTokens(T)$ (see lines 31f).

The final step of function $ReplaceHead$ is to remove the additional dependency of RT (see line 34). It is important to decrement RT at the end of $ReplaceHead$ to avoid races to the $tail$ counter in line 31. The out-of-order execution allows one that all tasks of the new head might already have been executed. Hence, RT of the new

head can reach 0, and therefore, it might trigger the *ReplaceHead* function for the new head (see line 35). This allows a recursive removal of CBS entries.

In general, it is possible that several threads try to decrement and read the same counter. Hence, the decrement and read operations must be made atomic. Some libraries, e.g., Intel TBB, provide functions and/or data structures to execute such operations atomically. An alternative implementation using ordinary locks and conditions that are provided by most operating systems are not recommended due to their costs. On architectures like the x86 or AMD64, it is better to use spin-locks, e.g., using compare and swap (CAS) operations, because these operations are light-weight and the probability that races occur is for most systems quite low.

4 Data Speculation in Out-of-Order Execution

This section introduces data speculation in addition to the previously presented out-of-order execution. The main goal of this approach is to improve the overall performance of SDFGs by a better utilization of the available processing elements. We will first motivate our approach of speculative execution, which also explains how values are speculated. Afterwards, we will explain the concept and the required modifications of the out-of-order execution.

4.1 Motivation

4.1.1 False Dependencies

In general DFGs, the number of tokens consumed and produced by a node can vary in every reaction step of the node. These numbers may depend on internal states, but also on the values read from the one or the other input buffer in the same reaction step. Since SDFGs allow one to statically derive a schedule, we may wish to convert a general DFG to a SDFG. This can be done by simply adding dummy values for reading and writing to every reaction so that each reaction step will always consume and produce the same number of tokens. The dummy values introduced this way lead then to *false dependencies* in the sense that a node need not really have to wait for data tokens from these input channels.

A dynamic resolution of false dependencies can be achieved by partial evaluation. This means that we start the computation of the task that defines the reaction of a node to determine which values are currently really of interest. If a dependency turns out to be a true one, the current task must be either canceled, which will add further communication, or the task must be blocked. The latter violates the task-based execution model, which requires that a task is completely executed without blocking. Hence, adding this technique to out-of-order execution is definitely not an option.

An alternative way for early evaluation is to start some task whenever computational power becomes available even if there are no pending tasks. This means that we assume that all missing values are only false dependencies and optimistically start tasks. Obviously, this may fail because some of the dependencies may turn out to

be true dependencies. To make a speculative execution of a task more effective we therefore add data speculation.

4.1.2 Speculation of Values

Speculating values is a subtle and non-trivial task. It can be characterized as a process of making a guess on future values based on present and past values. Obviously, the behavior of applications varies, and therefore, the values they compute are also varying. As a result, the actual hit rate of a speculation depends on the application. Lipasti and Shen [40] and Richardson [55] figured out that many real-world applications rarely modify values of variables. Hence, many speculation algorithms just use previous values as speculated values. This fits well with the common requirement to keep the computational effort for speculations low, i.e., speculations should be as fast as possible and require as few resources as possible, e.g., memory accesses.

Our language originates from hardware design and supports variables with different storage types. A detailed description of the source language and its semantics can be found in [58]. To discuss our idea about speculation, we mainly have to distinguish between memorized and event variables: while *memorized* variables behave as registers, i.e., they keep their values unless they are changed, *event* variables are reset to a default value, i.e., a constant value, in case that no assignment is made. Based on the assumption that applications rarely change values of variables, we use a speculation that considers the storage type of variables: For memorized variables, the speculation will return the value from the previous iteration, and for event variables, the pre-defined default value is chosen. In case of memorized variables, the value from the previous iteration might not have been computed, yet, due to out-of-order execution. Special handling of this case is relinquished to keep the speculation process simple. The speculation generally might produce invalid input sets for a node, which is handled by automatically generated code inside of the $specExec$ functions (see example, line 36 in Fig. 9).

4.2 Concept

The idea of our speculation is to select a task for speculative execution as soon as a core is idle, i.e., when no pending tasks are available. According to our motivation, missing input values for executing a task are assumed to be either not required by the selected task or to be some default values, i.e., values of the previous iteration or constants.

In particular, we memorize speculated input values that are used for a speculative task execution. As soon as all inputs for a task are known, we can compare them with the speculated ones. This allows us to use the result of a speculatively executed task in case that some missing values turn out to be true dependencies and their speculated values were correct. Modified and additional data structures are depicted in Fig. 8. In addition to the existing task functions from the out-of-order execution (see right-hand side of Fig. 4), we need the functions shown in Fig. 9. The modified code of the worker thread and additional functions are printed in Fig. 10. Modifications of the worker thread have been marked with underlined line numbers.

SDFG Description - Modified and Additional Structures for Speculative Approach:

```
 1: structdef SpecDesc = {NodeName : NodeID, iteration : int,
 2:                       inputs : SystemState, outputs : SystemState}
 3: structdef NodeDesc = {NodeName : NodeID, NodeFun : Function,
 4:                       specCopy : Function, specGuess : Function,
 5:                       specExec : Function, specCheck : Function,
 6:                       Successors : list < DependencyDesc >,
 7:                       speculationAllowed : boolean}
 8: structdef CBSEntry = {iteration : int, RT : int,
 9:                       executed : bool [] , numInDeps : int [] ,
10:                       values : SystemState,
11:                       speculationBuffer : list < SpecDesc >}
12: SDFG_Desc = {
13:     ( I0, Fun_I, specCopy_I, specGuess_I, specExec_I, specCheck_I, {(I0, 1), (T0, 0)}, false),
14:     ( T0, Fun0, specCopy0, specGuess0, specExec0, specCheck0,{(T1, 1), (T2, 0)}, true ),
15:     ( T1, Fun1, specCopy1, specGuess1, specExec1, specCheck1,        {(T3, 0)}, true ),
16:     ( T2, Fun2, specCopy2, specGuess2, specExec2, specCheck2,        {(T3, 2)}, true ),
17:     ( T3, Fun3, specCopy3, specGuess3, specExec3, specCheck3,        {(O0, 0)}, true ),
18:     (O0,Fun_O,specCopy_O,specGuess_O,specExec_O,specCheck_O,         {(O0, 1)}, false )
19: } : list < NodeDesc >
```

Fig. 8 Modified and additional data structures for speculative execution of our example SDFG: the description of the SDFG requires additional functions for speculating values, doing a speculative execution, and to compare a speculated result

In general, algorithms that use speculation must provide a roll-back mechanism to undo changes, which have been made due to wrong speculations. Similar to other speculation approaches, we hold back changes until the result of a speculation is known to be correct. In particular, input and result values of speculations are not written back directly to the CBS but into a buffer for speculations (see structure *SpecDesc* in line 1f of Fig. 5). A completed speculative execution puts its structure containing the speculated input values and the result into a dedicated buffer in the CBS (see member *speculationBuffer* of CBS in Fig. 6). For fast access to results of speculative executions, each task has a separated buffer.

In the following, we give a step-by-step explanation of code modifications and additions depicted in Fig. 10. The very first modification of the worker is to replace the blocking read from the task queue by a non-blocking read (see line 3). In case that no pending tasks are available, the call to tryPop will return without a token. If a pending task is available (line 4), the worker will first call *CheckSpeculation* to check whether speculation results are available (see line 5). If no speculation results are available or if all speculations failed, the actual task execution is started (see line 7ff). The procedure for executing a non-speculative task is identical to the one of the non-speculative execution. If a speculation has been correct, *CheckSpeculation* is responsible for copying the corresponding results to the *CBS*. Hence, no further work to copy speculative results is necessary in the worker thread. When no task is ready for execution (see line 14), i.e., the worker is idle, a speculation is triggered by calling *DoSpeculation* (see line 15).

The purpose of function *DoSpeculation* (see line 18ff of Fig. 10) is to do a complete speculative execution of a task including associated work. First, a speculation structure is created (see line 19), which will contain all inputs and will have

Task Functions for Speculation:

```
 1 : function specGuess_I(S, t){
 2 :     S.inputs.I = I_default
 3 :     return true
 4 : }
 5 : function specGuess_0(S, t){
 6 :     if (t = 1)  then S.inputs.W = W_default
 7 :     if (t = 0)  then S.inputs.X = X_default
 8 :     return true
 9 : }
   :
10 : function specCopy_I(S, i, t){
11 :     S.inputs.I = CBS'_V[i].I
12 : }
13 : function specCopy_0(S, i, t){
14 :     if (t = 1)  then S.inputs.W = CBS'_V[i].W
15 :     if (t = 0)  then S.inputs.X = CBS'_V[i].X
16 : }
   :
17 : function specExec_I(S){
18 :     assert (false)   // environmental communication nodes must not be speculatively executed
19 : }
20 : function specExec_0(S){
21 :     I = S.inputs.I
22 :     if (¬(I ≠ 0))  then return false // avoid division by zero
23 :     S.outputs.W = 1.0/I
24 :     S.outputs.X = g_T0(I)
25 :     return true
26 : }
   :
27 : function specCheck_I(i){
28 :     assert (false)   // environmental communication nodes must not be speculatively executed
29 : }
30 : function specCheck_0(S){
31 :     if (CBS'_V[i].I ≠ S.inputs.I)  then return false
32 :     CBS'_V[i + 1].W = S.outputs.W
33 :     CBS'_V[i].X = S.outputs.X
34 :     return true
35 : }
   :
```

