This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

Concurrency Analysis in Dynamic Dataflow Graphs

Name: Tiago A. O. Alves *Member, IEEE,* Leandro A. J. Marzulo *Member, IEEE,* Sandip Kundu *Fellow, IEEE,* and Felipe M. G. França *Senior Member, IEEE,*

***Abstract*—Dynamic dataflow scheduling enables effective exploitation of concurrency while making parallel programming easier. To this end, analyzing the inherent degree of concurrency available in dataflow graphs is an important task, since it may aid compilers or programmers to assess the potential performance a program can achieve via parallel execution. However, traditional concurrency analysis techniques only work for DAGs (directed acyclic graphs), hence the need for new techniques that contemplate graphs with cycles. In this paper we present techniques to perform concurrency analysis on generic dynamic dataflow graphs, even in the presence of cycles. In a dataflow graph, nodes represent instructions and edges describe dependencies. The novelty of our approach is that we allow concurrency between different iterations of the loops. Consequently, a set of concurrent nodes may contain instructions from different loops that can be proven independent. In this work, we provide a set of theoretical tools for obtaining bounds and illustrate implementation of parallel dataflow runtime on a set of representative graphs for important classes of benchmarks to compare measured performance against derived bounds.**

I. Introduction

In this work we will provide a set of theoretical tools adequate to obtaining bounds on the performance of dataflow programs. Although previous work exists for obtaining such bounds on the execution of programs described as DAGs [1], [2], [3], this kind of study is yet to be done for dynamic dataflow execution. Since DAGs are not entirely suitable to represent dataflow programs we first need to introduce a generic model to represent dynamic dataflow programs.

The obvious difference between DAGs and dynamic dataflow graphs is the presence of cycles, which represent dependencies between different iterations of loops. Simply applying the techniques employed to estimate performance of DAGs to DDGs (dynamic dataflow graphs) would not succeed, since problems like finding the longest path or the maximal independent sets can not be solved in DDGs using traditional algorithms. As follows, it is necessary to develop new theoretical ground that allows us to analyze the concurrency present in a DDG.

When analyzing the potential performance acceleration in a DDG we are interested in two concurrency metrics: the *average* and the *maximum* degrees of concurrency. The maximum degree of concurrency is the size of the maximum set of instructions which may, at some point, execute in parallel, while the average degree of concurrency is the sum of all work divided by the critical path. In a DAG, the maximum degree of concurrency is simply the size of the maximum set of instructions such that there are no paths in the DAG between any two instructions in the set. The average degree of concurrency of a DAG can be obtained by dividing it in maximum partitions that follow that same rule and averaging their sizes. In DDGs, however, it is a little more complicated to define and obtain such metrics.

The motivation behind these metrics is straightforward. The maximum degree of concurrency describes the maximum number of processing elements that are useful to execute the graph. Allocating additional processing elements will not lead to further improvement in performance. The average degree of concurrency can be shown to be the maximum speed-up that can be obtained by executing the graph in an ideal system with infinite resources, optimal scheduling and zero overhead. We also present a *lower bound* for the speed-up. Arora et al. had introduced a lower bound for dataflow execution in DAGs [4]; our work extends this to DDGs.

While static scheduling of DAGs have been treated extensively by previous researchers [5], [6], [7], [8], dynamic dataflow has not received similar attention. In this work, we present a theoretical framework for dynamic dataflow analysis and present experimental results on scheduling dynamic dataflow graphs with bounded execution resources.

The rest of this work is organized as follows: *(i)* In Section II we explain the dynamic dataflow graph (DDG) model; *(ii)* in Section III we present an algorithm to obtain the maximum degree of concurrency in a DDG; *(iii)* in Section IV we define and show how to calculate the maximum speed-up of a DDG; *(iv)* in Section VI we model and compare two different pipelines as an example of how our theory can be applied; *(v)* in Section VII we present a few experiments done to validate the theory; and *(vi)* in Section IX we conclude and discuss our contributions.

II. Dynamic Dataflow Graphs

First of all, we shall present a definition for Dynamic Dataflow Graphs (DDGs) such that the theory presented for them can be applied not only to our work, but also to any model that employs dynamic dataflow execution. For the sake of simplicity of analysis, first we consider a constrained dynamic dataflow program as follows, then we generalize the analysis to more general DDGs.

1) The only kind of conditional execution in loop bodies are loop test.
2) There are no nested loops.
3) Dependencies between different iterations only exist between two consecutive iterations (from the i-th to the $(i+1)$-th iteration).

Although these may seem too restrictive at first, we will show throughout this paper that the study of DDGs allows us to analyze an important aspect that is unique to dynamic dataflow, the asynchronous execution of cycles that represent

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

loops. Also, as mentioned, it is not hard to eliminate these restrictions and apply theses methods to programs that can not be represented in a graph that strictly follows DDG convention.

A DDG is a tuple $D = (I, E, R, W)$ where:

- I is the set of instructions in the program.
- E is the set of edges, ordered pairs (a, b) such that $\{a, b\} \subseteq I$, the instruction b consumes data produced by a and a is not inside a loop or a and b are inside a loop and this dependency occurs in the same iteration, i.e. the data b consumes in iteration i is produced by a also in iteration i.
- R is the set of return edges, they are also ordered pairs, just like the edges in E, but they describe dependencies between different iterations, consequently they only happen when both instructions are inside a loop. A return edge $r = (a, b)$ describes that b, during iteration i of the loop, consumes data produced by a in the iteration $i-1$.
- W is the set of *weights* of the instructions, where the weight represents the granularity of that instruction, since granularities may vary.

III. Maximum Degree of Concurrency in a DDG

We call the maximum degree of concurrency in a DDG the largest set of instructions that can, at some point, be executing in parallel. In DAGs, it is direct to verify that these *independent sets* must be such that there is no path in the DAG between any pair of instructions in the set. Since we are dealing with DDGs it is a bit more complicated to obtain this measure, as there may be dependencies (or the lack of them) between different iterations.

In order to delve deeper into the problem we should first introduce additional notation. If v is an instruction inside the body of a loop, we call $v(i)$ the instance of v executed in the $i\text{-}th$ iteration of the loop. Let us also indicate by $u(i) \rightarrow v(j)$ that $v(j)$ depends on $u(i)$ and by $u(i) \not\rightarrow v(j)$ that $v(j)$ does **not** depend on $u(i)$.

