Distributed Checkpointing in Dataflow with Static Scheduling

Tiago A. O. Alves
State University of Rio de Janeiro

tiago@ime.uerj.br

***Abstract*—The Dataflow model, where instructions or tasks are fired as soon as their input data is ready, was proven to be a good fit for parallel/distributed computation. Previous works have presented DFER (Dataflow Error Recovery Model), that allows transient error and recovery in dataflow by adding special tasks and edges to the dataflow graph itself. However, permanent faults or faults that cause a processing element (PE) to become irresponsive are not addressed by DFER. For those cases it is necessary to adopt a checkpointing method. Since the whole purpose of Dataflow is to achieve high levels of parallelism and explore the potential asynchronicity between PEs, it is clear that the checkpointing method adopted must be uncoordinated and distributed. Current algorithms for distributed checkpointing rely solely on guaranteeing that causality between checkpoints can be trackable. In the context of Dataflow with static scheduling, i.e. when the dataflow graph is partitioned among the available PEs at compile-time, causality trackability is not sufficient as we will show. Since static scheduling of dataflow graphs is very important in various scenarios, it calls for a new algorithm for distributed checkpointing that can be adopted for the execution of statically scheduled dataflow graphs. In this paper we describe why the ability to track causality is not enough for statically scheduled dataflow and introduce a new algorithm for distributed checkpointing specifically tailored for such model of execution.**

Keywords*—*dataflow, distributed checkpointing, fault tolerance, parallel programming

I. Introduction

The ever-increasing number of processing cores in a computing system demands parallel programming models that can take advantage of the available processing power to its full potential. The dataflow model [1], [2] assumes that instructions (or tasks) are fired as soon as their input operands are available, thus removing the need for a program counter. Dataflow can exploit parallelism in a natural way, which has made the model very appealing to the high performance computing community. Recent works used dataflow in different flavors and different levels of abstractions to extract performance from applications and make parallel programming easier [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13].

On the other hand, as computing systems grow in size and complexity, so does the probability of faults to occur. Previous works [14], [15] have shown that transient error and recovery in dataflow can be implemented by adding special tasks and edges to the dataflow graph itself. This solution was called DFER (Dataflow Error Recovery Model). However, permanent faults or faults that cause a processing element (PE) to become irresponsive are not addressed by DFER. For those cases it is necessary to adopt a checkpointing method. Since the whole purpose of Dataflow is to achieve high levels of parallelism and explore the potential asynchronicity between PEs, it is clear that the checkpointing method adopted must be uncoordinated and distributed [16].

Current algorithms for distributed checkpointing rely solely on guaranteeing that causality between checkpoints can be trackable [17], [18]. In the context of Dataflow with static scheduling, i.e. when the dataflow graph is partitioned among the available PEs in compile-time, causality trackability is not sufficient as we will show. Static scheduling of dataflow graphs is very important in various scenarios [19]. Hence, it calls for a new algorithm for distributed checkpointing that can be adopted for the execution of statically scheduled dataflow graphs.

In this paper we describe why the ability to track causality is not enough for statically scheduled dataflow and introduce new methods for distributed checkpointing that are actually suitable for such model of parallel execution.

For starters, we must clarify the concepts adopted in this work, such as Dynamic Dataflow, Static Scheduling and Distributed/Uncoordinated checkpointing. In this paper a section is dedicated to each of these concepts in order to make the need for our new checkpointing methods better understood by the reader. After introducing such concepts we present our novel solution for distributed checkpointing under the constraints of dataflow with static scheduling.

The rest of this paper is organized as follows: (*i*) Section II explains the main concepts of Dynamic Dataflow; (*ii*) Section III discusses some related works; (*iii*) Section IV presents our approach; (*iv*) Section V presents some experiments and results; (*v*) Section VI provides a overall discussion on the work and points out to some future works.