Fig. 9 Additional task functions for the speculation approach. **Functions** $specCopy_*$ are used to copy output values, which already have been created by the corresponding task, from the CBS to the local speculation structure. **Functions** $specGuess_*$ are used to do guesses for output values that are produced by the corresponding task. **Functions** $specExec_*$ apply the function of a node by using input values from a previously initialized speculation structure. **Functions** $specCheck_*$ are used to check a speculation result. If a speculation result is correct, the function also copies the output values to the CBS

space for the outputs for the task that is going to be speculated. After selecting a task for speculative execution (see line 20), it gets a copy of the actual (available) input values (see line 24f). Known inputs are initialized by copying the values from the CBS

Worker Thread (Scheduler):

```
 1 : thread Worker () {
 2 :     loop {
 3 :         T = TaskQueue.tryPop() // non-blocking!
 4 :         if (T ≠ nil) then
 5 :             specOkay = CheckSpeculation(T)
 6 :             if (¬specOkay) then
 7 :                 LoadFence() // required in weak-memory model
 8 :                 GetDescOfTask(T.NodeName).NodeFun(T.iteration)
 9 :                 StoreFence() // required in weak-memory model
10 :                 CBS'[T.iteration].executed.(T.node) = t
11 :                 v = fetch_and_decr(CBS'[T.iteration].RT)
12 :                 if (v == 0) then ReplaceHead()
13 :                 DecrDepsOfSucs(T)
14 :         else
15 :             DoSpeculation
16 :     }
17 : }
18 : function DoSpeculation() {
19 :     S = new SpeculationResult
20 :     (S.iteration, S.NodeName) = pick a task for speculation
21 :     LoadFence() // recommended in weak-memory model to ensure that inputs are up-to-date
22 :     validSpec = true
23 :     forall (n_P, t) ∈ GetPresOfTask(S.node)
24 :         if (CBS'[S.iteration].executed.n_P) then
25 :             GetDescOfTask(n_P).specCopy(S, S.iteration, t)
26 :         else
27 :             validSpec = validSpec ∧ GetDescOfTask(n_P).specGuess(S, t)
28 :     if (validSpec) then
29 :         validSpec = GetDescOfTask(n_P).specExec(S)
30 :     if (validSpec) then
31 :         StoreFence() // required in weak-memory model
32 :         CBS'[S.iteration].speculationBuffer.(S.node).push(S)
33 : }
34 : function CheckSpeculation(T : TaskDesc) {
35 :     specValid = false
36 :     while (¬specValid ∧ ¬CBS'[T.iteration].speculationBuffer.(T.node).isEmpty())
37 :         S = CBS'[T.iteration].speculationBuffer.(T.node).pop()
38 :         LoadFence() // required in weak-memory model
39 :         specValid = GetDescOfTask(P.node).specCheck(S, T.iteration)
40 :     return specValid
41 : }
```

Fig. 10 Pseudo-code of the out-of-order scheduler with speculation. **thread** *Worker* has to be adapted to the speculative approach. Modifications are highlighted in *dark red*/marked by *underlined italic line numbers*. **function** *DecrDepsOfSucs* and **function** *ReplaceHead* are left unmodified and can be found in Fig. 7. In addition, **function** *DoSpeculation* and **function** *CheckSpeculation* are required in our speculative approach

to the newly created structure using the functions $specCopy_*$. Missing data is initialized by dedicated functions, i.e., $specGuess_*$ (see line 27). Speculated input values have to be buffered to ensure a safe roll-back mechanism in case that a speculation was wrong. They cannot be stored in the CBS, because we must *not* assume that the tasks will write the input values *after* they have been speculated. A task might be currently in progress or might start during the speculation. Hence, the write access to

such an input would eventually end up in a conflict, because either the actual value is irreversibly overwritten or the speculated value cannot be read for comparison in the speculation check.

In addition to the obvious goal to make a "good" guess on missing input values, it is more important to have valid values that are sensible for the program's semantics. This becomes more difficult due to race conditions that might appear during the initialization of input values for a speculation. In particular, the latter issue can be handled only by additional synchronization effort, which should be avoided for performance reasons. Hence, it does not matter if some input values are invalid due to race conditions or based on a speculation, both may cause an assertion/error/fault/exception in specific cases. For instance, if a value for one input is set to 0 and the task is going to divide by this value, an assertion/error/fault/exception might be triggered depending on the final operating system and hardware architecture that is used to run the program. An optimal solution would be to avoid taking values that might result in such a faulty execution, but this is in general a hard task that entails additional computational effort for value speculation. Instead, we solved this issue by automatically adding code to the speculation of input values and the speculative task execution, which checks conditions that might trigger a runtime error. Typical candidates are e.g., division by zero, subtraction of unsigned numbers, out-of-bound array accesses, etc. Moreover, invalid input values can trigger application-specific assertions that have been created by the programmer. In particular, our synthesis method translates assertions in $specGuess_*$ and $specExec_*$ to if-then-statements. Figure 9 gives an example for adding safety code: All inputs of a node can potentially be speculated, and therefore, they can have arbitrary values. The divisor of the division in line 23 is based on an input. Hence, it is necessary to ensure that the divisor has a non-zero value (see line 22). In case that the divisor is zero, the speculative execution will return false to signalize invalid input values. If the speculation of input values or the speculative execution of the task signalize an invalid input value then $DoSpeculation$ will set $validSpec$ to false (see line 27 and line 29). As a result, no remaining code in the function will be executed, i.e., all computations of this function will be neglected.

The actual execution of a speculative task is done using the corresponding functions $specExec_*$ (see line 29 in Fig. 10) and differs in the target buffer, which is now the previously created structure. After the execution, we put the structure into the speculation buffer of the corresponding task (see line 32). The result remains there until a check for the correctness of the speculation is initialized (see line 37ff in Fig. 10).

Buffers may have initial tokens. In addition, a node may write to several buffers with different numbers of initial tokens, i.e., the node produces values that are consumed in different iterations of the CBS. In turn, this means that the input values for a task may be produced by tasks in different iterations. For instance, a task of node T_3 in the example SDFG in Fig. 4 has a dependence to the task of node T_1 in the current iteration and a dependence to the task of node T_2 in the iteration before the previous iteration. Even worse, consider the following example: let A and B be nodes in a DPN. A writes values to buffers x and y, where buffer x is initialized with one token and buffer b is initially empty. Furthermore, B reads from both buffers. Now assume that we want to speculatively execute node B. The input dependencies would show that B has two dependencies to A. In particular, this is a dependency to A in the previous iteration,

which is due to the required value of buffer x, and a dependency to A in the current iteration, which is due to the required value of buffer y. In out-of-order execution, it could be the case that A in the previous iteration has not yet been executed, while A in the current iteration has already been executed. Because each dependency is listed separately, we can decide separately whether to copy or to speculate about the corresponding value, i.e., whether to call $specCopy_*$ or $specGuess_*$. To this end, it is mandatory for a correct behavior of the speculation that the calls to these functions make sure that only the value of interest is written. Therefore, we need parameter t in our functions for copying and speculating input values. It allows us to distinguish between the different iterations that can be written to.

Some nodes do not only represent a function that maps inputs to outputs. For instance, nodes that communicate with the environment cannot be speculatively executed, because their changes cannot be held back and will lead to irreversible modifications. An additional flag in the node structure ($speculationAllowed$ in structure $NodeDesc$ in Fig. 8) determines, whether it is allowed to speculatively execute a task for a node.

Scheduling of non-speculative tasks is left unchanged, i.e., we do not check speculative results when tasks are scheduled. Hence, the functions $DecrDepsOfSucs$ and $ReplaceHead$ remain unchanged. Performing the check for correct speculations into the task execution, results in the additional benefit that the check is executed in parallel. When a task is dequeued, the corresponding worker starts with a check on whether speculative results are available. The worker dequeues all speculative results and checks for each speculation whether the inputs for this task were actually read or equal to the actual inputs, i.e., the inputs of the non-speculative one. Obviously, it should be sufficient to apply this check only to inputs that have been speculated. This raises however some issues due to potential races that can appear in the concurrent execution. In the following, we will discuss this issue and possible solutions.

4.3 Time-Insensitive Check

The easiest way to handle races is to ignore them at the point of the speculation. When a speculation result is being checked, we simply compare all inputs that actually have been used by the task. This is done by an additional function which is created by our code generator. The behavior of each node is deterministic. Hence, applying a task's function to the same inputs will always yield the same outputs. In turn, if the inputs of a speculation and the actual inputs of a pending task are equal, then the speculation result is exactly the same as the result of the non-speculative execution.

As simple as this mechanism is, it provides two advantages: First of all, it never fails because its correctness purely depends on the comparison of input values. Since the complete input set is taken for the comparison, there is no need to detect races that might appear during the initialization of the inputs for the speculative task. Second, this way to check speculation results provides more flexibility. In contrast to the method presented in Sect. 7.1 (least-effort check), it allows one to reuse speculations. It is easily conceivable that input sets for a node may reappear, such that a node may repeatedly produce the same output values. For instance, in audio synthesis and filtering programs, a node that generates continuously a wave, e.g., a sinus curve, the outputs will reappear.

Hence, the check of reusable speculation results would work like a look-up table with a bounded size and a fall-back mechanism in case that a value is not available in the look-up table, i.e., the buffer with speculation results.