In the DDG of Figure 1a, $b(i) \rightarrow c(i)$ and $c(i-1) \rightarrow b(i)$ (observe the return edge, represented as a dashed arc), for $i > 1$, so, transitively $b(i-1) \rightarrow b(i)$. In this case, at any point it is only possible to have exactly one instruction running at a time. We say the maximum degree of concurrency for this DDG is 1. Now consider the DDG of Figure 1b. This time, $b(i) \rightarrow c(i)$, $b(i-1) \rightarrow b(i)$ and $c(i-1) \rightarrow c(i)$, but $c(i-1) \not\rightarrow b(i)$. So, potentially, $b(i)$ and $c(i-1)$ can execute in parallel, yielding a maximum degree of concurrency of 2 for this DDG.

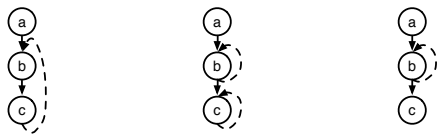


Fig. 1. Example of DDGs whose maximum degree of concurrency are 1, 2 and **unbounded**, respectively. The dashed arcs represent return edges.

In the DDG of Figure 1c $c(i)$ depends on $b(i)$ and $b(i)$ depends on $b(i-1)$ (observe the return edge, represented as a dashed arc), but $c(i)$ does not depend on $c(i-1)$. For instance, it would be possible for $c(i)$ and $c(i+1)$ to execute in parallel. As a matter of fact, if b runs fast enough it would be possible to have potentially **infinite** instances of c executing in parallel. We say that DDGs with this characteristic have an **unbounded** maximum degree of concurrency, since, given the right conditions, it is possible to have an unbounded number of instructions executing in parallel. The importance of this metric lies in the decision of resource allocation for the execution of the dynamic dataflow graph. If the maximum degree of concurrency C_{max} of a DDG is bounded, at any time during the execution of the DDG no more than C_{max} processing elements will be busy. Therefore, allocating more than C_{max} PEs to execute that DDG would be potentially wasteful.

Now that we have introduced the concept of maximum degree of concurrency in DDGs we shall present an algorithm to calculate it. Basically, the technique employed by the algorithm presented in this chapter consists of applying a transformation similar to *loop unrolling* [9] to the DDG.

The way we unroll a DDG is simply by removing return edges, replicating the portions of the DDG that are inside loops k times (where k is the number of iterations being unrolled) and adding edges between the subgraphs relative to each iteration in accordance with the set of return edges. The example of Figure 2 clarifies how it is done and in the algorithm we define the process of unrolling formally. In this example b, c, d and e are instructions inside a loop and a does some initialization and never gets executed again. In order to unroll the loop twice, we remove the return edge, replicate the subgraph that contains b, c, d, e twice, label the instructions accordingly to the iteration number, and add the edges $(e(0), b(1))$ and $(e(1), b(2))$ to represent the dependency correspondent to the return edge. Notice that the graph obtained through this process is not only a DDG, but it is also a DAG, since it has no cycles (because the return edge was removed) and the set of return edges is empty. We shall refer to the DAG obtained by unrolling D as D_k.

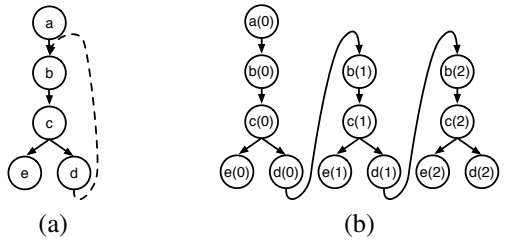


Fig. 2. Example of unrolling a DDG. Instructions b , c, d and e are inside a loop (observe the return edge) and a is just an initialization for the loop. The loop is unrolled twice, and the edges $(d(0), b(1))$ and $(d(1), b(2))$ are added to represent the dependency caused by the return edge.

After unrolling the DDG k times, the algorithm proceeds to obtain the maximum degree of concurrency present in the $k+1$ iterations. This is accomplished by creating a complement-path graph of the DAG D_k obtained by unrolling the DDG. A complement-path graph of a DAG $D_k = (I, E)$ is defined

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

as an undirected graph $C = (I, E')$ in which an edge (a, b) exists in E' if and only if a and b are in I and there is no path connecting them in D_k. An edge (a, b) in the complement-path graph indicates that a and b are independent, i.e., $a \not\rightarrow b$ and $b \not\rightarrow a$. Note that as the complement-path graph is constructed as a transformation of D_k, this relation of independence might refer to instances of instructions in different iterations. In Figure 3 we present as an example the complement-path graph obtained from a DAG. The only thing left for obtaining the maximum degree of concurrency in the $k + 1$ iterations is to find the largest set of independent instructions. Recalling that an independent set of instructions is defined as a set S such that, for all pair of instructions $a, b \in S$, $a \not\rightarrow b$ and $b \not\rightarrow a$. It follows from this definition that the largest set of independent instructions in D_k is simply the maximum clique in C. Consequently, the maximum degree of concurrency is $\omega(C)$, the size of the maximum clique in C.

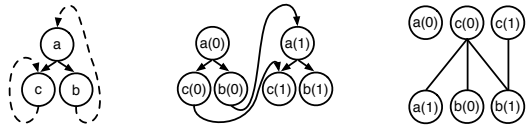


Fig. 3. Example of execution of the second iteration of the algorithm ($k = 1$). In (a) we have the original DDG, in (b) the DAG obtained by unrolling it once and in (c) we have the complement-path graph. The maximum degree of concurrency is 2, since it is the size of the maximum cliques.

The algorithm then proceeds to repeat the procedure described above, replacing D_k with D_{k+1}. The algorithm stops if: *(i)* the maximum degree of concurrency remains the same as the last step (i.e. the largest independent set in D_k and D_{k+1} have the same size) or *(ii)* the maximum degree of concurrency is greater then the number of instructions in D, in this case the maximum degree of concurrency is **unbounded**, i.e. it will always grow with the number of iterations.

Input: A DDG $D = (I, E, R, W)$

1) Initialize $k := 0$.
2) Define L as the subset of I that contains only instructions that are inside loops.
3) Unroll D k times by replicating L k times (resulting in $I_k = I \cup \{l_0(1), .., l_0(k - 1), l_1(0), l_1(1), .., l_1(k), .., l_n(k)\}$).
4) Create $E_k = \{(a(i), b(i)) : (a, b) \in E, (a, b), 0 < i < k\} \cup \{(a(i), b(i + 1)) : (a, b) \in R, 0 < i < k - 1\}$.
5) Define $D_k = (I_k, E_k, W)$.
6) Create a complement-path graph C_k of D_k defined as $C_k = (I_k, \bar{E_k}, W)$ where $\bar{E_k} = \{(a, b) : a, b \in I_k$, there is no path in D_k between a and b and vice-versa $\}$.
7) Find a maximum clique in C_k, and define the new maximum degree of concurrency as $\omega(C_k)$, which is the size of a maximum clique in C_k.
8) If $\omega(C_k)$ increased the maximum degree of concurrency, and if $\omega(C_k) < |I| + 1$, make $k := k + 1$ and go back to (2).
9) If $\omega(C_k) = |I|+1$, the maximum degree of concurrency is infinite (unbounded), otherwise it is $\omega(C_k)$.