II. Dynamic Dataflow

The Dataflow model [1], [2] was conceived as an alternative to the Von Neumann model. Dataflow machines (or runtime environments) do not require a program counter nor register files. Instead, it assumes that instructions (or tasks) are triggered as soon as their input operands are available, naturally exposing parallelism. Dataflow has been gaining traction in the high performance computing community and recent work show that it is possible to use dataflow concepts, in different levels of abstractions, to ease parallel programming [3], [7], [8], [9], [10], [11], [12]. Also, as shown in [3], [13], dataflow execution can present performance comparable to traditional tools as OpenMP [20], Pthreads [21] and TBB [22].

979-8-3503-8160-3/23/$31.00 ©2023 IEEE
DOI 10.1109/SBAC-PADW60351.2023.00021

In order to understand our checkpointing solution, in this section we provide information on the dataflow architecture used to validate our approach and explain the most important structures that need to be saved upon checkpointing. Moreover, we cover the basics of dataflow, including explanations on how it deals with control instructions to implement loops and conditional execution and the implications this brings to the architecture.

TALM (*TALM is an Architecture and Language for Multi-threading*) [3], [4] is a dataflow execution model conceived to enable dataflow style execution on top of traditional Von Neumann machines. TALM toolchain was used to validate the solutions proposed in this work. It provides an instruction set that allows the description of complete dataflow applications, including conditional execution and loops. Since no dataflow hardware is used, *TALM* adopts a runtime environment (or virtual machine), called *Trebuchet*. To avoid interpretation overheads *TALM* supports the creation of customized super-instructions, blocks of code written in regular imperative language.

The *TALM* Reference Architecture, shown in Figure 1, assumes a set of identical processing elements (PEs) interconnected in a network, each one being responsible for interpretation and execution of a group of instructions, according to the dataflow principles. Each PE has the following components:

- A communication buffer to receive operands produced by instructions executed in other PEs.
- An instruction list that stores information related to each instruction mapped to a PE. Mapping can be done statically (in compile time) or dynamically (at execution time).
- A list of operands related to each instance of an instruction. Each instance is related to a loop iteration (or function call, as discussed in [3]). This structure is extremely important, since it will be used to check if all operands are available for a certain instruction.
- A ready queue that contains a list of instructions ready to be executed and that are only waiting for the processor to be available.
- The execution logic itself.

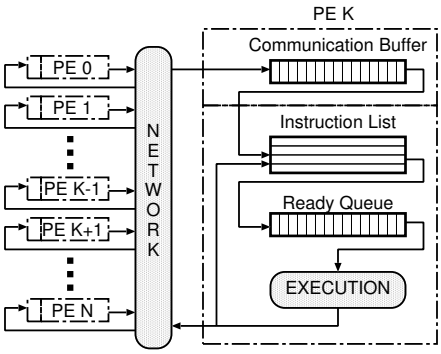
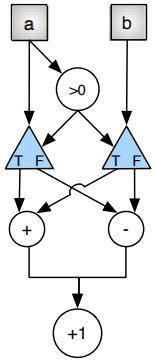


Fig. 1. TALM architecture

One of the main issues with dataflow is that, since there is no program counter, control instructions need to be able to change the flow of data in a dependency graph that represents the application, the so called dataflow graph. This is traditionally done by `Steer` instructions, that receive an operand O and a boolean selector S and will send O to one of two possibles paths, according to the value of S, as shown in Figure 2. Figure 2a shows the C-Code fragment of an if-then-else statement and Figure 2b shows the corresponding dataflow graph, where `Steer` instructions are represented by triangles. Notice, since there is no global state (no register files), that there has to be one `Steer` instruction for each input operand.

```
if (a>0)
   x = a+b;
else
   x = a-b;
x+=1;
```

(a) C-Code



(b) Dataflow graph

Fig. 2. Conditional execution in dataflow: Pane (a) shows a fragment of high-level C-Code containing an if-then-else statement; pane (b) shows the corresponding dataflow graph, where `Steer` instructions are represented as triangles.

Loops in dataflow also require a special attention. In Von Neumann machines, since all data are already stored in the register file or memory, all that is needed is a branching instruction to point to next iteration. In dataflow, steer instructions will have to be used, as in the example of the if-then-else statement. However, portions of the loop body may run faster than others, producing multiple operands that will wait to be consumed by the slower portion of the graph. Instructions that need more than one input operand may get triggered by operands from different iterations and produce erroneous results. A simple solution for that is to only allow a loop iteration to start when the last one is completely finished, the so called Static Dataflow.

