Out-Of-Order Execution of Synchronous Data-Flow Networks

Daniel Baudisch, Jens Brandt and Klaus Schneider
Embedded Systems Group
Department of Computer Science
University of Kaiserslautern
http://es.cs.uni-kl.de

Abstract—Data flow process networks (DPNs) have been introduced as a convenient model of computation for distributed and asynchronous systems since each process node can work independently of the other nodes, i.e. without the need of a global coordination. Synchronous and cyclo-static data flow process networks even allow to derive at compile-time efficient static schedules that allow one to run these systems with an efficient use of available resources, e.g. in embedded systems. Single process nodes of DPNs are stream-based computing devices that transform input streams to uniquely defined corresponding output streams such that single values of the output streams are computed as soon as sufficient input values are available. In this sense, they are related to the execution of an instruction stream by a conventional microprocessor. In this paper, we show how out-of-order execution that has been introduced for the efficient use of multiple functional units in microprocessors can also be used for the implementation of DPNs on multiprocessors. This way, the implementation of DPNs on multiprocessors allows one to optimize the throughput of single process nodes, and as shown by our experiments, also of the entire DPN.

I. INTRODUCTION

Data-flow process networks (DPNs) [9, 21, 27, 30] 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 and to synchronize, 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 data values or reads data 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 hardware architectures [20, 34], as well as for the design of programming languages and libraries, e.g. OpenDF [5, 17, 33]. It does not demand the use of special programming languages, and instead, allows one to use traditional sequential programming languages for its implementation. 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 lead to the same outputs (independent of transmission delays) 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 subset of DPNs are deterministic and have decidable boundedness or liveness problems. Synchronous data-flow (SDF) networks [4, 6, 9, 21, 25–27] and cyclo-static data-flow networks [7, 12] are such restricted nets, which have become very successful, in particular, for the synthesis of signal processing systems [1, 6, 13, 14, 22, 23, 29, 31]. They are a special kind of DPNs where in each firing step, the 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 SDF 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, in principle, this class of DPNs can be implemented completely sequentially. However, in order to exploit the power of state-of-the-art multi-core processors, concurrent implementations are necessary to achieve the best performance.

On these concurrent systems, the scheduling of functions can be organized in different ways, which may be roughly categorized into static (at compile-time) and dynamic (at runtime) variants [24]. 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 other applications for the following reasons: First, as the schedule depends on the architecture, the dynamic variant is necessary for the transfer of compiled code from one machine to another one. Second, even with a fixed architecture, it is very hard to estimate the run-time of tasks in the context of other running processes and cache effects. Therefore, we focus on the fully dynamic variant in this paper which wraps the code for the individual nodes into tasks that are fed to a number of worker threads. Thereby, this number should match with the number
of available computing units to avoid additional overhead due to context switches.

In this paper, we show how out-of-order execution, which is a well-known technique from processor architecture, can be used in this context. In an unbalanced DPN, the values of some input buffers of a node may be produced faster than they were consumed by the node. Thus, it is reasonable to execute this node several times in parallel, leading to a better utilization of resources. Moreover, varying computational effort of single executions in this node will produce outputs that arrive out-of-order, enabling other nodes to fire out-of-order. This out-of-order execution avoids idle time and can speed up the program execution. As data is now reordered, the approach involves additional effort to reorder the correct flow of output data. While static scheduling of DPNs follows the philosophy of statically scheduled processors (like VLIW processors) [10], our OOO-DPNs refer to dynamic scheduling similar to processors with out-of-order execution.

The main contribution of this paper is an approach to generate multi-threaded code exploiting out-of-order execution for synchronous data-flow networks. Thereby, we target standard commercial processors, e.g., ones of the i86 and i86-64 architecture and do not rely on any particular hardware extension. We implemented our approach and evaluated it with the help of several case studies, which show the performance gains and overheads.