We propose the following sequence to do a speculation:

1. *Create a copy of valid input values and speculate about missing input values*: A task is the execution of one iteration of a node. The required inputs to execute the non-speculative task are found by the dependencies described in the SDFG. For a speculative task, we check for each preceding task (i.e., for each incoming dependency) whether it has already been executed. If this is the case, the corresponding inputs must be available, and therefore, we can copy them into the local copy of the CBS entry. If it is not the case, we call a function for the corresponding node that writes a set of default values to the local copy.
2. *Execute task*: The task is executed with the previously initialized inputs. Thereby, it checks safety conditions as described above, e.g., if the divisor of a division is non-zero. In case that such a safety-condition is not fulfilled, the execution of the task and the remaining speculation are canceled because the actual inputs will result in non-sensitive output values.
3. *Store speculation result*: If the task execution successfully terminates, i.e., it is not aborted by a safety condition as described above, the result is put into the corresponding speculation buffer. The content of this buffer is evaluated at the beginning of the non-speculative execution of the task.

4.4 Smart Task Selection for Speculation

This section describes a dynamic approach to select a task for efficient speculation. We will explain how the selection of a task for speculation influences the hit ratio of the overall speculation. Using the following approach, we are able to dynamically adapt the task selection until a user-defined hit ratio in the speculative execution is achieved.

The selection of a task for speculation has to consider the time (row in the CBS) and the node (column in the CBS). The time refers to the distance to the CBS head, i.e., the number of iterations that an entry is ahead of the earliest CBS entry. The node simply determines for which node a speculation is started. Both parameters are independent of each other and may have an effect on the speculation hit ratio.

Each node has different inputs that may be more or less suited for speculation. For instance, input variables that often have the same value are more suited than variables that are likely to change. Depending on the characteristics of a node's input variables, the hit ratio of speculations for this node may vary. For efficient speculation, it is reasonable to prefer nodes with a high speculation hit ratio.

The time and its effect in speculation is explained by the following thoughts: most SDFGs represent systems that have a state. This basically means that the outputs depend on the actual inputs and (directly or indirectly) on inputs of previous iterations. Hence, despite of the parallel and out-of-order execution, the execution of the tasks in the CBS usually underly a partial order. As a result, the CBS head entry will usually contain more executed/less pending tasks than the CBS tail. In turn, there will be a

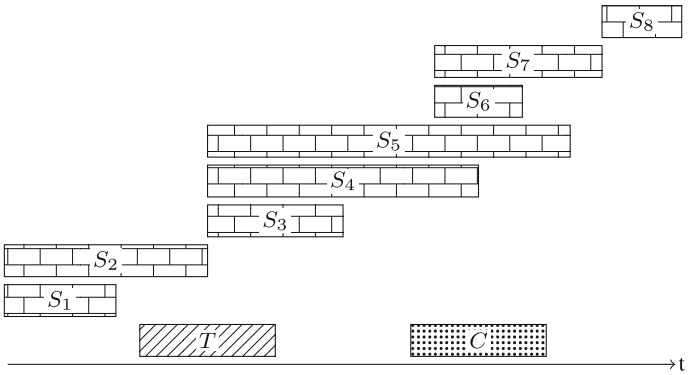


Fig. 11 Different timelines of a concurrent execution of a speculative task initialization (*boxes* denoted with S_1 to S_1) compared with the execution of the non-speculative execution of the same task (the *box* denoted with T) and the removal of an entry in the CBS (the *box* denoted with C). The left and right borders of *each box* represent the start and end times of that execution, respectively. This figure is used to elaborate the different timings and their related race conditions

gradient of known values in the CBS, where the CBS head entry will contain more known values than the CBS tail entry. The hit ratio of a speculation will be inversely proportional to the number of speculated values. Hence, doing a speculation for a task close to the CBS head will generally end up with a higher hit ratio due to more known input values. The more input values are known for a task, the higher is the probability that its actual execution is started in the near future and will overlap with the speculative execution. The actual execution will only check completed speculations (see timeline of $S_2, \ldots, S_8$ in Fig. 11). Results of concurrently running speculations will arrive too late, and therefore, they will be neglected. To this end, choosing a speculation close to the CBS head will generally result in a higher hit ratio but also in a higher probability of arriving to late. Choosing a speculation close to the CBS tail will end up with a low hit ratio and a high probability to be completed before the actual execution is started. The optimal range for the time parameter of a speculative execution depends on the program and the final hardware architecture.

Our approach evaluates statistic information and provides a solution to dynamically adapt the use of speculations by means of a user-defined parameter for the desired speculation hit ratio. For each node, we define a speculation window, consisting of a minimal and a maximum distance to the CBS head. This window is initialized with optimistic values, i.e., the range covers the complete CBS. A set of counters is used to keep track of the number of speculations, speculations that came too late, and bad speculations. During the execution, we regularly check after a defined number of speculations whether the observed numbers exceed their thresholds. In these cases, the speculation window of the corresponding node is reduced and the counters are reset to measure the hit-rate in the next interval. The speculation window will level out after a while, such that the hit ratio for each task will be above the user-defined parameter. Note that a window may shrink to zero size in case that too many speculations fail. A zero-sized window does not allow us to select a task. Hence, speculations for nodes with a zero-sized speculation window are stopped.

5 Weak Memory

Modern processors use weak memory models [45] to improve their performance, which give different cores different views on the memory accesses of other cores. Handling weak memory correctly is very subtle, and an extensive presentation of this topic is beyond the scope of this paper. In the following, we focus on the particular problems in our context and their solutions (the interested reader is referred to [45,59] for an overview on the topic).

First, consider a simple example. Assume that a particular core first updates a variable A and then it updates a variable B. In a sequential memory model, each core will see these updates in exactly the given order. In a weak memory model, in contrast, a core might see the change of B before A is updated in its cache.

Communication between threads requires sometimes to ensure that some memory accesses are performed in a specific order. This can be achieved using memory barriers (or also called fences). In general, there are three types of barriers: the store fence ensures that all changes are committed, before further changes are written. The load fence ensures that the cache is updated before any further memory reads are done. Finally, the memory fence combines the store and load fences.

In our implementation, most variables in the CBS require atomic read-and-modify access, e.g., fetching and decrementing a counter. Since most processors have at least an internal RISC behavior, their instruction set does not provide atomic read-and-modify operations. We rely on functions or structures from other libraries that provide the required functionality. The template class `tbb::atomic` from the Intel TBB library allows the instantiation of atomic variables. Amongst others, this class provides also the fetch-and-decrement action used in our implementation of worker threads (see line 8 of Fig. 7 and line 14 of Fig. 10). The corresponding functions are responsible for considering weak memory models, such that we do not have to add further operations for accessing these variables.

Access to variables of the system is done using native operations, i.e., without any special functions to ensure memory consistency. Within the execution of a single task, this is fine, since it is executed by one thread, which is assumed to have sequential memory consistency. Hence, consistency must be only considered, after a task has been finished and other tasks may be scheduled, which might be executed by other worker threads. In particular, a worker has to ensure that all changes of a task are committed before any succeeding task may read previously written variables. Therefore, a store fence is executed after the task execution. Conversely, a worker that removes a task description from the task queue must ensure that the system variables are updated before the task is executed. Therefore, a load fence is inserted before the task execution. The members "$iteration$" of the CBS and the variables $head$ and $tail$ require also explicit synchronization. Analogous changes to these variables have to be committed before any task is scheduled, and a store fence is put at corresponding places (see line 30 of Fig. 7). An update of these variables can be done after an element is removed from the task queue. Since there is already a load fence, no changes are necessary.

One further detail is left: a worker might remove a head while another worker executes a task. Since the latter worker might trigger a removal of the head with

outdated values in *head* and *tail*, it has to execute a load fence at the beginning of the head removal.

In our speculation, the communication especially relies on the pure memory communication without synchronization mechanisms. Hence, it is necessary to ensure memory consistency at certain points. For the speculation using the "Time Insensitive Check" (see Sect. 4.3), we extend the sequence for speculation as follows:

1. ***Load Fence*** (***line 24 in Fig.*** 10, ***optional***): We recommend to use a load fence before input values are copied. This may reduce bad speculations due to outdated input values.
2. *Create a copy of valid input values and speculate about missing input values*: While the input values are copied to a local buffer, new updates may be done by other threads. Periodic updates during the copying process may result in more up-to-date inputs, but also in lower performance due to time-consuming cache updates of the processor, which depends on the final target architecture. We left memory fences at this point for the latter reason.
3. *Execute task*: During the speculation, the algorithm works on a local copy of the system, which is not visible to other threads. Hence, the values will not change during the speculative execution. As a result, no memory consistency instructions are necessary during that time.
4. ***Store Fence (line 34 in Fig.*** 10): Before the result of the speculation is put into the speculation buffer, a store fence might be necessary depending on the actual implementation. In our case, we store pointers into the speculation buffer, which makes a store fence mandatory to ensure an update of the actual data.
5. *Store speculation result*

The existing out-of-order execution requires the following modifications:

- *Cleaning a CBS entry*: We must ensure that no data is reset after the timestamp has been updated. Hence, a store fence must be placed right before the timestamp is updated and after all other data of the CBS entry was updated (see line 23 of Fig. 7).
- *Check of speculation results*: When a speculation result is available, a token is taken from the corresponding buffer and checked. As already mentioned, our specific implementation does not put the complete structure containing the speculation result into the queue, but only a pointer. Hence, to ensure that the data at that address is up-to-date, we must call a load fence after the token has been taken (see line 41 of Fig. 10).