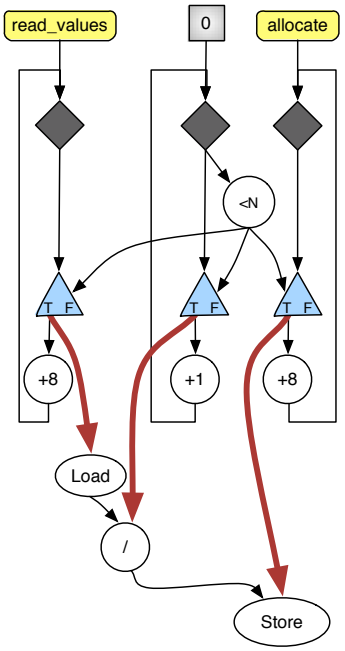
Loops are a major source of parallelism and are usually responsible for most of the work performed in a piece of code. If our main goal is to improve performance, Static Dataflow is not the solution. The alternative is called Dynamic Dataflow, where there are no barriers between iterations. In this approach, each operand will carry a tag that identifies the number of the iteration it belongs to. In the beginning of each iteration `Inctag` instructions will increment the tags and instruction execution will only be triggered when all input operands with the same tag are available.

Dynamic Dataflow also needs to be carefully explored. In loops where there are no dependencies between iterations and where the loop body is much slower that the control instructions, several iterations may be dynamically unrolled, causing a **parallelism explosion**, which may end up consuming all storage in the communication buffers of the processing elements. Those operands may take a long time to get consumed and jeopardize performance. Figure 3a shows a fragment of

high-level C-Code containing a loop statement that performs a double precision floating point division in all elements of a vector. Figure 3b shows the corresponding graph, where `Inctag` instructions are represented as diamonds. The `Load`, `/` and `Store` may take more time than the computation i and the addresses of v and x. This may cause a parallelism explosion related to the edges that are marked thicker in the graph.

```
double * x = allocate();
double * v = read_values();
for (i=0; i<N; i++)
    x[i]=v[i]/i;
```

(a) C-Code



(b) Dataflow graph

Fig. 3. Loops in Dynamic Dataflow: pane (a) shows a fragment of high-level C-Code containing a loop; pane (b) shows the corresponding dataflow graph, where `Inctag` instructions are represented as diamonds and thicker edges are the ones that could suffer from parallelism explosion.

TALM toolchain also has the *Couillard* Dataflow Compiler [4] that translates code written in THLL (*TALM* High-Level Language), an extension of C that allows definition of super-instructions and their dependencies, into *TALM* assembly language. It also extracts code form super-instructions to function to a separate C file. The super-instruction code is compiled with `gcc` and run natively by the host machine, while the dataflow graph that calls the super-instructions and contains all control instructions is interpreted by *Trebuchet*.

III. Related Work

This sections describes previous works on distributed uncoordinated checkpointing and show why such methods do not apply to statically scheduled dataflow. In [14], [15] we addressed the issues of transient errors in dataflow execution with DFER (Dataflow Error Recovery). DFER relies basically on the addition of special instructions and edges to the original dataflow graph of the program. DFER, however, by itself is not capable of handling permanent faults that cause the processing element (PE) to become completely irresponsive (or when all PEs halt due to power-outage, for instance). This lead us to the current research introduced in this paper.

Checkpointing algorithms (or protocols, depending on the context) periodically save the state of the application with added overhead for the extra-communication and the I/O to save the states. When dealing with parallelized applications, the checkpointing algorithm must make sure that the set of states used to rollback the application after a fault is consistent (i.e. the application may resume and still provide deterministic results as if the fault never occurred).