The rest of the paper is structured as follows: Section II gives an introduction to data-flow process networks and related work. Section III introduces a method to improve the multi-threaded execution of DPNs in several ways, i.e., load-balancing and additional flexibility in the provided parallelism. The feasibility of this method is demonstrated in Section IV with some benchmarks. Finally, Section V concludes the paper.

II. Preliminaries

A. Synchronous Data-Flow

According to Kahn [18], a data-flow process network (DPN) consists of several nodes that are connected by unbounded FIFO buffers. The behavior of the individual nodes of the DPN is often described by a set of so-called firing rules. 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 when the corresponding action is executed and how many tokens from the input buffers are read. Hence, the nodes of the DPN are triggered by the occurrence of input data values, and there is no global coordination like a global clock.

A special variant of DPNs are synchronous data-flow graphs (SDF) [26], where the number of tokens read from and written to a particular buffer is always constant. The left part of Figure 1 shows an example of an SDF 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 Figure 1.

![Figure 1. Left: SDF node with 2 inputs and 2 outputs. Right: Pseudo-code for left SDF node](image)

Obviously, there is a straightforward approach to create (multi-threaded) code from a SDF: each node is translated to a separate thread [3], and all threads communicate by simple FIFO queues with each other. To decouple the created threads, each thread runs an infinite loop which fires the rules, i.e., runs code similar to the one given on the right-hand side of Figure 1. Thereby, the scheduling is left to the operating system, which schedules the created threads according to the available input data and resources.

Apparently, the communication overhead of this solution may be significant – depending on the computation complexity of the nodes, threads maybe too heavyweight. For an efficient implementation, the regular communication behavior of SDFs is used. Purely sequential software (without any buffer synchronization) can be obtained by creating a static schedule of all nodes, i.e., a compiler precomputes the flow of tokens: when the input data is guaranteed to be present, the node can be executed without a previous check of its inputs.

In the 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 need to cooperate by explicit synchronization again. Furthermore, in order to fully exploit the parallelism of the target architecture, a compiler has to have knowledge about the computational effort of each node, which is in general a very hard problem. In particular, this is the case in the context of caches 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 software that dynamically adapts to its target architecture and the available resources.

B. Out-of-Order Execution in Microprocessors

Out-of-order execution is well-known in the domain of computer architecture for a long time, originating in early work by Tomasulo [37]. Its basic idea is to execute a sequential instruction stream of a usual von-Neumann architecture in data-flow 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 or not; if it is not, 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 operand registers and the 
tags in case that the operand is a target register that will be overwritten 
by another instruction. Instructions are loaded by the instruction 
decoder into the RS. Concurrently, the decoder loads available 
operand registers or sets 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 makes the data-flow driving the computation, and 
not the original control-flow given by the order of instructions. 
The ability of processors to be interrupted by external signals 
requires changes of the register set to be done in order. The 
reorder buffer serves as an intermediate buffer for finished calculations, which are applied in order to the register set. This preserves an external visible in-order behavior.

C. Related Work

In the context of SDF, several frameworks such as KAAPI [19] or StarSS [32, 36] exist, which ease the programming of 
multi-threaded implementations. Both provide libraries for the 
concurrent execution of nodes and the 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 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 of 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 the prevalently used 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 to 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 run-time: 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 SDF. 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 
data-flow has already been considered by Dennis and Misunas. 
In [11], they propose a custom-tailored hardware architecture 
to execute a specific light-weight format of SDFs. 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 proposed hardware architecture is inspired from general 
data-flow 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 SDF 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 units and operation units, handles the 
actual scheduling 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.

III. OUT-OF-ORDER EXECUTION

SDF has been successfully applied in many areas, in particular 
in digital signal processing. Usually, the systems continuously 
read an inputs from their environment, calculate the output 
values and update their states (with the help of the actual func-
tion T) and finally, write the output values to the environment. 
Figure 2 sketches this general scheme.

In contrast to reactive systems, applications from this do-
main often work in so-called bursts: they supply many inputs at 
onece, 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 itera-
tions and not only within a single iteration. In consequence, we 
obtain an out-of-order execution (with respect to iterations).