6 Results

6.1 Experimental Results

The presented approach was integrated into our existing out-of-order execution (OoO) [5]. The synthesis tool is part of our Averest[3] framework. The benchmarks for our experiments were written in the synchronous programming language Quartz [50].

[3] http://www.averest.org.

After compiling the programs to DPNs as described in [3], C code is generated from the DPN description. Synchronization and communication between cores is done using the Intel TBB library. Since the partitioning affects also the performance, we used the same partitioning for all variants of a benchmark. All the benchmarks were finally executed on an i5-750 with four physical/logical cores and a system with two Xeon X5450 with hyper-threading turned off, i.e., eight cores in overall. The created C programs were compiled with gcc 4.4.5 under Linux.

The OoO and OoOSpec execution are designed to process streams. Hence, most benchmarks have been designed as streaming applications. For instance, *MatrixMult* processes a larger number of matrices, while each multiplication is done in a single step. The *MatrixMultSeq* is an example from hardware design. To test such an implementation, one would feed its circuit with a series of different matrices. The *LU Decomposition* requires several iterations per matrix to compute the result. Similar to MatrixMult, it is fed with a sequence of input matrices. The size of the input and output matrix for these benchmarks is given in parentheses. *SquareRoot* computes the square root of a 1024 bit wide positive integer using a digit-by-digit calculation, which computes one bit of the result per iteration. *Pitchshift* is a practical DSP application, which modifies the pitch of a sound without changing the playback speed. The *audio delay* is a commonly used element to create for instance echo effects in audio processing. *DFT-IDFT* applies a discrete Fourier transform to an audio signal, i.e., a signal is transformed from time to frequency space. The resulting spectrum is immediately back-transformed using an inverse DFT. *Transform 3D* creates a transformation matrix to map three-dimensional vectors into two-dimensional screen coordinates. This is often necessary as a pre-transformation to visualize three-dimensional graphics. *Mandelbrot* is the computation of a fractal—the Mandelbrot set—which is named after Benoît Mandelbrot. The result is written as a 256×256 large picture. The landscape generator renders a three-dimensional landscape from a two-dimensional function, which serves as a height-field function of the landscape. The height of the picture is given as parameter H in parentheses. *Stochastic Ray Tracer* is a ray tracer that uses Monte Carlo methods to improve the image quality of a rendered scene, i.e., to make it look more realistic. Finally, *Modular Audio Synthesizer* is a set of audio processing modules, which are connected to create artificial sounds, i.e., a so-called synthesizer instrument. It provides a high amount of parallelism, due to its polyphonic implementation, i.e., several sounds can be played at the same time. Note that we added some benchmarks compared to the ones in [6]. For the sake of completeness, we add the speedups of the non-speculative OoO execution to our results.

Table 1 shows the speedups of our benchmarks. We start with considering the results that were made on the i5-750 system. In most cases, the OoO approach can speed up the execution compared to in-order execution. A strange effect can be seen in the "MatrixMult" and "Transform 3D" benchmark. Both achieve a speedup that is higher than the number of cores. Actually, we can only make an assumption on this: first, the data alignment may be better in the out-of-order execution, thereby enabling a better cache behavior in the processor. Second, although the in-order DPN execution is parallel, it is likely that the communication overhead is larger than the communication overhead of the OoO execution. Despite of the great result, we consider this as a success by accident, because the results are not reflected by the dual Xeon server.

Table 1 Experimental results for several benchmarks

Benchmark Name	Times and Speedup							
	i5-750 (4 cores)				2x Xeon X5450 (8 cores)			
	OoO *speedup to IO*		OoOSpec *speedup to OoO*		OoO *speedup to IO*		OoOSpec *speedup to OoO*	
	WS = 8	WS = 16	WS = 8	WS = 16	WS = 8	WS = 16	WS = 8	WS = 16
MatrixMult (16 × 16)	5.23	5.30	0.87	0.87	1.41	1.40	1.76	2.00
MatrixMult (32 × 32)	3.42	3.43	0.97	0.99	1.56	1.81	0.98	0.93
MatrixMultSeq (16 × 16)	1.20	1.20	1.67	1.67	1.12	1.12	0.43	0.43
MatrixMultSeq (32 × 32)	2.20	2.20	1.18	1.07	1.45	1.44	0.23	0.16
LU Decomp. (16 × 16)	1.07	0.74	0.89	2.29	0.92	0.68	0.75	0.60
LU Decomp. (32 × 32)	1.01	0.81	1.34	2.28	0.96	0.80	0.66	0.50
SquareRoot	1.29	1.27	2.11	2.12	1.11	1.14	0.69	0.55
Pitchshift	1.22	1.21	2.26	1.94	1.34	1.31	0.53	0.35
Audio Delay	1.27	1.27	1.32	0.63	1.07	1.05	0.32	0.32
DFT-IDFT	1.14	1.08	1.75	1.83	1.13	1.12	0.69	0.65
Transform 3D	8.21	8.98	1.02	0.98	1.79	1.80	0.95	1.39
Mandelbrot	1.20	1.20	0.90	0.88	0.99	0.99	0.48	0.37
Landscape Gen. ($H = 200$)	1.69	1.77	1.73	1.65	1.28	1.26	0.55	0.41
Landscape Gen. ($H = 800$)	2.74	2.70	1.90	1.57	1.24	1.22	0.66	0.57
Stochastic Ray Tracer	1.04	1.01	1.04	1.32	0.99	1.00	0.37	0.27
Modular Audio Synthesizer	1.34	1.35	1.58	1.39	1.04	1.03	0.33	0.19
Quadratic mean	2.91	3.05	1.46	1.58	1.23	1.24	0.75	0.77

Each column shows the speed up of the benchmarks for a specific execution type. OoO and OoOSpec, respectively, denote the out-of-order execution without and with speculation. The parameter *WS* gives the used window size for the benchmarks. The speedups of OoO execution are related to the in-order execution (IO) and the speedups of the OoOSpec execution are related to the OoO execution

Moreover, we used different window sizes for the benchmarks to analyze its effect on the execution time. On the one hand, a larger window size contains more tasks, and therefore, it promises more independent tasks, i.e., more parallelism. On the other hand, the additional memory requirements could result in a diminished cache performance of the processor. Only a few benchmarks have been affected by the given window sizes. While "MatrixMult (16×16)" and "Transform 3D" were able to gain a benefit of a larger window size, "LU Decomp." and "DFT-IDFT" are slowed down. In some additional preliminary benchmarks with a window size of 32 ($\mathrm{WS} = 32$), we observed that for most benchmarks the performance is decreased, which confirms the negative effect of a large window size on the cache behavior.

The OOO execution on the system with Xeon processors leads also to a speedup for most of the applications. Other benchmarks, in particular Mandelbrot and the ray tracer, are nearly unaffected by the out-of-order execution. The LU Decomposition is the only benchmark that is slowed down for a larger window size ($WS = 16$). In general, the speedups are smaller compared to the single processor quad core. This version of the Xeon processor communicates using the front-side bus, while newer versions use the faster QPI interconnect. Hence, the communication between these processors is quite expensive compared to on-chip communication. To this end, the slow inter-processor communication makes the scheduling of new tasks expensive and results in lower speedups.

In addition to the runtimes of the benchmarks, Tables 2 and 3 show the hit ratio statistics for the speculative execution. As already mentioned in Sect. 4.1.2, our language originates from hardware design and is based on the synchronous model of computation. Programs written in synchronous languages contain often a control part to ensure deterministic behavior. This control part is usually based on Boolean variables and is the key for the high hit ratio that was generally achieved.

The execution time of MatrixMult (16×16) with OoOSpec is slowed down compared to the execution time with the OoO approach. Despite of the high degree of parallelism, the hit ratio statistic shows that workers still come to the point where a speculation is triggered. For this benchmark, the number of speculations is relatively low and most of these speculations come to late. Hence, a small amount of the computational power is to no avail. Since the OoO approach already achieved a very high speedup, we did not consider the slightly slower execution as an issue. The larger variant of MatrixMult (32×32) is approximately as fast as the non-speculative variant. The low number of speculations implies that the utilization of the i5 seems to be better for this variant, which results in no significant change in the execution time.

The sequential matrix multiplication computes only one element of the target matrix per iteration. For that reason, this benchmark does not provide the same trivial parallelism as its parallel implementation. This results in more cores that can do speculative task executions. The hit rate for this application exceeds 50 % leading to a speedup of up to a factor 1.67. The larger variant of the sequential matrix multiplication (32×32) requires more memory per CBS entry and is likely to need more memory for the speculation results. Hence, the initialization phase of a speculation takes more time. As a result, more speculations are finished too late and the speedup turns out to be lower.