Checkpoint algorithms for distributed (or multithreaded) algorithms usually fall under three categories [16]:

- Uncoordinated: these algorithms/protocols make no assumption about the global state of the application and, therefore, may cause domino effect.
- Coordinated: coordinated algorithms/protocols trigger a global synchronization among the processing elements in order to assure that a consistent global state is captured.
- Communication-induced: In communication-induced checkpointing, from time to time a special message is sent to the PEs forcing them to save their current state.

Since we aim at high performance by exploiting dataflow's natural parallelism, it makes sense that we choose an uncoordinated algorithm for checkpointing, since it would be counterproductive to add another level of synchronization between PEs, i.e. messages that do not necessarily carry data pertaining to the computation. In our proposed approach, the PEs must be able to decide independently when to save a checkpoint, for instance, it might be a good heuristic to save a checkpoint when the PE finds itself idle, awaiting data from other PEs to trigger the execution of instructions that were statically mapped to such PE.

Furthermore, coordinated checkpointing algorithms, such as Chamdy and Lamport's [23] require communication between **all** processing elements, even those that do not exchange data between each other during the computation. That can be a serious hazard to the scalability of the application.

It is important to notice that the concept of *consistent global snapshots* [24] adopted by early checkpointing algorithms does not fit our needs in the scenario of statically scheduled dataflow, since such concept only aims at avoiding causality between checkpoints in the set that comprises the global state (or global snapshots, the term adopted in these works). Take, for instance, the pattern of communication between PEs of Figure 4, where $C_{i,x}$ representes the x-th local checkpoint of the i-th processing element. According to the concept of consistent snapshot adopted by Netzer and Xu [24], $C_{0,2}$ and $C_{1,1}$ could belong to the same consistent snapshot, since there is no direct causality or *zigzag path* (a concept introduced by Netzer and Xu in this same work) between them. However, in our case of dataflow applications statically scheduled such snapshot would not work. Consider that PE 0 rollbacks to $C_{0,2}$ and PE 1 rollbacks to $C_{1,1}$. In this case, the

data originally sent by PE 0 to PE 1 in the message $m1$ would not be sent again during rollback, and PE 1 would be stuck waiting for such data. An obvious solution for such problem would be to also save the communication channels' states to preserve in-flight messages, but this is an extra burden we do not want to inflict on the dataflow execution. Instead, in our proposed approach (presented in the next section) we will choose to rollback PE 0 to $C_{0,1}$, since the work done by PE 0 between $C_{0,1}$ and $C_{0,2}$ will only have to be re-executed in case of a fault (which we assume to be of low probability).

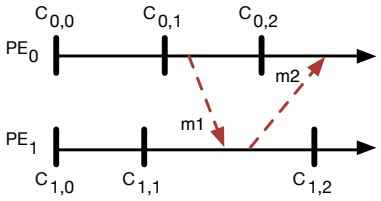


Fig. 4. Example of checkpoints that are considered *consistent* with each other but do not meet our criteria for dataflow.

IV. Proposed Approach

To address the conflict between the widely adopted concept of consistent snapshot and our dataflow model with static scheduling, we shift the focus from causality between checkpoints to the amount of data sent and received between PEs. Instead of verifying if one checkpoint *happens-before* another, we verify if the amount of data sent by PE i to PE j matches the information stored in $C_{i,x}$ and $C_{j,y}$. If so we consider $C_{i,x}$ and $C_{j,y}$ to be eligible to comprise a global snapshot. These amounts of data are represented by $sent(i)$ and $recvd(j)$ respectively and their latest value is stored in the checkpoints.

Without loss of generality, suppose that every message exchanged between PEs carries a unit of data. Therefore, in Figure 4 we have that, in $C_{0,2}$, $sent(1) = 1$, while, in $C_{1,1}, recvd(0) = 0$, rendering both checkpoints incompatible with each other. Ultimately, the only solution in this figure would be to have PE 0 rollback to $C_{0,1}$ and PE 1 to $C_{1,1}$.

Now consider the example of Figure 5, which can be representative of the message exchange of the partitioned dataflow graph of Figure 6a. Here, we have a constant back and forth exchange of data between PE 0 and PE 1. This situation does not favor our checkpointing model, since ultimately both PE's will end up having to rollback to their first checkpoint (the beginning of the program) due to the incompatibility of their checkpoints according to our criteria. This is called a *domino effect* and was already described by Alvisi in [25].