As Figure 2 shows, the state of the system S prevents a 
trivial parallel execution of several iterations. As the inputs are 
read in each step, they are responsible for the dependencies 
between different iterations. In the following Section III-A, we 
explain the basic idea of out-of-order execution of tasks, while 
Section III-B deals with a particular implementation issue, 
namely weak memory.

A. General Idea

a) SDF and Task Functions: Throughout this section, we use a running example, an SDF consisting of four nodes 
T₀, T₁, T₂ and T₃ (see SDF in Figure 3). The code of the 
individual nodes is given in the Task Functions section: when
Figure 3. Structures, data and pseudo-code for examplary SDF. 

Top-left: the examplary SDF. 
Top-right: the description of the SDF and functions to access members of the description. 
Middle-left: the function of the nodes and how to access input and output buffers, i.e. data in the CBS. 
Middle-right: the CBS after the initialization process and the task queue containing tasks that can be started. 
Bottom: Pseudo-code of the out-of-order scheduler.
this nodes is triggered, the given task is executed in the final implementation, i.e. the given function is applied to the set of variables belonging to the corresponding iteration.

Before we present our out-of-order execution, we first modify the given SDF so that the communication with the environment is accomplished by dedicated nodes \( I_0 \) and \( O_0 \) (as shown in the SDF 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.

b) SDF Description: As already explained above, our approach dynamically maintains a list of dependencies between task executions (which is defined by a tuple containing the SDF node and the iteration). We use the data structures given in the upper right part of Figure 3 (see SDF Description) to store these dependencies and the actual structure of the SDF graph.

c) Central Buffer Station: Due to the out-of-order execution of nodes of the SDF, 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 SDFs 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 decide to 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, it is organized as a ring buffer that provides access to at least \( WS \) iterations. \( WS \) is the number of newest iterations that can be concurrently handled by the scheduler and is 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 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 \( CBS_{size} = N_{max} + WS \) 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 \( (E_{head}) \) and the tail entry \( (E_{tail}) \) of the ring buffer address the oldest and the newest, respectively, iteration that are processed.

The read and write positions of a buffer in an iteration \( i \) are determined as follows: All nodes will always read the \( i \)th element, and therefore, the read position is \( \text{ReadPos}(B) = i \mod CBS_{size} \). 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 \( \text{WritePos}(B) = (i + B_{init}) \mod CBS_{size} \) (see pseudo-code of nodes in the middle-left part of Figure 3).

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 \( c_T[E] \) and initialize it with the number of incoming dependencies which can be obtained from the SDF description (see upper right part of Figure 3). Every time a task has been executed, the scheduler must decrement each counter of its successors. As soon as a counter reaches the value 0, the corresponding task must be scheduled. To keep track 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. Every time a task for iteration \( i \) has been executed, the counter \( RT[i \mod CBS_{size}] \) is decremented. When \( RT[E_{head}] \) reaches 0, the head can be removed. In practice, this means to reset the counters to their initial values, such that a new iteration can be processed. In particular, the counter \( RT[E_{head}] \) for each task \( T \) is initialized to the number of incoming dependencies of task \( T \), and the counter \( RT[E_{head}] \) is initialized to the number of tasks. An SDF may have source nodes, i.e. nodes that have no incoming dependencies (i.e. only outgoing dependencies), e.g. a node that reads inputs from an environment. For that reason, after an entry has been initialized for re-use, the corresponding thread must schedule all tasks where \( c_T[E] \) is 0.

d) Scheduler: 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 a race occurs is for most systems very low.

The reset of the 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 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 will overwrite the counters with wrong values. A separation of these phases, such that the reset of the head is enforced to be completed before any schedule is made, solves this problem.