Algorithm that returns the maximum degree of concurrency C_{max} of a graph or decides that the degree of concurrency is unbounded.

Lemma 1: If in the i-th, $i > 0$ iteration of Algorithm III there is an instruction such that $a(i-1) \not\rightarrow a(i)$, the maximum degree of concurrency in the DDG is unbounded.

Proof: If there is an instance $a(i), i > 0$, it means a is inside a loop and was added as part of the unrolling process. If $a(i - 1) \not\rightarrow a(i)$, an instance of a does not depend on its execution from the previous iteration, so given the right conditions there can be potentially an unboundend number of instances of a executing in parallel. ■

Lemma 2: If Algorithm III reaches an iteration k such that $\omega(C_k) = \omega(C_{k-1})$, it will not find a clique with size greater than $\omega(C_k)$ and therefore can stop.

Proof: Suppose, without loss of generality, that a maximum clique K in C_{k-1} does not have an instance of an instruction b. If $b(k)$ can not be added to K to create a greater clique, we have that there is an instance $a(i), i < k$, in K such that $a(i) \rightarrow b(k)$, since $b(k) \not\rightarrow a(i)$, because dependencies toward previous iterations are not possible. Consequently, another clique could be constructed such that, in iteration i of the algorithm, $b(i)$ was inserted instead of $a(i)$ and in iteration k there will be a clique where all $u(j), j \geq i$, were replaced by $u(j + 1)$ and this clique is greater than K. But, according to the hypothesis, the $\omega(C_k) = \omega(C_{k-1})$, so we must have that $b(i) \rightarrow a(j), i < j$ as well. In that case, it is not possible to obtain a clique with more instructions than K.

■

Theorem 1: Algorithm III finds the maximum degree of concurrency of a DDG, if it is bounded, or decides that it is unbounded if it in fact is.

Proof: Algorithm III stops at iteration k if either $\omega(C_k) > |I|$ or if $\omega(C_k) = \omega(C_{k-1})$. In the former case, it decides that the maximum degree of concurrency of the DDG is unbounded and in the latter it returns that $\omega(C_k)$ is the maximum degree of concurrency. First, suppose that for a DDG $D = (I, E, R, W)$ the algorithm obtained $\omega(C_k) > |I|$. In this case, there is at least one instruction a in I such that $a(i) \not\rightarrow a(i + 1)$, and thus, according to Lemma 1 the DDG is indeed unbounded. Now consider the other stop criteria, where $\omega(C_k) = \omega(C_{k-1})$. According to Lemma 2, it is not possible to have a clique with a greater number of instructions than $\omega(C_k)$, thus there is no point in continuing to run the algorithm and $\omega(C_k)$ really is the number of instructions in a maximum clique possible for that DDG. ■

IV. Average Degree of Concurrency (Maximum Speed-up)

The other important metric we adopt in our analysis is the average degree of concurrency, which is also the maximum possible speed-up. The average degree of concurrency of a program is the average amount of work that can be done in parallel along its critical path. Following the notation adopted in [10] and [4], we say that T_n is the minimum number of computation steps necessary to execute a DDG with n PEs,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

where a computation step is whatever unit is used for the instructions' weights. T_1 is thus the sum of all instruction weights, since no instructions execute in parallel when there is only one PE. T_∞ is the minimum number of computation steps it takes to execute a DDG with an infinite number of PEs. T_∞ is also the *critical path* of that DDG, since in a system with infinite PEs the instructions in the critical path still have to execute sequentially. We can thereby say that the maximum speed-up S_{max} of a DDG is T_1/T_∞.

Figure 4a show an example of peseudocode, while its corresponding DDG is represented in Figure 4a. Since we know that the number of iterations executed is n, we can represent the entire execution as a DAG by unrolling the loop n times the same way we did in Algorithm III. This process gives us the DAG of Figure 4b. Let us represent with w_{init}, w_a and w_b the *weights* of the instructions that execute procedures $init$, $procA$ and $procB$, respectively. The critical path of the execution is the longest path in the DAG (considering weights), which is $w_{ini} + w_a.n + w_b$ and, consequently, the maximum speed-up is:

$$\frac{w_{init} + (w_a + w_b) \cdot n}{w_{init} + w_a \cdot n + w_b}$$

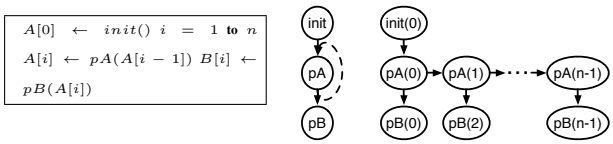


Fig. 4. Example corresponding to a loop that executes for n iterations. In (a) we present the pseudo-code, in (b) the corresponding DDG and in (c) the DAG that represents the execution.

Now suppose we do not know the number of iterations that are going to be executed a priori. In this case an exact value for n to calculate maximum speedup is unavailable. On the other hand, if we define a function $S(n)$ equal to the above equation for the maximum speedup, we find that the speed-up of this DDG corresponds to a function with asymptotic behaviour, since:

$$\lim_{n\to\infty} S(n) = \frac{w_a + w_b}{w_a}$$

This means that if the maximum speed-up is bounded and converges to a certain value as the number of iterations grows. As will be shown in Theorem 2, this property is true for any DDG, even if the maximum degree of concurrency of the DDG is unbounded. And it is a very important property since it is often the case in which loops will run for a very large number of iterations and this property allows us to estimate the maximum speed-up for them. We shall thus enunciate this property in a theorem and then prove it.

Theorem 2: Given a dynamic dataflow graph $D = (I, E, R, W)$, let D_k be the k-th graph generated by unrolling D as described in Algorithm III, W' the weights of instructions that are inside loops and let L_k be the longest path in D_k, we have

$$S_{max} = \lim_{k\to\infty} \frac{(\sum_{w_i \in W'} w_i) \cdot k}{L_k}$$

and S_{max} converges even if C_{max} is unbounded.

Proof:

The equation comes directly from the definition of maximum speed-up. Since $L_k \geq min(W) \cdot k$, we get

$$\lim_{k\to\infty} \frac{(\sum_{w_i \in W'} w_i) \cdot k}{L_k} \leq \lim_{k\to\infty} \frac{(\sum_{w_i \in W'} w_i) \cdot k}{min(W) \cdot k}$$

and since the right-hand part of the inequality is bounded, S_{max} must be bounded. ■

The property enunciated in Theorem 2 is also important because it shows us that one can not expect to get better performance just by adding processing elements to the system indefinitely, without making any changes to the DDG, since at some point the speed-up will converge to an asymptote. Now that we have an upper bound for speed-up, it can also be useful to have a lower bound for it as well, considering that it would provide us the tools to project the expected performance of a DDG. In Section V we discuss an upper bound on the number of computation steps it takes to execute a DDG, which will give us a lower bound for speed-up in an ideal system.