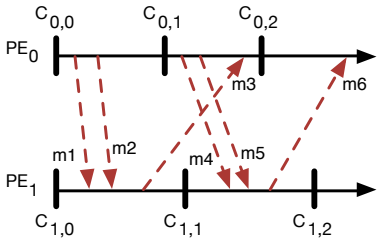
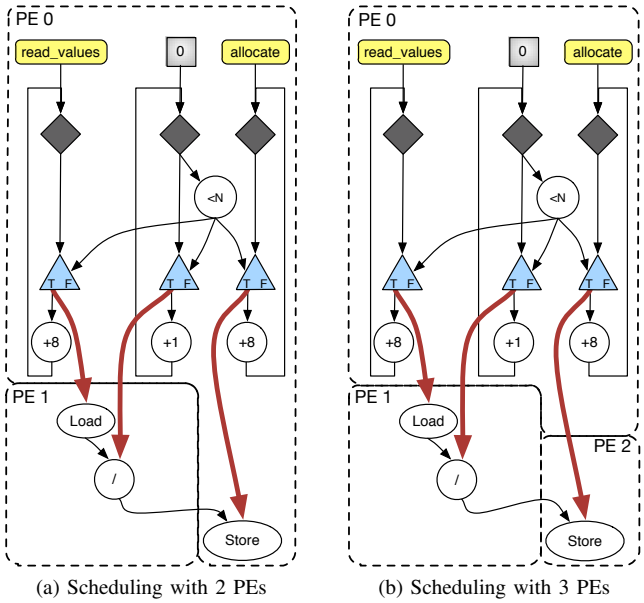


Fig. 5. Example of checkpoints that cause a domino effect.

The domino effect in the previous example is due to the fact that the communication pattern between PE 0 and PE 1 form a cycle. Such cycle could easily be broken by adding a third PE to the system and allocating the `Store` instruction to it, as is shown in Figure 6b. Of course, the actual addition of a third PE just for one instruction would probably be too costly, although it helps us picture the solution adopted. Instead of adding new PEs to break communication cycles we created the concept of *virtual* PEs. Basically, virtual PEs share the same hardware but have different contexts and, therefore, different checkpoints, just as what is done with entire virtual machines.



(a) Scheduling with 2 PEs (b) Scheduling with 3 PEs

Fig. 6. Scheduling examples.

In Figure 6a we have the same dataflow graph as in Figure 6b, but with the addition of the virtual PE 2, to which the `Store` instruction is statically scheduled. A potential pattern of communication in this case can be seen in Figure 7. Here, a possible global snapshot would be $\{C_{0,2}, C_{1,2}, C_{2,2}\}$.

Since we shifted our focus from causality between checkpoints to the amount of data exchanged between them, we can actually accept two checkpoints with a happen-before relation between them in the same global snapshot. For this purpose, we added a structure similar to a reorder buffer (ROB) to the original instruction list in the TALM base architecture. In this ROB-like structure, the PE stores a sorted list of the iteration tags of the input data received for each instruction. This sorted list only grows if input data arrives out of order (iteration order), which can possibly happen if different iterations of a loop execute different paths in the graph. When there are no gaps in the ROB, i.e. all the tags in the ROB are equal to the tag in the previous position plus one, the ROB can be flushed, remaining only the last iteration tag received.

Considering this addition to the TALM dataflow architecture, in Figure 7, $\{C_{0,1}, C_{1,2}, C_{2,2}\}$ could also comprise a global snapshot, since PE 1 would be able to know that the data received in $m4$ and $m5$ have already been received and just ignore them.