Table 2 Hit ratio for the OoOSpec executions on the i5-750 quad-core

Speculation Hit Ratio on i5-750					
Benchmark name	#Task executed		#Tasks speculated		
		$\sum$	Correct (%)	Too late (%)	Wrong (%)
MatrixMult (16 × 16) (WS = 8)	1900074	21591	40.01	55.80	4.19
MatrixMult (16 × 16) (WS = 16)	1900174	13978	39.04	59.65	1.31
MatrixMult (32 × 32) (WS = 8)	1600073	3458	81.58	17.93	0.49
MatrixMult (32 × 32) (WS = 16)	1600105	453	60.71	39.07	0.22
MatrixMultSeq (16 × 16) (WS = 8)	78665	9147	54.37	13.69	31.55
MatrixMultSeq (16 × 16) (WS = 16)	78677	9413	51.24	5.01	43.38
MatrixMultSeq (32 × 32) (WS = 8)	592469	38716	47.93	31.60	20.29
MatrixMultSeq (32 × 32) (WS = 16)	592493	41979	54.13	11.44	34.24
LU Decomp. (16 × 16) (WS = 8)	687153	50578	15.14	20.92	63.94
LU Decomp. (16 × 16) (WS = 16)	937366	344654	34.71	6.05	59.24
LU Decomp. (32 × 32) (WS = 8)	296681	68062	9.43	14.82	75.75
LU Decomp. (32 × 32) (WS = 16)	359097	122609	16.62	5.65	77.73
SquareRoot (WS = 8)	665789	141348	34.76	16.96	32.24
SquareRoot (WS = 16)	676067	133772	18.12	10.21	62.70
Pitchshift (WS = 8)	1400025	57006	11.00	55.42	33.58
Pitchshift (WS = 16)	1400047	74765	9.75	17.37	72.88
Audio Delay (WS = 8)	650077	17225	0.03	99.91	0.06
Audio Delay (WS = 16)	650152	50483	85.93	13.96	0.11
DFT-IDFT (WS = 8)	600030	311023	68.68	5.56	25.76
DFT-IDFT (WS = 16)	600033	313427	58.15	4.12	37.73

Table 2 continued

Speculation Hit Ratio on i5-750					
Benchmark name	#Task executed		#Tasks speculated		
		$\sum$	Correct (%)	Too late (%)	Wrong (%)
Transform 3D (WS = 8)	1900039	14188	69.81	12.57	17.62
Transform 3D (WS = 16)	1900055	163	32.52	3.07	64.42
Mandelbrot (WS = 8)	560712	210539	50.26	9.50	40.24
Mandelbrot (WS = 16)	560808	213387	34.13	7.90	57.97
Landscape Gen. (H = 200) (WS = 8)	892715	104973	33.45	24.64	41.56
Landscape Gen. (H = 200) (WS = 16)	892867	106755	31.29	7.28	60.77
Landscape Gen. (H = 800) (WS = 8)	6060560	76276	1.95	92.70	5.35
Landscape Gen. (H = 800) (WS = 16)	6060688	118669	4.31	80.78	14.91
Stochastic Ray Tracer (WS = 8)	963966	405247	36.34	5.05	57.98
Stochastic Ray Tracer (WS = 16)	968675	703405	9.84	1.42	86.90
Modular Audio Synthesizer (WS = 8)	3241397	267511	40.12	21.26	38.62
Modular Audio Synthesizer (WS = 16)	3241405	287311	5.76	21.24	73.00

Column "#Task Executed" denotes the number of actual tasks that have been executed. The remaining columns show the number of speculative executions. Percentages in parentheses denote "correct" speculations, "speculations" that have not been finished before the actual execution was started, and "wrong" speculations. The remaining percentages denote the speculations that were canceled due to invalid inputs

Table 3 Hit ratio for the OoOSpec executions on the 2x Xeon X5450

Speculation Hit Ratio on 2 Xeon X5450					
Benchmark Name	#Task Executed		#Tasks Speculated		
		$\sum$	Correct (%)	Too late (%)	Wrong (%)
MatrixMult (16 × 16) (WS = 8)	1900104	149843	20.99	7.11	71.89
MatrixMult (16 × 16) (WS = 16)	1900253	106139	13.07	3.84	83.10
MatrixMult (32 × 32) (WS = 8)	1600096	61162	30.92	19.63	49.45
MatrixMult (32 × 32) (WS = 16)	1600202	38437	20.24	12.74	67.03
MatrixMultSeq(16 × 16) (WS = 8)	78649	69066	57.57	0.12	41.93
MatrixMultSeq(16 × 16) (WS = 16)	78661	69169	51.48	0.02	48.13
MatrixMultSeq(32 × 32) (WS = 8)	592478	342071	71.55	7.01	21.30
MatrixMultSeq(32 × 32) (WS = 16)	592469	461565	62.85	1.38	35.58
LU decomp.(16 × 16) (WS = 8)	687141	169612	23.70	8.58	67.71
LU decomp.(16 × 16) (WS = 16)	936683	465280	36.03	2.83	61.13
LU decomp.(32 × 32) (WS = 8)	296633	67926	9.09	14.18	76.73
LU decomp.(32 × 32) (WS = 16)	358969	157325	14.65	3.91	81.44
SquareRoot (WS = 8)	666474	314702	42.09	6.19	36.64
SquareRoot (WS = 16)	676544	482778	24.60	3.91	61.62
Pitchshift (WS = 8)	1400033	326856	39.32	15.57	45.11
Pitchshift (WS = 16)	1400033	603484	21.08	7.49	71.43
Audio Delay (WS = 8)	650078	152720	44.39	16.05	39.56
Audio Delay (WS = 16)	650151	150232	51.97	9.47	38.57
DFT-IDFT (WS = 8)	600025	426396	66.74	2.47	30.79
DFT-IDFT (WS = 16)	600039	458389	48.51	0.91	50.58

Table 3 continued

Speculation Hit Ratio on 2 Xeon X5450					
Benchmark Name	#Task Executed		#Tasks Speculated		
		$\sum$	Correct (%)	Too late (%)	Wrong (%)
Transform 3D (WS = 8)	1900025	889962	49.32	2.71	47.98
Transform 3D (WS = 16)	1900144	511007	31.06	2.13	66.81
Mandelbrot (WS = 8)	560712	313647	51.70	6.00	42.30
Mandelbrot (WS = 16)	560808	433315	29.24	1.49	69.27
Landscape Gen. (H = 200) (WS = 8)	892715	270704	35.60	18.38	45.64
Landscape Gen. (H = 200) (WS = 16)	892867	365640	31.47	5.22	62.62
Landscape Gen. (H = 800) (WS = 8)	6060560	285463	22.21	31.59	45.80
Landscape Gen. (H = 800) (WS = 16)	6060688	382822	19.18	26.55	53.83
Stochastic Ray Tracer (WS = 8)	968660	513951	30.50	5.36	63.35
Stochastic Ray Tracer (WS = 16)	968657	740002	12.77	1.04	84.41
Modular Audio Synthesizer (WS = 8)	3241403	760764	33.61	11.24	55.15
Modular Audio Synthesizer (WS = 16)	3241577	1646991	11.54	5.66	82.80

See Table 2 for explanation of keys

The LU decomposition gains are speedup close to 2.3 for the execution with the larger window size ($WS = 16$). The speculation results in more hits compared to the execution in a smaller CBS ($WS = 8$). Since we applied an automatic partitioning, the generated task functions of this benchmark are different for the different benchmark sizes, i.e., the matrix size. For that reason, we get different behaviors of the speculative execution, which results in different speedups for a smaller window size.

SquareRoot has a large control part, which basically allows only a few possibilities per iteration depending on the input for this benchmark. Hence, speculative execution can achieve a good hit ratio, which results in a high speed up. The previously mentioned possibilities propagate with each iteration, which results in a lower hit ratio for the larger window size. At that point, it is unclear why the speedup remains the same as for the smaller window size.

The Pitchshift strongly depends on its input data and speculation results in a high miss rate. As a result, the smart task selection iteratively reduces the interval for speculative task selection to zero. The speculation is finally deactivated, which leads to a low count of speculative task executions. Hence, one would expect a speedup close to 1.0. At that point, it is unclear why the execution using the speculative approach results in a speedup of 2.26 and 1.94.

Audio Delay shows a similar effect. The speculation in a smaller CBS ($WS = 8$) gains a speedup of 1.32 despite of the low count of speculative task executions and the low hit rate. In contrast, the speculative execution in the larger CBS ($WS = 16$) ends up with a slower execution, although the speculation achieves a high hit rate. This benchmark is quite memory expensive. We assume that the initialization phase and the additional memory required by the larger CBS ($WS = 16$) lead to a poor cache behavior.

The DFT-IDFT allows a high degree of parallelism in each iteration, but not across the iterations. Therefore, the out-of-order execution is not able to significantly accelerate the in-order execution. Nevertheless, the control flow for this application is trivial and can be perfectly speculated. The speculative execution achieves a high count of speculatively executed tasks with a high hit ratio, which leads to a speedup of 1.75 and more.

The parallelism exhibited by Transform 3D is already exploited by the non-speculative out-of-order execution. Hence, no idle worker remain to do speculations. As a result, this application gains no acceleration and stays close to the runtime of the out-of-order execution.

The speculation in Mandelbrot is quite successful regarding the hit rate. Nevertheless, the initialization phase to start a speculation is quite expensive regarding memory bandwidth. Although the results of the speculations are computed in time, i.e., not too late, the allocation and copy process seem to need additional bandwidth. Thus, the speedup for the speculation is decreased by 10 %, yet, it remains above the in-order execution.

The speculative execution of the smaller variant of Landscape Generator ($H = 200$) results in a high amount of speculative task execution with a high hit ratio. As a result, the speedup turns out to be 1.73 ($WS = 8$) and 1.65 ($WS = 16$), respectively. Despite of the low amount of speculative task executions in the larger variant ($H = 800$), we still get perceptible speedups. We believe that the reason for this effect is the same as for the effect that appears in the execution of the Pitchshift benchmark.