V. Lower Bound for Speed-up

Blumofe et al. [1] showed that the number of computation steps for any computation with total work T_1 and critical path length T_∞ is at most $T_1/P+T_\infty$ in a system with P processors and **greedy scheduling**. The definition of greedy scheduling is that at each computation step the number of instructions being executed is the minimum between the number of instructions that are *ready* and the number of processors available.

This bound fits our target executions, since we know how to calculate T_1 and T_∞ for DDGs and dataflow runtimes with work-stealing can be characterized as having greedy scheduling policies. We therefore shall use that upper bound on computation steps to model the performance of DDGs.

Actually, since we are interested in modeling the potential speed-up of a DDG, the expression we must use is the total amount of work T_1 divided by the upper bound on computation steps, which will give us a lower bound on speed-up:

$$S_{min} = \frac{T_1}{T_1/P + T_\infty}$$

It is important to be very clear about what this lower bound describes in order to understand the behaviour of the experimental results we present in the end of this chapter in comparison with the theoretical analysis. The upper bound on time proved by Blumofe et al. takes into consideration the number of computation steps it takes to execute the work using a greedy scheduling. Basically, it means that overheads of the runtime are not taken into account and there is the assumption that the system never fails to schedule for execution all instructions that are ready, if there is a processor available. Clearly that is not the situation we face when doing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

actual experimentation, since the dataflow runtime occurs in overheads besides the ones inherent to the computation being done. Consequently, this lower bound on speed-up S_{min} will serve us more as a guideline of how much the dataflow runtime overheads are influencing overall performance, while the upper bound S_{max} indicates the potential of the DDG. If the performance obtained in an experiment falls below S_{min} that indicates that the overheads of the runtime for that execution ought to be studied and optimized.

VI. Example: Comparing different Pipelines

In this section we will apply the theory introduced to analyze DDGs that describe an import parallel programming pattern: *pipelines*. We will take the DDGs of two different pipelines that have the same set of instructions and calculate the maximum degree of concurrency C_{max}, the maximum speed-up S_{max} and the minimum speed-up S_{min} for both of them. In the end of the section we will see that the theoretical bounds obtained correspond to the intuitive behaviour one might expect just by reasoning about both pipelines.

Figure 5a shows the DDG for the first pipeline and figures 5b and 5c represent the stages of the last iteration of Algorithm III for this DDG. In Figure 5b the DDG was unrolled four times and in Figure 5c the complement-path graph was obtained. Since the clique number of C_4 is the same as the clique number of C_3, the maximum degree of concurrency is $C_{max} = \omega(C_3) = 3$. To obtain S_{max} we apply the same principle introduced in IV. If we consider $w_b > w_a > w_c$, where w_u is the weight of an instruction u:

$$S_{max} = \lim_{k\to\infty} \frac{(w_a + w_b + w_c) \cdot k}{w_a + w_b \cdot k + w_c} = \frac{w_a + w_b + w_c}{w_b}$$

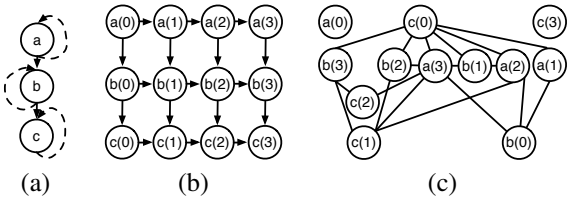


Fig. 5. Example of execution of Algorithm III in which the maximum concurrency converges.

Now we consider, in Figure 5, the same instructions connected in a pipeline, but this time the second stage of the pipeline (instruction b) does not depend on its previous execution. In this case, Algorithm III will reach $\omega(C_3) = 4$ and decide that C_{max} is **unbounded**. We then calculate S_{max}, obtaining:

$$S_{max} = \lim_{k\to\infty} \frac{(w_a + w_b + w_c) \cdot k}{w_a \cdot k + w_b + w_c} = \frac{w_a + w_b + w_c}{w_a}$$

And since $w_a < w_b$, the S_{max}, the average degree of concurrency for this DDG is greater than the average degree of concurrency in the the DDG of Figure 6. This corroborates the conclusion one might come to intuitively, since it makes perfect sense that a pipeline in which the slowest stage does not depend on its previous execution will most likely execute faster than one that is identical but has this restriction.

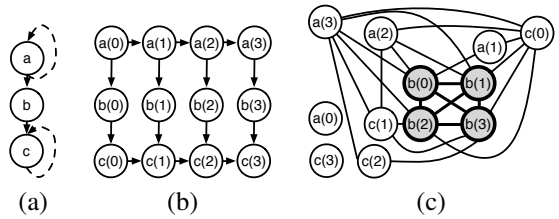


Fig. 6. Example of execution of Algorithm III in which the maximum degree of concurrency **does not** converge.

VII. Experiments

In this section we are going to present a series of experiments in which we compared the actual speed-up obtained for benchmark executions with the theoretical bounds for the DDG. In a first set of experiments we devised three artificial benchmarks representing different categories of DDGs. The choice for the adoption of artificial benchmarks in these experiments was due to the precise parametrization of the tasks' weights and the freedom to define the topology of the graphs. Such decision was important since our focus was to also provide empirical proof of the theory presented in this paper and artificial benchmarks allow us to put to test various dataflow scenarios, which would take a long survey of real applications to find ones that are adequate to the points we want to prove. Next we perform experiments with two real applications: Ferret, from the PARSEC Benchmark Suite [11] and a classic implementation of Conway's Game of Life. All results presented in this section came from executions of each benchmark in the Trebuchet Dataflow Runtime [12] running on a Intel Xeon® machine with 36 cores.

Information retrieved from Algorithm III was used to enhance Trebuchet's static scheduling. If two instructions are iteration independent, they do not get scheduled to the same PE, even if this means overriding the decision of the original static scheduler. This enhancement to Trebuchet's static scheduler, which is based on List Scheduling, is specially important for applications that display a pipeline pattern. In the original algorithm, the intra-iteration dependencies would hint the scheduler to map the pipeline stages to the same PE. The extra information provided by Algorithm III mitigates that problem, since the instances of the pipeline stages from different iterations are not dependent.

The Trebuchet runtime acts as facilitator for the parallelized execution in accordance to the dataflow paradigm. However, the same results can be expected if the benchmarks are implemented and executed using any other runtime or library that orchestrates parallel computation in a dataflow fashion. The only variation in performance for different implementations would be due to the overheads of the orchestrator itself, however, the trends in the behaviour of the results should be the same.