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, i.e. the iteration that is addressed must correspond to the iteration that is considered in the addressed entry. To avoid the scheduling of tasks that would address non-available entries, their number of incoming dependencies is incremented by one. 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 \( N \) (the number of tokens of the buffer with the most initial tokens). It follows that, when the entry addressed by the largest offset gets available, every preceding entry must be also available. Conversely, every time the head is removed, for each task \( T \) an entry becomes available and the dependency counter of task \( T \) in entry \( E = (E_{\text{head}} - N_T) \mod \text{CBS.size} \) has to be decremented. Similar to the task check, which is done after a task has been executed and the counter of its successors have been decremented, the succeeding task \( T \) is scheduled when its dependency counter reaches 0.

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 but 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. After the head of the CBS has been removed and a new tail element has been inserted, we must decrement the dependency counter \( RT[(E_{\text{head}})] \) for the new head entry, i.e. after \( E_{\text{head}} \) has been incremented.

### B. Weak Memory

Modern processors use weak memory models [28] to improve their performance, which allow each core to access the shared memory out-of-order. 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 [28, 35] for an overview of 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.

Since modern architectures have different levels of weakness of their memory model, we require in the following at least a model with a sequential memory consistency within a single core, i.e. a single thread that is executed, sees its own changes immediately. Sometimes, communication between threads requires several memory accesses that require a specific order. This can be done 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 succeeding changes are written. The load fence ensures that the cache is updated before any succeeding 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 Worker Threads in Figure 3). 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 affected places (see bottom part of Figure 3). 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 remaining 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.

### IV. Experimental Results

The approach as described in the previous section has been implemented with the help of our Averest framework. The following benchmarks were written in the synchronous programming language Quartz and compiled to DPNs as described in [2]. The resulting DPN is synthesized to C code using the Intel TBB library. Thereby, we compare three methods. The first one corresponds to the naive approach (see Section II) which translates each node of the DPN to a single thread. The second version is a task-based approach that executes the nodes of the DPN in order. These versions are compared to the third version which implements the out-of-order scheduling described in Section III. For all variants, we use the same partitioning, i.e. each created DPN has 16 nodes. All the benchmarks were finally executed on two SMP multicore machines: an i5-750 with four physical/logical cores and a server equipped with two X5450 Xeon with eight physical/logical cores in total. The created C programs were compiled with gcc 4.4.5 under Linux.
The results of all benchmarks (see Figure 4) show that a task-based execution generally provides better results compared to the 1-node-1-thread approach (1n1t). In addition, the out-of-order execution speeds up the benchmarks by an average of 1.18 and a peak of 1.80. Benchmarks that provide no parallelism across iterations nearly keep the runtime of in-order execution. In the following, we discuss the results for each benchmark w.r.t. to its characteristics. In general, the i5 achieves better results compared to the Xeon system. One can suppose that the communication between cores on the i5 is better implemented than on the Xeon.

MatrixMult (see Figure 4) iteratively takes two matrices as inputs and performs a matrix multiplication. As an example of trivial parallelism, it has been chosen to see the potential of the out-of-order execution. The size of the matrices has been consciously chosen to be small to observe the effects of communication costs on our approach. To get meaningful results, the sequence of matrices that is processed has been chosen quite large to have runtimes in the range of several seconds. Unfortunately, the translation of synchronous guarded actions to DPNs inserted control dependencies between the iterations. Nevertheless, the speedup is in the range of 1.05 to 1.33.

The LU decomposition of a matrix requires several iterations to calculate a result. Compared to a matrix multiplication, it provides less parallelism in each iteration. In addition, the LU decomposition has dependencies between iterations and limits the parallelism across iterations. Nevertheless, the out-of-order execution still remains faster than the in-order execution, especially with extremely light-weight tasks.

The DFT benchmark applies a DFT-IDFT transformation to an audio file with several million samples. Although the DFT/IDFT provides parallelism within each iteration, the structure of this benchmark allows no calculation across the iteration. Despite this limitation, the speed-up of the out-of-order execution stays close to 1. Obviously, the overhead of out-of-order scheduling can be neglected.