For the ray tracer benchmark, we get a very high amount of speculative task executions. While the execution with a small window size results in a stagnant speedup, the execution with ($WS = 16$) allows an acceleration by a factor of 1.32.

Finally, the parallelism provided by the Audio Synthesizer allows only a few speculative task executions. While the execution with a smaller window size resulted in a high hit ratio, the execution in a larger window size resulted only in a small hit ratio. Nevertheless, our approach is able to accelerate this benchmark for both window sizes.

On the i5 system, most benchmarks can be further sped up using our speculation approach. The average speedup compared to the out-of-order execution is 1.46 ($WS = 8$) and 1.58 ($WS = 16$). Except for Mandelbrot and MatrixMult (16×16), the applications are either sped up or stay close to the execution time of the out-of-order approach. In contrast, the execution on the Xeon system results in a slow down for nearly all benchmarks. As already mentioned, the inter-processor connection is slower than the one of the i5. Although a speculation is only triggered for idle cores, the speculative execution of a task still requires memory bandwidth to initialize a structure to store its results. The additional effort to initialize this structure impairs the concurrent execution of actual tasks. We assume that this effect is exaggerated by the higher number of cores that can speculate in the Xeon system. Moreover, this system shows that the hit rate for speculative execution is not a guarantee for a speedup. Handling these issues remains to be done, for instance by limiting the number of workers that are allowed to execute speculative tasks.

For some benchmarks, the speculative execution is faster despite of the fact that only a few speculative task executions were done. This effect can not be explained, yet. To ensure correctness of the numbers, respective benchmarks have been double-checked. We repeated the experiments at the example of the landscape generator to replicate the measured values. Additionally, the rendered pictures of both variants have been compared to check the presence of a bug in the execution. Finally, the execution on another quad core system (AMD Phenom II 965) showed similar behavior, i.e., the speculative variant runs perceptible faster despite of a low number of speculative task executions.

We believe that the performance of the speculation approach may be generally increased by taking locally cached data into account. In particular, if we had knowledge about the cached data in a processor, we could consider this information in the task selection algorithm. Selecting tasks for speculation whose data is already in cache would eventually reduce accesses to the shared memory. In turn, this would leave more memory bandwidth to actual task executions.

In general, the hit ratio statistics for the benchmarks show that the hit ratio is quite high (see Tables 2, 3). Note that this is not due to good speculation about input values, but due to dynamically adapting the selection of tasks, which prefers tasks with a high hit ratio. The targeted hit ratio is controlled by two threshold values as explained in Sect. 4.4. The adaptive selection always starts with an optimistic initialization, i.e., tasks for speculation are taken from the complete CBS. The adaption is done using speculations and checks of speculations. Hence, the more speculations are made for an application, the higher the hit ratio becomes. This can be observed for many benchmarks, e.g., for DFT-IDFT and the Mandelbrot.

An expected, but still interesting observation is the relationship between window size and hit ratio. A larger window size allows one to do speculations in a more distant future. As explained in Sect. 4.4, the number of known values in the CBS decreases with the distance to the CBS head. Hence, the probability of bad speculations becomes then higher. This is confirmed by all benchmarks on both systems. Moreover, an inverted effect can be observed for speculations that come too late. The probability density of selecting a task for speculation which will be too late depends on the application, the node and the distance to the CBS head. The selection algorithm, which selects a task for speculation, is based on a random selection and has a uniform probability distribution in the range of the CBS size. As a result, if the window size of the CBS is increased, the lower the probability becomes that a task is selected that comes too late, and the higher the probability becomes that input values for a task are missing, i.e., the risk for bad speculation rises.

The re-usability of speculations as described in Sect. 4.3 has not been implemented, yet. This work and its analysis remains to be done. Moreover, an integration into the smart task selection (see Sect. 4.4) is desirable, because it will continue the idea of dynamic execution and adaption.

To this end, we want to emphasize that all benchmarks have been executed with the same parameters. There was no manual optimization or adjustment of parameters to improve the performance. Hence, there is a high probability to obtain better results with other parameters. Available parameters are e.g., the window size, the adaption speed and the thresholds for the task selection.

The framework for speculative out-of-order execution (OoOSpec) of DPNs in software still provides potential for optimizations. We ignored these optimizations so far, because the framework is considered as a starting point for further research. Hence, it should be kept readable and simple to enable easy integration of future ideas. As a result, the benchmark results should not be considered as an upper limit for our approach.

6.2 Execution Time Analysis

The advantage of this approach is its highly dynamic adaption to available hardware, which is however a disadvantage for worst and average case execution time estimations. Since there are no assumptions about the hardware and especially about its performance, a precise execution time cannot be determined. The speculation complicates to predetermine the execution time, because its benefits with respect to the performance generally depend on the actual input values. A general worst case execution time can be easily determined by considering a SDFG that only allows sequential execution of its tasks. In addition, if all speculations fail, the final execution time will correspond to the sequential execution time of all tasks including the overhead for scheduling and speculation checks (which depends on the final hardware).

Our approach still has a broad range of applications, where the average execution time is of more interest than the worst case execution time. For instance, multimedia applications have deadlines, but no hard deadlines like in safety-critical applications. Moreover, our Averest framework allows us to develop hardware, and one would expect simulations in software to be as fast as possible.

To this end, this approach has to be considered as a generic throughput optimization for systems that do not have to comply with hard deadlines.

7 Discussion

This section discusses several variations of some methods in our approach.

7.1 Least Effort Check

This section discusses a different way to check the results of speculations. Although it has not been implemented, we consider it as an interesting alternative to the "time-insensitive check" in Sect. 4.3.

The critical point in the check for correct speculations are potential races in the concurrent execution of a speculated task and its non-speculative antagonist. A speculation is always initiated before its non-speculative antagonist. However, this is not guaranteed for its completion, i.e., the non-speculative execution of a task may overtake the speculation. Hence, the result of this speculation may be put into the speculation queue after the non-speculative execution of this task already has been evaluated and cleared its speculation queue.

Adding a mechanism to prevent *old* speculation results from being added to the speculation queue requires some steps in the algorithm to be atomic. As a consequence, this will end up in adding synchronization, which would typically slow down the execution. In general, the execution of a speculative task should introduce as few changes as possible to the existing out-of-order execution. The worst case, i.e., if all speculations fail, should be that the performance of the speculative out-of-order execution is left unchanged compared to its non-speculative version. Hence, the execution of a speculative task should be as non-communicative as possible to avoid expensive synchronization overhead. As a result, there is no global knowledge about speculations that are currently in progress, i.e., no worker but the one that executes the speculative task knows about a running speculation. An alternative is to implement a mechanism into the speculation to safely recognize invalid speculations.

Adding a time-stamp to allow the association of speculations to an iteration seems to solve this issue. Basically, it would allow a fast way to check whether the speculation belongs to the currently checked iteration. Nevertheless, this is a very subtle solution since it leads to potential races.

Figure 11 shows an execution that can appear when a speculative task is executed. The potential speculative task executions are denoted with S_1 to S_1. Races appear when the initialization of the speculation, particularly copying the available inputs, overlaps with the non-speculative execution (denoted by T) or if the CBS entry that the speculation belongs to is removed (denoted by C). The best case that can happen is S_1, which simply means that the speculation is completed before the non-speculative execution is started. Case S_2 shows a start of the speculation initialization before the non-speculative task is scheduled, but the speculation is not completed in time, such that the result is useless. A check in the speculative task execution to avoid adding out-dated speculation results to the speculation queue, or clearing the

speculation queue in the head-removal step is not sufficient, because both methods cannot be implemented atomically without additional synchronization (which still should be avoided). As a result, when speculation results are checked, the corresponding worker might access out-dated results, and therefore, it has to have the ability to recognize them. As already stated, using iteration indexes as timestamps provides this ability.

The key is to make a consistent copy of the available inputs and to identify the iteration to which the copied input values belong to. A worker that is going to execute a speculative task might be interrupted during the copying process of the input values (e.g., case S_4 and S_5 in Fig. 11). While the worker is postponed, the actual execution might take place. To make it worse, the CBS entry might be cleared before the speculative task is continued, i.e., the inputs are generally invalidated. When the worker which executes the speculative task is waked up to proceed its calculations, it accesses invalid input values. As a result, its copy of the input values may contain values from different iterations, thereby making that copy inconsistent and invalid.

We extend the previous sequence to do a speculation in two steps:

1. *Create timestamp*: The behavior of a SDFG can be split into logical reaction steps. The index of the reaction step is stored in the CBS and copied to the speculation structure. This serves as the unique timestamp of the CBS entry that is handled. Whenever the entry of the CBS is cleared, this index is set to a new value and allows us to identify whether the speculation was overtaken by the non-speculative execution and the cleaning of the CBS entry.
2. *Create a copy of valid input values and speculate on missing input values*: remains as described before.
3. *Check timestamp*: At this step, we read again the unique timestamp from the CBS entry and compare it with the local copy. In case that they are not equal, the speculation has been overtaken by the cleaning process of the CBS entry. The speculative process can be stopped, because its result will eventually be neglected. If the timestamps are equal, the copies of the input values are either valid, making the speculation result valid, or—if the cleaning of the CBS entry has been started, but not finished—the inputs might be invalid (see S_4 and S_6 in Fig. 11). The latter case will still result in a correct execution, because the non-speculative execution of this task with exactly this timestamp must have already been done. Future task executions of the same node will have different timestamps and consider the result as outdated. Thus, they will neglect the result.
4. *Execute task*: remains as described before.
5. *Store speculation result*: remains as described before.