The first artificial benchmark is a pipeline depicted in Figure 7. The first and last stages of the pipeline ($read$ and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

$write$, respectively) are iteration-dependent, hence the return edges. The second stage ($proc$) is iteration-independent and, thus, it is also possible to exploit data parallelism by adding more $proc$ nodes to the graph that are each responsible for processing a part of the data received from the first stage. This DDG represents the functionality of the fork/join parallel programming pattern.

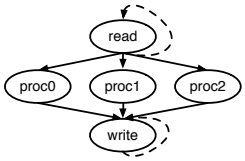


Fig. 7. DDG that describes the TALM implementation of a pipeline, considering that the number of available PEs for the *proc* stage is 3. The three instructions in the middle are the instances of the parallel super-instruction that implements the second stage of the pipeline.

To obtain the experimental results, we implemented the DDG using the THLL language and executed it on Trebuchet [12], varying the number of processing elements used and the parameters discussed above. To exploit the *data parallelism* of the middle stage we implemented it as a *parallel super-instruction*. Notice that by using parallel super-instructions we are reducing the granularity of the super-instruction, since each of its instances will process a portion of the data, thus the greater the number of instances, the finer the granularity. Typically, there will be as many instances of a parallel super-instruction as there are processing elements available in the system, so that the work will be spread among them. However, reducing the granularity of a super-instruction can change the critical path of the DDG, since the weight of that task is being reduced. This scenario will not be the case for the first experiment, since the first stage of the pipeline dominates the critical path (see Section VI).

Our model of computation expresses total execution time in terms of *computation steps*, which in turn should be represented in the same unit as instruction weights. In case of comparing theoretical bounds with real execution, the unit for computation steps and instruction weights should be something that can be measured in the experimental environment. In the case of simulations, one could use processor cycles for that purpose. However, since in the case of this work it is real execution on a multicore machine, our best option is to measure time. To obtain instruction weights we executed test runs of the application with profiling enabled in order to obtain the average execution time of super-instructions. The weight of fine grained instructions (instructions used for the inner workings of loops in Trebuchet, see [12]) was not taken into consideration and hence not represented in the DDG used for obtaining the theoretical bounds, because we consider them as part of the overheads of the TALM implementation.

The comparison of experimental results with the upper and lower bounds shows us how well the parallelism present in the application is being exploited by the dataflow runtime. A performance closer to the lower bound, or sometimes even below the lower bound, shows us that the dataflow implementation is not fully taking advantage of the potential for parallelism of the application and also may indicate that the overheads inherent to the dataflow synchronization are overwhelming the useful computation.

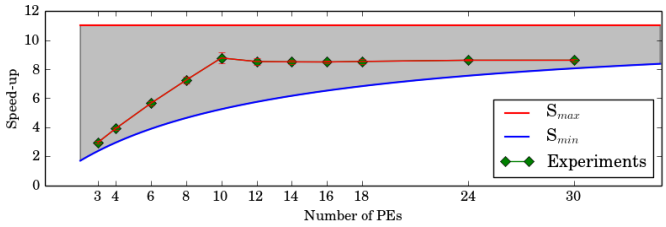


Fig. 8. Results for the pipeline with iteration-independent processing stage. The shaded area represents the potential for speed-up between S_{max} and S_{min}. S_{max} does not vary with the number of PEs.

Figure 8 shows the results for the first experiment. In this experiment we intended to show the asymptotic behaviour of DDGs enunciated in Theorem 2. The shaded area corresponds to the zone between the upper and lower bounds on speed-up, i.e. S_{max} and S_{min} respectively. The curve that represents S_{min} is simply the equation presented in Section V plotted as a function of the number of PEs. S_{max}, on the other hand, is plotted as a horizontal line because its equation is not a function of the number of PEs (see the equivalent example in Section VI), so it is just a constant function in this graph.

The results for this first experiment show us that the TALM implementation of the first benchmark achieves good performance, since it gets close to the upper bound S_{max}. The experiment also proves that the theoretical groundwork established is well-suited for actual dataflow execution, since the experimental results behave within the theoretical upper and lower bounds introduced.

For the second artificial benchmark we wanted to show the effect of forcing the middles stage of the pipeline to be iteration-dependent. For that purpose we included the dependency in the THLL code of the DDG, making it similar to the pattern of Figure 9. Now if $w_{proc} = max\{w_{read}, w_{proc}, w_{write}\}$, where w_{read}, w_{proc} and w_{write} are the weights for the super-instructions of the read, process and write stages respectively, the critical path is dominated by the iterations of the process stage. Therefore, in this scenario S_{max} is affected by the granularity of this super-instruction. Similar to the DDG of Figure 7, the more instances of the parallel super-instruction, the finer grained it gets. Furthermore, since the number of instances of a parallel super-instruction is dependent on the number of PEs available, the granularity and consequently S_{max} can be expressed as a function P, the number of PEs. As the parallel super-instruction gets more fine-grained it may be the case that a different super-instruction becomes the *heaviest*, and thus the critical path would change. The critical path T_∞ for this DDG can be expressed, for k

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

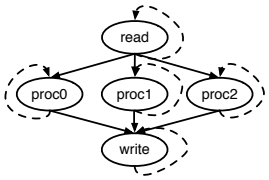


Fig. 9. DDG that describes the TALM implementation of a pipeline, considering that the number of available PEs for the *proc* stage is 3. In this DDG the middle stage (*proc*) is iteration-dependent. Since the middle stage is split among the PEs used, the granularity of that super-instruction is reduced, which in turn reduces the critical path length. Also, at some point the granularity of the middle stage becomes so fine that the critical path becomes the first stage, like in the original version. This explains abrupt changes in the lines.

iterations, as:

$$T_\infty(k) = \begin{cases} \frac{(w_{read}+w_{proc}+w_{write})\cdot k}{w_{read}+(w_{proc}/P)\cdot k+w_{write}} \\ \text{if } w_{proc}/P = max\{w_{read}, w_{proc}/P, w_{write}\}, \\ \frac{(w_{read}+w_{proc}+w_{write})\cdot k}{w_{read}\cdot k+w_{proc}/P+w_{write}} \\ \text{if } w_{read} = max\{w_{read}, w_{proc}/P, w_{write}\}, \\ \frac{(w_{read}+w_{proc}+w_{write})\cdot k}{w_{read}+w_{proc}/P+w_{write}\cdot k} \\ \text{otherwise.} \end{cases}$$

We can apply the same reasoning explained in Section VI to obtain $S_{max}(P)$, the maximum speed-up as a function of P, using the equation above and obtaining the limit for $S_{max}(P,k)$ as k approaches infinity. Hence the following equation:

$$S_{max}(P) = \frac{w_{read} + w_{proc} + w_{write}}{max\{w_{read}, w_{proc}/P, w_{write}\}}$$

Observe that, $S_{max}(P)$ only varies with P if the parallel super-instruction remains the heaviest. Figure 10 has the first results for this experiment. In the graph, S_{max} grows linearly with the number of PEs in the system until it reaches a point where $w_{read} > w_{proc}/P$ and becomes a constant function. S_{min} changes its behaviour for the same reason. The curve of the experimental results also has a knee, which reflects that in the actual execution the shift of the critical path is a real influence as projected by the theoretical bounds.