Memory operations pose a special challenge for check-

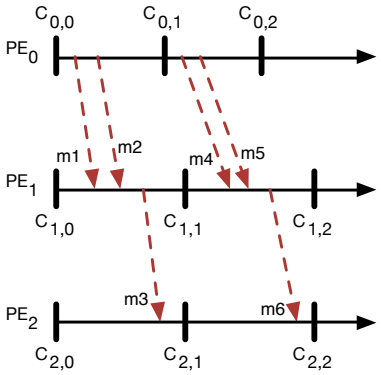
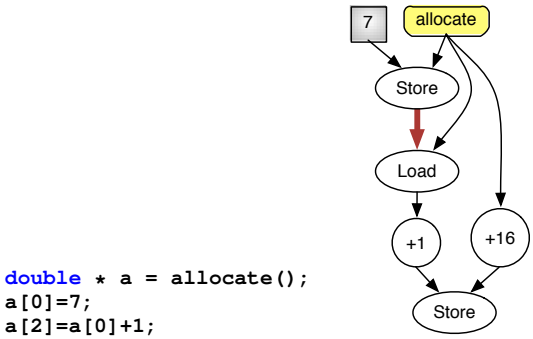


Fig. 7. Example of checkpoints where a global snapshot can be composed.



(a) C-Code (b) Dataflow graph

Fig. 8. The problem with memory operations: pane (a) shows a fragment of high-level C-Code where element $a[2]$ depends on $a[0]$; pane (b) shows the corresponding dataflow graph compiled without optimizations, just to illustrate that an edge has to be inserted between the `Store` and `Load` of $[a]$. The extra ordering edge is red and ticker than the others.

pointing. For starters, we must save all the data written by `Store` instructions since the last checkpoint was taken. We call this data the *Store Log*, which is part of the local checkpoint.

The solution for data dependencies between `Load` and `Store` instructions derives directly from the ordering of the sequential version of the application. Although the `Store` instruction does not produce any output data, since it writes the input value received directly in the memory, the dataflow compiler may add an outgoing edge from a `Store` to a `Load` instruction in case it is able to detect such ordering in the original sequential code. This edge only creates a dependency forcing the `Load` instruction to execute after the corresponding `Store`. As the `Store` instruction does not produce any data, the message sent through that additional edge is just an ordering token. Once such dependency is incorporated into the dataflow graph, the parallel execution will follow program order in respect to these memory accesses and our algorithm will also be able to coordinate checkpoints to form global snapshots, since the ordering token will be treated as any other message.

The code and respective dataflow graph in Figure 8 represent an example of such approach to memory dependencies. The highlighted edge from the first `Store` instruction to the `Load` instruction is introduced to guarantee the ordering of memory operations according to the program semantics.

V. Experiments

To evaluate the overheads imposed by our checkpointing algorithm we executed in the Trebuchet runtime [3] a simple dataflow graph containing a loop similar to the one in Figure 6. First we measured total execution time in the original version of Trebuchet and then compared to the total execution time in a version of Trebuchet modified to include our checkpointing algorithm. The checkpointing criteria adopted at each PE was to save a checkpoint after every 1K instructions executed or whenever the PE found itself idle (waiting for input data to trigger the execution of new instructions). The overhead observed when comparing the two versions, for this micro-benchmark, was less than 2%.

As an experiment to verify the algorithm's correctness we also killed the entire application in random moments and them restarted it with the set of checkpoints that met our criteria and would cause the least amount of re-executions. We did not, however, measure the average time taken by re-executions, such experiment shall be an object of future work.

VI. Discussion

In this paper we introduced an uncoordinated algorithm for distributed checkpointing of statically scheduled dataflow graphs. To the best of our knowledge, this problem, with the given constraints, was not treated in any previous work. The preliminary experimental results are promising, showing low levels of overhead added by the checkpointing algorithm. Although this work was based on the TALM architecture, the algorithm introduced can be applied to any system with statically scheduled dynamic Dataflow, which, as discussed previously, is a growing trend for achieving performance at scale. Two key aspects of the algorithm which pertain specifically for Dataflow are the ROB to treat out-of-order messages (i.e. messages pertaining to loop iterations that arrive out-of-order) and the total lack of synchronization or any kind of coordination between PEs, meaning PEs can checkpoint their state at convenient moments and checkpoints of two PEs that did not exchange data do not to be checked together for consistency.