The existing out-of-order execution requires the following changes: when the CBS head is removed, the actual entry is reused by resetting its contents. To ensure a correct execution in the speculation approach, the update of the timestamp has to be finally done, i.e., after all other members of the CBS entry have been reset.

In general, timestamps require an infinite range of values, which is not available on computers. A state-of-the-art implementation for timestamps is to simply use a counter. Overflows of such a counter ("wrapping") can cause an incorrect execution, because it makes timestamps reappearing and therefore non-distinguishable. Increasing the

value range of a counter, i.e., its bit-width, will decrease the probability of such faults, but it does not solve the issue. Complete correctness is only provided by methods that can be reduced to the following idea: the value range is split into subranges. When the global timestamp shifts from one subrange to another one, a synchronization mechanism ensures that all old timestamps are removed, e.g., by completing all tasks that use old timestamps. The use of timestamps was motivated to avoid synchronization for performance reasons. Hence, solving problems related with timestamps by synchronization is pointless. Since correctness without additional synchronization is however not possible, this approach has not been implemented.

The advantage of this approach is its lightweight effort to compare speculation results, while non-reusability and timestamp issues define its downside. Extensions to handle weak memory models differ to the modifications for the time insensitive approach. Fences have to be added as described in the following:

1. ***Load Fence (optional)***: A load fence at the beginning of the speculation process is strongly recommended for two reasons: Similar to the previous method, an update may result in fewer wrong speculations due to outdated input values. In addition, it may be mandatory depending on a solution for the timestamp issue. A load fence at the beginning may end up with a lower performance, but not with incorrect executions.
2. *Create timestamp*
3. ***Load Fence***: After the timestamp was copied, a load fence is mandatory, because an update of the required inputs may have been done by another thread before the timestamp was copied (and after its optional load fence).
4. *Create a copy of valid input values and speculate missing input values*: The least effort check requires to correctly copy values from already executed tasks to the local buffer. Considering the pseudo-code given in line 24 of Fig. 10, there is a potential risk that $executed.n_P$ has been set after the last load fence. Hence, the copy function in line 25 might access outdated input values. To ensure that they are updated, it is mandatory to call a load fence after checking the $executed.n_P$ flag and right before copying the input values. In particular, we must add a $LoadFence()$ between line 24 and line 25 in Fig. 10.
5. ***Load Fence***: In the next step, we need an up-to-date timestamp, making it necessary to call again a load fence.
6. *Check timestamp*
7. *Execute task*: (identical to "Time Insensitive Check" approach, see Sect. 4.3)
8. ***Store Fence***: (identical to "Time Insensitive Check" approach, see Sect. 4.3)
9. *Store speculation result*

Modifications to the out-of-order execution are identical to the modification for the time insensitive approach (see Sect. 5).

7.2 Energy Consumption

Another important aspect concerning speculation is the increase of energy consumption. While correct speculations can speedup the execution of a program, wrong speculations will be simply neglected without any performance benefits. As a consequence,

the energy that was used to execute such a speculation was wasted. Hence, the presented approach is *not* intended to be used in low-energy systems, but in high-performance systems. Despite of the fact that the logic for speculations requires more energy, speculation techniques are still commonly used in desktop and server PCs, containing e.g., Intel x86(-64) and AMD processors. Considering energy consumption, Intel's VLIW architectures, i.e., the Intel Itanium, would have been definitely the more reasonable solution. Nevertheless, dynamic processors with speculation still dominate the market. To this end, the decision to use the speculative approach depends on whether performance or energy consumption has the priority. Independently, our software solution provides an invincible advantage compared to the hardware solution: our out-of-order execution is scalable (window size can be set at the beginning of the program) and its speculation can be switched on and off. Beyond that, the task selection in our speculation dynamically adapts to a desired hit ratio. To conclude, our software solution allows much more customization compared to hardware solutions.

7.3 Multiple Speculations Per Task

The necessity to store speculated values in a structure which is separated from the CBS allows us to use more than one speculation with different values for the same task. A similar technique is predicated execution and can be found in some VLIW architectures like the Intel Itanium. The predicated execution is basically the execution of an instruction that depends on a condition—the predicate. The result of the instruction is calculated, even if the predicate has not been evaluated, yet. The result is held back until the value of the predicate becomes known. An effective implementation requires a speculation that provides varying values which is not yet available.

7.4 Caching Speculation Results

In our implementation, we use the time insensitive check as explained in Sect. 4.3. After a check for the correctness of a speculation has been completed, the speculation result is removed from the speculation buffer. However, the time insensitive check allows us to keep the result in the buffer and to reuse it for later computations, i.e., we could use it as a speculation result of other iterations of the same node. Moreover, we could also keep results of actual node executions as speculation results in the speculation buffer. In particular, this idea is based on Richardson's approach in [55], which uses a result cache to speedup time-consuming functions with frequently used input sets. Note that Richardson's approach did not consider speculation at all. For timing issues, we had to postpone this idea.

8 Conclusions

In this paper, we considered how one can speedup the execution of DPNs by techniques that are already used in processor architectures. In particular, we considered out-of-order execution that allows nodes to compute next reactions before the current one as long as the results are finally given to the environment in the correct order.

Moreover, we considered in detail how speculation which is a widely used technique in processor architectures, e.g. for branch prediction, can be used to predict input values so that nodes can be executed before the real input values are available. Despite the simplicity of the implementation, we are able to improve the performance of most benchmarks on a single-chip multicore processor system. Further research on the reduction of bandwidth in our approach remains to be done to improve performance in manycore systems. While most related work is based on specialized hardware, our approach is completely implemented in software, thereby addressing a broad field of multicore processors. Moreover, our techniques are completely automated, i.e., no manual work has to be done by the programmer. As a main feature of our approach, we emphasize the flexibility in terms of the number of processing units (e.g., cpus, cores), i.e., there is no need to repartition a given DPN when the target system is extended or changed.

References

1. Allen, J. (ed.): Software Synthesis from Dataflow Graphs. Kluwer, Dordrecht (1996)
2. Arvind, Nikhil, R.: Executing a program on the MIT tagged-token dataflow architecture. IEEE Trans. Comput. (T-C) **39**(3), 300–318 (1990)
3. Balakrishnan, S., Sohi, G.: Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In: International Symposium on Computer Architecture (ISCA), pp. 302–313. IEEE Computer Society, Boston, Massachusetts, USA (2006)
4. Baudisch, D., Brandt, J., Schneider, K.: Dependency-driven distribution of synchronous programs. In: Hinchey, M., Kleinjohann, B., Kleinjohann, L., Lindsay, P., Rammig, F., Wolf, M. (eds.) Distributed and Parallel Embedded Systems (DIPES), pp. 169–180. International Federation for Information Processing (IFIP), Brisbane, Queensland, Australia (2010)
5. Baudisch, D., Brandt, J., Schneider, K.: Translating synchronous systems to data-flow process networks. In: Yeo, S.S., Vaidya, B., Papadopoulos, G. (eds.) Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 354–361. IEEE Computer Society, Gwangju, Korea (2011)
6. Baudisch, D., Brandt, J., Schneider, K.: Out-of-order execution of synchronous data-flow networks. In: McAllister, J., Bhattacharyya, S. (eds.) International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS), pp. 168–175. IEEE Computer Society, Samos, Greece (2012)
7. Bhattacharyya, S., Brebner, G., Janneck, J., Eker, J., von Platen, C., Mattavelli, M., Raulet, M.: OpenDF-a dataflow toolset for reconfigurable hardware and multicore systems. ACM SIGARCH Comput. Archit. News **36**(5), 29–35 (2009)
8. Bhattacharyya, S., Lee, E.: Scheduling synchronous dataflow graphs for efficient looping. J. VLSI Sig. Process. **6**(3), 271–288 (1992)
9. Bhattacharyya, S., Lee, E.: Looped schedules for dataflow descriptions of multirate signal processing algorithms. Formal Methods Syst. Des. **5**(3), 183–205 (1994)
10. Böhm, A., Oldehoeft, R., Cann, D., Feo, J.: SISAL 2.0 Reference Manual. Technical Report CS-91-118, Computer Science Department of Colorado State University (1991)
11. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cyclo-static dataflow. IEEE Trans. Sig. Process. **44**(2), 397–408 (1996)
12. Bonfietti, A., Benini, L., Lombardi, M., Milano, M.: An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms. Design, Automation and Test in Europe (DATE), pp. 897–902. EDA Consortium, Dresden, Germany (2010)
13. Buck, J., Lee, E.: The token flow model. In: Bic, L., Gao, G., Gaudiot, J.L. (eds.) Advanced Topics in Dataflow Computing and Multithreading, pp. 267–290. IEEE Computer Society, Hamilton Island, Queensland, Australia (1995)
14. Cintra, M., Martínez, J., Torrellas, J.: Architectural support for scalable speculative parallelization in shared-memory multiprocessors. International Symposium on Computer Architecture (ISCA), pp. 13–24. ACM, Vancouver, British Columbia, Canada (2000)