Furthermore, for this benchmark we opted to compare executions with and without dynamic scheduling (work-stealing, in this case) in order the observe the trade-offs of each approach. The speed-up for the curve "Static Only" begins below the S_{min} lower bound. This is by all means an intuitive result, since Trebuchet with work-stealing disabled does not fall into the category of parallel computation with *greedy* scheduling, which is a fundamental premise for the S_{min} bound. After reaching peek performance, the executions with work-stealing enabled show slowdowns, due to the extra synchronization overhead inherent to it. Most importantly, both versions follow the exact behaviour predicted with S_{max}, which is the main point to be proven here.

Ferret is an application that does image similarity search. We chose this application because since it is divided in stages (image read, similarty search and output) a pipeline

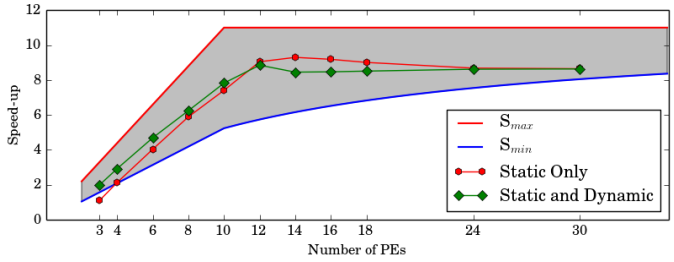


Fig. 10. Results for the pipeline with iteration-dependent processing stage. The shaded area represents the potential for speed-up between S_{max} and S_{min}. S_{max} varies with the number of PEs due to data-parallelism until the critical path of the DDG changes. Static/Dynamic are the results for the execution that utilized both static and dynamic scheduling. The results labled as Static Only came from executions where work-stealing was disabled in Trebuchet.

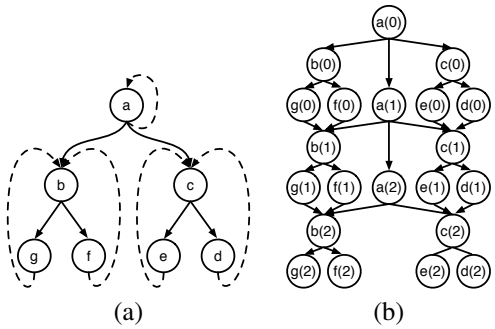


Fig. 11. DDG for a benchmark with topology similar to a binary tree with *return edges* added to represent inter-iteration dependencies.

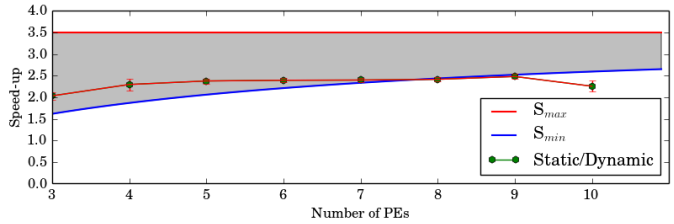


Fig. 12. Results for the "Tree" benchmark of Figure 11.

implementation is possible, thus it represents an important class of parallel algorithms. The dependencies present in Ferret are exactly like the ones in the DDG of Figure **??**. The first and last stages (image read and result output, respectively) depend on the previous iteration of their executions, but the middle stage (similarity search) is iteration-independent, i.e. one iteration does not depend on the previous one.

To obtain the experimental results, we implemented a dataflow version of Ferret using our THLL language and executed it on Trebuchet, varying the number of processing elements used. Since the middle stage of the application is iteration-independent, we can also exploit *data parallelism*, so we implemented it as a *parallel super-instruction*. Figure 13 shows the results for Ferret with and without static scheduling.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

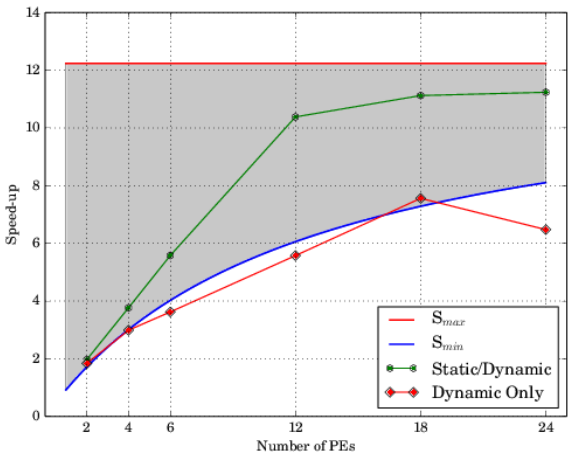


Fig. 13. Results for the original version of Ferret. The shaded area represents the potential for speed-up between S_{max} and S_{min}. S_{max} does not vary with the number of PEs. Static/Dynamic are the results for the execution that utilized both static and dynamic scheduling, Dynamic Only is the execution with just work-stealing, S_{max} is the upper bound for speed-up and S_{min} the lower bound.

Results show us that when using both static scheduling and dynamic scheduling, the TALM implementation of the algorithm achieves good performance, since it gets close to the upper bound S_{max}. On the other hand, the execution that had to rely only on dynamic scheduling performed poorly, even falling below the lower bound S_{min}. The poor performance of this execution can be attributed to the overheads of the work-stealing implementation and the contention on the concurrent data structure (the double ended queue).

VIII. Eliminating the Restrictions of DDGs

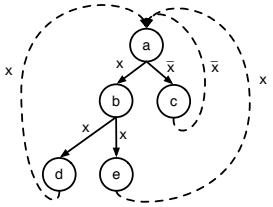
As mentioned in Section II, the restrictions imposed to DDGs may be eliminated in order to make the model generic, allowing it to represent any dataflow program. The reason we chose not to relax the restrictions in the previous sections was to ease the formalization of the model and the related proofs. In this section, however, we will introduce how each of the three restrictions can be eliminated.