Nonetheless many more experiments still must be done as well as a formal proof of correctness of the algorithm. Both will be addressed in future work.

References

[1] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *SIGARCH Comput. Archit. News*, vol. 3, no. 4, pp. 126–132, 1974.

[2] Arvind and R. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *Computers, IEEE Transactions on*, vol. 39, no. 3, pp. 300–318, March 1990.

[3] T. A. Alves, L. A. Marzulo, F. M. Franca, and V. S. Costa, "Trebuchet: exploring TLP with dataflow virtualisation," *International Journal of High Performance Systems Architecture*, vol. 3, no. 2/3, p. 137, 2011.

[4] L. A. Marzulo, T. A. Alves, F. M. França, and V. S. Costa, "Couillard: Parallel programming via coarse-grained data-flow compilation," *Parallel Computing*, vol. 40, no. 10, pp. 661 – 680, 2014.

[5] T. A. O. Alves, B. F. Goldstein, F. M. G. França, and L. A. J. Marzulo, "A minimalistic dataflow programming library for python," in *Proceedings of the 2014 International Symposium on Computer Architecture and High Performance Computing Workshop*, ser. SBAC-PADW '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 96–101.

[6] R. J. Silva, B. Goldstein, L. Santiago, A. C. Sena, L. A. Marzulo, T. A. Alves, and F. M. Frana, "Task scheduling in sucuri dataflow library," in *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, 2016, pp. 37–42.

[7] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, pp. 173–193, 2011-03-01 2011.

[8] S. Balakrishnan and G. Sohi, "Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs," in *33rd International Symposium on Computer Architecture (ISCA'06)*. Washington, DC, USA: IEEE, 2006, pp. 302–313.

[9] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 59–70.

[10] "Tbb flowgraph," http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm, accessed on August 8, 2014.

[11] G. Bosilca, A. Bouteiller, A. Danalis, T. Hrault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing." *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[12] R. Giorgi, R. M., F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R., A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. Minh, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis, S. Zuckerman, and M. Valero, "TERAFLUX: Harnessing dataflow in next generation teradevices," *Microprocessors and Microsystems*, pp. –, 2014, available online 18 April 2014.

[13] T. A. Alves, L. A. J. Marzulo, and F. M. G. Franca, "Unleashing parallelism in longest common subsequence using dataflow," in *4th Workshop on Applications for Multi-Core Architectures*, 2013.

[14] T. A. O. Alves, S. Kundu, L. A. J. Marzulo, and F. M. G. França, "Online error detection and recovery in dataflow execution," in *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, July 2014, pp. 9–104.

[15] T. A. O. Alves, L. A. J. Marzulo, S. Kundu, and F. M. G. França, "Domino effect protection on dataflow error detection and recovery," in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFT 2014, Amsterdam, The Netherlands, October 1-3, 2014*. IEEE Computer Society, 2014, pp. 147–152.

[16] X. Besseron and T. Gautier, "Optimised recovery with a coordinated checkpoint/rollback protocol for domain decomposition applications," *Modelling, Computation and Optimization in Information Systems and Management Sciences (MCO '08)*, pp. 497–506, 2008.

[17] R. Baldoni, J.-M. Helary, a. Mostefaoui, and M. Raynal, "A communication-induced checkpointing protocol that ensures rollback-dependency trackability," *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, 1997.

[18] I. C. Garcia and L. E. Buzato, "An efficient checkpointing protocol for the minimal characterization of operational rollback-dependency trackability," *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pp. 126–135, 2004.

[19] T. A. Alves, "Dataflow execution for reliability and performance oncurrent hardware," Ph.D. dissertation, Federal University of Rio de Janeiro, 5 2014.

[20] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[21] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[22] J. Reinders, *Intel threading building blocks : outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[23] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[24] R. H. B. Netzer and J. Xu, "Necessary and sufficient conditions for consistent global snapshots," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 2, pp. 165–169, Feb. 1995.

[25] L. Alvisi and K. Marzullo, "Message logging: pessimistic, optimistic, and causal," in *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*, May 1995, pp. 229–236.