15. Colohan, C., Ailamaki, A., Steffan, J., Mowry, T.: CMP support for large and dependent speculative threads. IEEE Trans. Parallel Distrib. Syst. **18**(8), 1041–1054 (2007)
16. Colwell, R., Hall, W., Joshi, C., Papworth, D., Rodman, P., Tomes, J.: Architecture and implementation of a VLIW supercomputer. Supercomputing, pp. 910–919. IEEE Computer Society, New York, NY, USA (1990)
17. Dennis, J.: Data flow supercomputers. IEEE. Comput. **13**(11), 48–56 (1980)
18. Dennis, J., Misunas, D.: A preliminary architecture for a basic data-flow processor. 25 Years of the International Symposia on Computer Architecture (ISCA), pp. 125–131. ACM, Barcelona, Spain (1998)
19. Dennis, J., Misunas, D., Thiagarajan, P.: Data-flow computer architecture. Technical Report CSG-MEMO 104, MIT Lab for Computer Science, Cambridge, Massachusetts, USA (1974)
20. Engels, M., Bilsen, G., Lauwereins, R., Peperstraete, J.: Cyclo-static dataflow: Model and implementation. In: Asilomar Conference on Signals, Systems and Computers (ACSSC). IEEE Computer Society, Pacific Grove, California, USA (1994)
21. Fisher, J., Faraboschi, P., Young, C.: Embedded Computing: A VLIW Approach to Architecture. Compilers and Tools. Morgan Kaufmann, San Francisco (2005)
22. Gao, G., Govindarajan, R., Panangaden, P.: Well-behaved programs for DSP computation. International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 561–564. IEEE Computer Society, San Francisco, California, USA (1992)
23. Genin, D., De Moortel, J., Desmet, D., van de Velde, E.: System design, optimization, and intelligent code generation for standard digital signal processors. International Symposium on Circuits and Systems (ISCAS), pp. 565–569. IEEE Computer Society, Portland, Oregon, USA (1989)
24. Hammond, L., Willey, M., Olukotun, K.: Data speculation support for a chip multiprocessor. In: Bhandarkar, D., Agarwal, A. (eds.) Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 58–69. ACM, San Jose, CA, USA (1998)
25. Janneck, J., Miller, I., Parlour, D., Roquier, G., Wipliez, M., Raulet, M.: Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. Signal Processing Systems (SiPS), pp. 287–292. IEEE Computer Society, Washington, District of Columbia, USA (2008)
26. Johnson, T., Eigenmann, R., Vijaykumar, T.: Min cut program decomposition for thread level speculation. In: Chambers, C. (ed.) Programming Language Design and Implementation (PLDI), pp. 59–70. ACM, Washington, DC, USA (2004)
27. Johnston, W., Hanna, J., Millar, R.: Advances in dataflow programming languages. ACM Comput. Surv. (CSUR) **36**(1), 1–34 (2004)
28. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J. (ed.) Information Processing, pp. 471–475. North-Holland, Stockholm, Sweden (1974)
29. Kazi, I., Lilja, D.: Coarse-grained thread pipelining—a speculative parallel execution model for shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst **12**(9), 952–966 (2001)
30. Le Mentec, F., Gautier, T., Danjean, V.: The X-Kaapi's application programming interface. part I: Data flow programming. Technical Report RT-0418, Institut National de Recherche en Informatique et en Automatique (INRIA) (2011)
31. Lee, B., Hurson, A.: Dataflow architectures and multithreading. IEEE. Comput. **27**(8), 27–39 (1994)
32. Lee, E.: Consistency in dataflow graphs. IEEE Trans. Parallel Distrib. Syst. 2(2) (1991)
33. Lee, E.: The problem with threads. IEEE. Comput. **39**(5), 33–42 (2006)
34. Lee, E.: Computing needs time. Commun. ACM (CACM) **52**(5), 70–79 (2009)
35. Lee, E., Ha, S.: Scheduling strategies for multiprocessor real-time DSP. In: Global Telecommunications Conference (GLOBECOM), pp. 1279–1283. IEEE Computer Society (1989)
36. Lee, E., Messerschmitt, D.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**(1), 24–35 (1987)
37. Lee, E., Messerschmitt, D.: Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987)
38. Lee, E., Parks, T.: Dataflow process networks. Proc. IEEE **83**(5), 773–801 (1995)
39. Lilja, D.: Reducing the branch penalty in pipelined processors. IEEE Comput. **21**(7), 47–55 (1988)
40. Lipasti, M., Shen, J.: Exceeding the dataflow limit via value prediction. Microarchitecture (MICRO), pp. 226–237. IEEE Computer Society, Paris, France (1996)
41. Madriles, C., López, P., Codina, J., Gibert, E., Latorre, F., Martínez, A., Martínez, R., González, A.: Boosting single-thread performance in multi-core systems through fine-grain multi-threading. In: Keckler, S., Barroso, L. (eds.) International Symposium on Computer Architecture (ISCA), pp. 474–483. ACM, Austin, TX, USA (2009)

42. Marcuello, P., González, A.: Exploiting speculative thread-level parallelism on a SMT processor. In: Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B. (eds.) International Conference on High-Performance Computing and Networking (HPCN), LNCS, vol. 1593, pp. 754–763. Springer, Amsterdam, The Netherlands (1999)
43. Marcuello, P., González, A., Tubella, J.: Thread partitioning and value prediction for exploiting speculative thread-level parallelism. IEEE Trans. Comput. **53**(2), 114–125 (2004)
44. McGraw, J.: The VAL language: description and analysis. ACM Trans. Program. Lang. Syst. **4**(1), 44–82 (1982)
45. McKenney, P.: Memory barriers: A hardware view for software hackers. http://www.rdrop.com/users/paulmck (2010)
46. Moshovos, A., Breach, S., Vijaykumar, T., Sohi, G.: Dynamic speculation and synchronization of data dependences. In: International Symposium on Computer Architecture (ISCA), pp. 181–193 (1997)
47. Murthy, P., Bhattacharyya, S., Lee, E.: Joint minimization of code and data for synchronous dataflow programs. Formal Methods Syst. Des. **11**(1), 41–70 (1997)
48. Nikhil, R.: Dataflow Programming Languages. Technical Report CSG-MEMO 333, Computer Science and Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA (1991)
49. Pajuelo, A., González, A., Valero, M.: Speculative execution for hiding memory latency. In: MEmory Performance: DEaling with Applications, Systems and Architecture (MEDEA), pp. 49–56. ACM, Antibes Juan-les-Pins, France (2004)
50. Parks, T.: Bounded Scheduling of Process Networks. Ph.D. Thesis, Princeton University (1995)
51. Powell, D., Lee, E., Newmann, W.: Direct synthesis of optimized DSP assembly from signal flow diagrams. In: International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 553–556. IEEE Computer Society, San Francisco, California, USA (1992)
52. Pérez, J., Badia, R., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: International Conference on Cluster Computing (CLUSTER), pp. 142–151. IEEE Computer Society, Tsukuba, Japan (2008)
53. Ramamoorthy, C., Li, H.: Pipeline architecture. ACM Comput. Surv. **9**(1), 61–102 (1977)
54. Renau, J., Strauss, K., Ceze, L., Liu, W., Sarangi, S., Tuck, J., Torrellas, J.: Thread-level speculation on a CMP can be energy efficient. International Conference on Supercomputing (ICS), pp. 219–228. ACM, Cambridge, Massachusetts, USA (2005)
55. Richardson, S.: Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical Report SMLI TR-92-1, Sun Microsystems Inc., Mountain View, CA, USA (1992)
56. Roquier, G., Lucarz, C., Mattavelli, M., Wipliez, M., Raulet, M., Janneck, J., Miller, I., Parlour, D.: An integrated environment for HW/SW co-design based on a CAL specification and HW/SW code generators. In: International Symposium on Circuits and Systems (ISCAS), pp. 799–799. IEEE Computer Society, Taipei, Taiwan (2009)
57. Rumbaugh, J.: A data flow multiprocessor. IEEE Trans. Comput. **26**(2), 138–146 (1977)
58. Schneider, K.: The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany (2009)
59. Steinke, R., Nutt, G.: A unified theory of shared memory consistency. J. ACM **51**(5), 800–849 (2004)
60. Stulova, A., Leupers, R., Ascheid, G.: Throughput driven transformations of synchronous data flows for mapping to heterogeneous MPSoCs. In: McAllister, J., Bhattacharyya, S. (eds.) International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS), pp. 144–151. IEEE Computer Society, Samos, Greece (2012)
61. Tejedor, E., Farreras, M., Grove, D., Almasi, G., Labarta, J.: ClusterSs: a task-based programming model for clusters. In: High Performance Distributed Computing (HPDC), pp. 267–268. ACM, San Jose, CA, USA (2011)
62. Tomasulo, R.: An efficient algorithm for exploiting multiple arithmetic units. IBM J. Res. Dev. **11**(1), 25–33 (1967)
63. Vachharajani, N., Rangan, R., Raman, E., Bridges, M., Ottoni, G., August, D.: Speculative decoupled software pipelining. Parallel Architectures and Compilation Techniques (PACT), pp. 49–59. IEEE Computer Society, Brasov, Romania (2007)
64. Zilles, C., Sohi, G.: Master/slave speculative parallelization. Microarchitecture (MICRO), pp. 85–96. IEEE Computer Society, Istanbul, Turkey (2002)