The landscape generator represents an image processing application. In particular, this benchmark renders a sequence of images. The parameter H in parenthesis is the height of the generated pictures and basically scales the computational effort of the created nodes. The dependency graph of the DPN of this benchmark has pipeline characteristics: the nodes build a chain and only a few splits allow some parallelism within an iteration. Whereas, the pipeline character allows parallelism across the iterations which is reflected in the results.

V. CONCLUSIONS

This paper presents an efficient scheduling strategy of synchronous data-flow process networks that does not need detailed knowledge of the target architecture. Moreover, it provides flexibility with respect to 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. Moreover, the benchmarks have shown that even for programs that do not allow parallelism across iterations, the scheduling effort is negligible and the runtime is near to that of the in-order execution.

Although the presented benchmarks were executed on a shared memory system, this approach is not limited to such an architecture and can extend other architecture, e.g. to clusters. Using task pool teams as described in [15, 16], the presented approach can be applied to create cluster applications. At the time of publication, the library of Hippold was however not available, and therefore, it could not be considered to our approach.

REFERENCES

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Speedup on i5-750</th>
<th></th>
<th>Speedup on 2xXeon</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1n1t⇒io io⇒oooo</td>
<td></td>
<td>1n1t⇒io io⇒oooo</td>
<td></td>
</tr>
<tr>
<td>MatrixMult (16x16)</td>
<td>14.40</td>
<td>1.33</td>
<td>23.22</td>
<td>1.06</td>
</tr>
<tr>
<td>MatrixMult (32x32)</td>
<td>11.48</td>
<td>1.12</td>
<td>28.95</td>
<td>1.05</td>
</tr>
<tr>
<td>MatrixMult (48x48)</td>
<td>10.85</td>
<td>1.10</td>
<td>25.83</td>
<td>1.05</td>
</tr>
<tr>
<td>LU Decomp. (4x4)</td>
<td>0.75</td>
<td>1.95</td>
<td>0.67</td>
<td>1.37</td>
</tr>
<tr>
<td>LU Decomp. (8x8)</td>
<td>3.97</td>
<td>1.23</td>
<td>2.54</td>
<td>1.05</td>
</tr>
<tr>
<td>LU Decomp. (16x16)</td>
<td>4.39</td>
<td>1.07</td>
<td>7.17</td>
<td>1.04</td>
</tr>
<tr>
<td>LU Decomp. (32x32)</td>
<td>5.26</td>
<td>1.04</td>
<td>5.46</td>
<td>1.05</td>
</tr>
<tr>
<td>DFT (4 tasks)</td>
<td>6.20</td>
<td>1.00</td>
<td>7.10</td>
<td>0.97</td>
</tr>
<tr>
<td>DFT (8 tasks)</td>
<td>6.64</td>
<td>0.99</td>
<td>9.22</td>
<td>1.01</td>
</tr>
<tr>
<td>DFT (16 tasks)</td>
<td>4.87</td>
<td>1.01</td>
<td>5.56</td>
<td>0.99</td>
</tr>
<tr>
<td>DFT (32 tasks)</td>
<td>4.80</td>
<td>1.02</td>
<td>5.57</td>
<td>0.99</td>
</tr>
<tr>
<td>Landscape Gen. (H=20)</td>
<td>1.84</td>
<td>1.11</td>
<td>2.56</td>
<td>1.12</td>
</tr>
<tr>
<td>Landscape Gen. (H=200)</td>
<td>2.09</td>
<td>1.34</td>
<td>6.75</td>
<td>1.14</td>
</tr>
<tr>
<td>Landscape Gen. (H=400)</td>
<td>3.20</td>
<td>1.43</td>
<td>11.15</td>
<td>1.12</td>
</tr>
<tr>
<td>Landscape Gen. (H=800)</td>
<td>4.84</td>
<td>1.80</td>
<td>17.28</td>
<td>1.28</td>
</tr>
</tbody>
</table>

Figure 4. Experimental results for several benchmarks. Column 1n1t⇒io and io⇒oooo, respectively, show the speed-up of the out-of-order execution compared to the 1-node-1-thread approach and the in-order execution, respectively.