A. Conditional Execution

The first restriction has to do with conditional execution. In order to allow DDGs to have conditional execution, just like any dataflow graph, we will have to take into consideration each possible execution path when calculating T_1 and T_∞ for the DDG. Basically, the critical path T_∞ and the total work done T_1 both depend on the boolean control expressions evaluated during execution (i.e. the equivalent of the branches taken/not taken in Von Neumann execution). To address this first restriction we extend the DDG's with the concept of *Conditional* DDG's (CDDG). Let a CDDG be a tuple $D = (I, E, R, W, X, \phi)$, where:

- I, E, R, W are the same as in the original DDG definition
- X is a set of boolean variables used to express the logical conditions for data to be sent through each edge.
- $\phi : E \rightarrow F(X)$ attributes a logical expression to each edge of the graph.

With this definition, we have that the logical expression for an edge $e \in E$ is the boolean function $\phi(e) \in F(X)$, where $F(X)$ is the set of all boolean functions using the variables in X. This way, if $\phi(e)$ evaluates to **True**, data will be carried through edge e. Figure 14 shows an example of a loop with an *if* statement inside its body, which causes the subgraphs corresponding to the *then* and *else* clauses to be mutually exclusive.



e	$\phi(e)$
(a, b)	x
(a, c)	$\overline{x}$
(b, d)	x
(d, a)	x
(e, a)	x
(c, a)	$\overline{x}$

Fig. 14. Example of Conditional DDG representing a loop with an *if* statement inside its body. Since $y = \overline{x}$, we have mutually exclusive sets of instructions in each iteration, which represent the *then* and *else* clauses.

In order to apply our methodology to Conditional DDG's, we also introduce the concept of *projections* of CDDG's. A projection of a CDDG $D = (I, E, R, W, X, \phi)$ is the attribution of a value $\alpha \in \{0, 1\}$ to a variable $x \in X$ in an iteration i (represented as $D\ |_{x(i)=\alpha}$). For instance, if we unroll the CDDG k times, we can attribute a boolean value to a variable $x \in X$ in each of the k iterations. A *complete projection* of a CDDG unrolled k times is a projection such that all variables have boolean values attributed to them. Notice that complete projections can be obtained by accumulating the projection operation, i. e., $D\ |_{x_0=\alpha_0}|_{x_1=\alpha_1}\ ..\ |_{x_n=\alpha_n}$, where each x_i is a variable in the unrolled CDDG. For the sake of simplicity, we define the function $\psi : X_k \rightarrow \{0, 1\}$ which attributes a boolean value to each variable in X_k (the set of variables obtained when unrolling the CDDG k times). This way, we say that, if $\phi\ |_\psi\ (e_k) = 1$, data will be carried through edge e_k in the unrolled CDDG because its condition evaluates to true.

With this definition of complete projections, we can reduce unrolled Conditional DDG's to unrolled DDG's simply by applying a complete projection to D_k, a CDDG unrolled k times, and then removing all edges e_i such that $\phi\ |_\psi\ (e_i) = 0$ and all instructions I_k that become unreacheable in the complete projection (meaning that such instance of the instruction would not be executed in the scenario represented by the attribution function ψ).

Now, we can adopt these definitions to adapt Algorithm III for Conditional DDG's. The basic idea is to consider all

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

possible complete projections $D_k \mid_\psi$ in each iteration and reduce the conditional graph to a simple dataflow graph by removing edges whose expression $\phi \mid_\psi$ evaluates to 0. Having done that, we pick the unrolled graph with the maximum $\omega(C)$ and consider that the maximum degree of concurrency for the current iteration. The same rules for continuing onto the next iteration of Algorithm III or for terminating it apply, as we will show below.

As for S_{max} and S_{min} we have to move toward a probabilistic view, treating the dataflow execution as a stochastic process. A complete projection of a CDDG unrolled k times represents an execution scenario where each conditional branch is considered taken or not taken. We can use this complete projection to obtain $L_k \mid_\psi$, the critial-path, and $W_k \mid_\psi$, the weight of the instructions that get executed, for the corresponding scenario. As mentioned before, the instructions that remain on the unrolled graph after a complete projection are those that are still reacheable, which can be formalized as follows: an instance of instruction $u \in W_k$ is reacheable in a complete projection with attribution function ψ if the logic-OR of all $\phi \mid_\psi (e), e = (\cdot, u)$, evaluates to true. If we can assume uniform distribution for the probability density function of each boolean variable $x \in X_k$ in the CDDG unrolled k times, we can estimate the bounds for speed-up as follows. Given $L_k \mid_\psi$ and $L_k \mid_\psi$ we can obtain $E(S_{max})$ and $E(S_{min})$ for each complete projection ψ using the probability p_ψ of that projection, i.e., the probability of each variable having the value attributed in the projection. $E(S_{max})$ is given below and $E(S_{min})$ can be calculated analogously using the definition from Section V.

$$E(S_{max}) = \sum_{\psi} \left(\lim_{k \to \infty} \frac{\sum_{w_i \in W_k \mid_\psi} w_i}{L_k \mid_\psi} \right) \cdot p_\psi$$

Figure 15 presents an example of CDDG and the process to unroll it and obtain a complete projection. Although we only show one complete projection, in (c), there are 2^{2k} possible complete projections since we have two boolean variables in the CDDG, x and y. Each of these complete projections must be taken into account, with their corresponding probabilities, to calculate C_{max}, $E(S_{max})$ and $E(S_{min})$.

B. Dependencies Over Multiple Iterations

The restriction which states that dependencies can only be carried from one iteration to the next one can be relaxed by starting with the DDG unrolled as many times as the longest distance (in iterations) among the loop carried dependencies in the graph. Consider the loop and the corresponding DDG in Figure 16. The integers used as labels for the return edges represent the distance (in iterations) corresponding to the edge , i.e., if the label is d the dependency is carried from iteration i to iteration $i + d$.

Figure 16 shows the first step for adapting both Algorithm III and the calculation of the bounds for speed-up to overcome the restriction on the distance of loop-carried dependencies. Since, the longest distance in inter-iteration dependencies is 3 (the return edge from c to c itlsef), we unroll the DDG 3 times.

For Algorithm III we also have to alter the stop condition (step 8 of the pseudo-code). First, as Figure 16b cleary shows, $\omega(C_k) > |I|$ is not a sufficient condition to consider C_{max} unbounded. Therefore, we refine this condition to the following: if there is an instance of an instruction $u(j)$, if there is no $u(i), i < j$ such that $u(i) \rightarrow u(j)$, than C_{max} is unbounded. Note that this condition is only applicable if we already started by unrolling the DDG d times, where d is the longest inter-iteration distance of a dependency in the original DDG. As to stopping the algorithm after asserting that C_{max} is bounded to $\omega(C_k)$, the rule stays the same.

For nested the loops, a simple variation of the same approach can be adopted. Since nested loops can be represented as flat loops that carry dependencies over a iteration distance equal to the number of iterations in the inner loops.

IX. Conclusion

In this work we present concurrency analysis techniques that extend prior work that focused only on DAGs. We presented a definition of Dynamic Dataflow Graphs (DDGs), which we used as our model of computation for the analysis. This new model of computation allowed us to abstract details of dataflow implementation, since DDGs are more simple and easier to analyze than TALM graphs, for instance, and gain important insight into the behaviour of dataflow parallelization. We showed how to obtain two important metrics via analysis of DDGs: the maximum speed-up (or average degree of concurrency) S_{max} and the maximum degree of concurrency C_{max}. We also were able to present an important result, the fact that the former is bounded even if the latter is not. Our experiments with a benchmark that represented the *streaming* parallel programming pattern showed us that our analysis indeed is able to model and predict the performance behaviour of applications parallelized with dataflow execution.

An important lesson learned with this work: all the information that describes the parallel programming pattern in the program is in the dataflow graph, i.e. the data dependencies in the program. Therefore, high-level annotations or templates that specify what programming pattern is adopted are not necessary. For instance, using our techniques it is not necessary to specify (or use a special template) that a certain application has a pipeline pattern, the programmer just has to describe the dependencies and the static scheduling algorithm will be able to determine to number of PEs to allocate (using C_{max}) and how to spread the pipeline stages among them.

Acknowledgments

The authors would like to thank FAPERJ, CNPq and CAPES for the financial support to this work.

References

[1] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

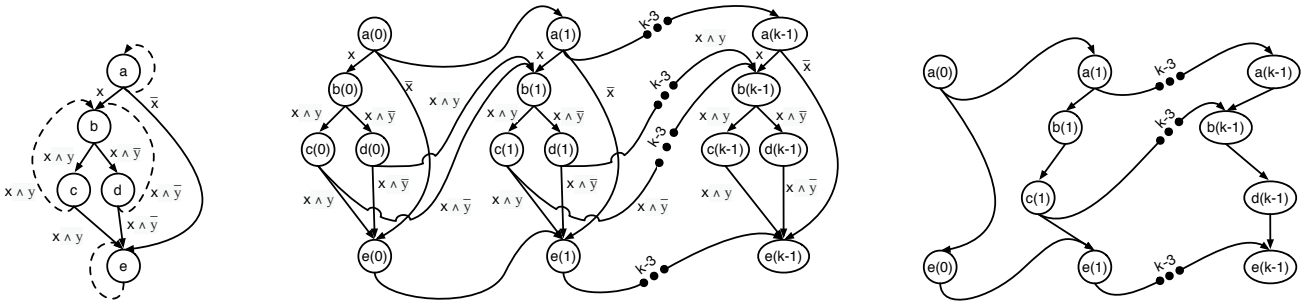


Fig. 15. A Conditional DDG with two nested loops, the innermost inside the *then*-clause of the outmost, (a) and the graphs obtained after unrolling it k times (b) and applying a complete projection (c).

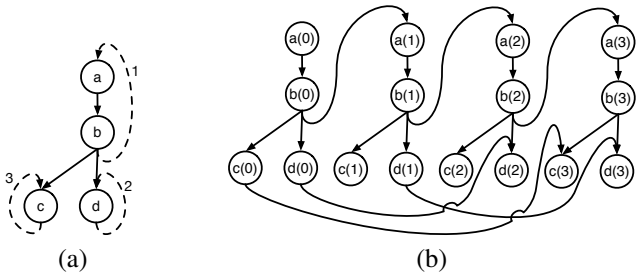


Fig. 16. DDGs where dependencies are carried over a distance than more than one iteration.

[2] D. Eager, J. Zahorjan, and E. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Transactions on Computers*, vol. 38, no. 3, pp. 408–423, Mar. 1989.

[3] B. Kumar and T. A. Gonsalves, "Modelling and analysis of distributed software systems," in *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, ser. SOSP '79. New York, NY, USA: ACM, 1979, pp. 2–8.

[4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread Scheduling for Multiprogrammed Multiprocessors," *Theory of Computing Systems*, vol. 34, no. 2, pp. 115–144, Jan. 2001.

[5] W. F. Boyer and G. S. Hura, "Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1035–1046, Sep. 2005.

[6] M. Lombardi and M. Milano, "Scheduling conditional task graphs," in *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, ser. CP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 468–482.

[7] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175–187, Feb 1993.

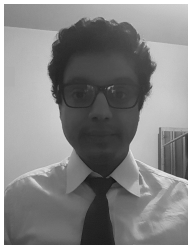
[8] M. T. Anticona, *A GRASP algorithm to solve the problem of dependent tasks scheduling in different machines*. Boston, MA: Springer US, 2006, pp. 325–334.

[9] A. V. Aho and J. D. Ullman, *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1977.

[10] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[12] 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.



Tiago A. O. Alves is a Professor of Computer Science at the State University of Rio de Janeiro. He received his BSc in Computer Science, his MSc and his DSc in Computer Systems Engineering from the Federal University of Rio de Janeiro. His main areas of interest are Parallel Programming with focus in Dataflow, distributed algorithms and resilient systems.

Leandro A. J. Marzulo is a Professor of Computer Science at the State University of Rio de Janeiro. He received his BSc in Computer Science from the State University of Rio de Janeiro, his MSc and his DSc in Computer Systems Engineering from the Federal University of Rio de Janeiro. His areas of interest are Computer Architecture, Dataflow, Parallel Programming, Compilers and Distributed Algorithms.

Sandip Kundu is a Professor of Electrical and Computer Engineering at University of Massachusetts, Amherst. Previously, he was a Principal Engineer at Intel Corporation and Research Staff Member at IBM Corporation. He has published more than 200 papers in VLSI design and CAD, holds 12 patents, and has co-authored multiple books He served as the Technical Program Chair of ICCD in 2000, co-Program Chair of ATS in 2011, ISVLSI in 2012 and 2014, DFT in 2014. Prof. Kundu is a Fellow of the IEEE and has been a distinguished visitor of the IEEE Computer Society.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2799078, IEEE Transactions on Emerging Topics in Computing

Felipe M. G. França received his BSc in Electrical Engineering from Universidade Federal do Rio de Janeiro (1982), M.Sc. in Computer Science from Universidade Federal do Rio de Janeiro (1987) and Ph.D. in Neural Systems Engineering from Imperial College of Science Technology And Medicine (1994). He is Professor (Full) at Universidade Federal do Rio de Janeiro. Has experience in Computer Science and Electronics Engineering, acting on the following subjects: artificial neural networks, complex systems, computer architecture, distributed algorithms, computational intelligence, collective robotics, complex systems, and intelligent transportation systems.