Scheduling Cyclic Task Graphs with SCC-Map

Alexandre Sardinha*, Tiago A. O. Alves*, Leandro A. J. Marzulo†,
Felipe M. G. França*, Valmir C. Barbosa* and Vítor Santos Costa‡
*Universidade Federal do Rio de Janeiro
Programa de Engenharia de Sistemas e Computação, COPPE, Rio de Janeiro, RJ, Brasil
Email:{sardinha, tiagoaoa, felipe, valmir}@cos.ufrj.br
†Universidade do Estado do Rio de Janeiro
Instituto de Matemática e Estatística, Departamento de Informática e Ciência da Computação, Rio de Janeiro, RJ, Brasil
Email: leandro@ime.uerj.br
‡Universidade do Porto
Departamento de Ciência de Computadores, Porto, Portugal
Email: vsc@dcc.fc.up.pt

***Abstract*—The Dataflow execution model has been shown to be a good way of exploiting Thread-Level Parallelism (TLP), making parallel programming easier. In this model, tasks must be mapped to processing elements (PEs) considering the trade-off between communication and parallelism. Previous work on scheduling dependency graphs have mostly focused on directed acyclic graphs, which are not suitable for dataflow (loops in the code become cycles in the graph). Thus, we present the SCC-Map: a novel static mapping algorithm that considers the importance of cycles during the mapping process. To validate our approach, we ran a set of benchmarks using our dataflow simulator varying the communication latency, the number of PEs in the system and the placement algorithm. Our results show that the benchmark programs run significantly faster when mapped with SCC-Map. Moreover, we observed that SCC-Map is more effective than the other mapping algorithms when communication latency is higher.**

I. Introduction

Recent work has pointed at the Dataflow execution model as a good alternative to exploit thread-level parallelism [1]–[3]. In the Dataflow model, programs can be described as a graph, where nodes represents *Instructions* (or tasks) and edges in the *graph* describe their dependencies. Execution is guided by the dataflow firing rule: instructions are fired as soon as all of their input operands are ready (i.e., all of their parents have completed).

A problem that arises from this strategy is the need to map instructions to processing elements (PEs). Once you have described the program dependencies in a dataflow graph, you must decide where each instruction will be placed (i.e., which available PE will execute which instruction). A good mapping must balance the fact that the more instructions are spread among PEs, the more parallelism will be available, but also the more communication overhead one will have. Therefore, an ideal scheduling strategy must aim at obtaining a good trade-off between communication and parallelism. Throughout this work, we use the terms "map" and "schedule" interchangeably.

Previous work on scheduling dependency graphs have mostly focused on DAGs (directed acyclic graphs), with good results being achieved for statical (or offline) mapping [4]–[8]. However, mapping algorithms for DAGs are not suitable for dataflow graphs, since dataflow programs often contain cycles corresponding to the loops in the code. We present the SCC-Map: a new static mapping algorithm for dependency graphs that contain cycles (namely, for dataflow graphs). Our proposal is based on the work of Boyer and Hura [6], which was aimed at DAGs. We further compare our novel approach towards dataflow graphs with other algorithms, such as the ones presented in [6], [7].

In order to validate our ideas, we developed a cycle-by-cycle dataflow simulator, allowing us to investigate in detail the effects of each mapping strategy. Then, we compiled a set of benchmarks to run on the simulator varying the placement algorithm, communication latency, and the number of available processing elements in the system. Mappings were obtained by a set of reference algorithms and SCC-Map. We compared the speedups in all scenarios, taking as baseline the serial execution, i.e., the case where all instructions mapped to the same PE. Moreover we provide a theoretical maximum speedup that could be achieved for each application. This value was obtained by placing each instruction of the program in a distinct PE and setting the communication latency to 1 clock cycle (minimum possible latency).

Our results show that, for most of the tested scenarios, our set of programs run significantly faster with the instruction-to-PE mappings obtained with our algorithm. Moreover, we observed that SCC-Map is more effective than the other mapping algorithms when communication latency is higher in the system.

The rest of this paper is organized as follows: Section II discusses the relevance of the mapping problem in the context of dataflow systems and presents TALM (the dataflow model used as basis to build the simulator to test SCC-Map); in Section III we discuss some related works; Section IV presents SCC-Map (our mapping algorithm); results are presented and discussed in Section V; we conclude and indicate possible future works in Section VI.

978-0-7695-4916-3/12 $26.00 © 2012 IEEE
DOI 10.1109/WAMCA.2012.8

II. The Dataflow Task Scheduling Problem

Many algorithms have been proposed for static scheduling of tasks (or instructions) onto processing elements (PEs), such as [4]–[7]. Those solutions have considered only DAGs (directed acyclic graphs), or just ignored the existence and influence of cycles. The focus of this work is the mapping of tasks in a dataflow program. In this case, loops in the program are translated into cycles in the related dataflow graph. Therefore, the influence of cycles is a determinant factor in program execution.

TALM (TALM is an Architecture and Language for Multithreading) is a model designed to use dataflow principles of execution to exploit thread-level parallelism in Von Neumann machines, using traditional imperative languages [1], [2]. This is achieved by allowing the definition of coarse-grained instructions, or *super-instructions*, as an extension of the standard dataflow instruction set. These *super-instructions* are connected in a dataflow graph, which may also contain simple fine-grained instructions, and the dataflow firing rule is used to naturally paralellize program execution. The dataflow firing rule is the basic concept behind dataflow architectures and goes as follows: an instruction may be fired (dispatched for execution) as soon as all its input operands are ready. Thus, if two instructions have no precedence relation between them they may execute concurrently, if they are not mapped to the same PE. Similarly to other dataflow models, in TALM loops are represented as cycles in the dataflow graph and the iterations are controlled by special instructions, namely the `steer` and the `inctag` instructions [1], [2].

TALM has been implemented as a runtime system for multicore machines: the Trebuchet [1], [2]. Moreover, the Couillard Compiler [1] has been developed to transform annotated C programs into TALM's dataflow graph. Since Trebuchet's processing elements are implemented as threads in the host machine, instructions mapped to different PEs will run concurrently, in different threads. Thus, we can say that Trebuchet has the ability to convert instruction-level parallelism, into thread-level parallelism.

In order to observe the effects of different mappings and to evaluate SCC-Map, we developed an architectural dataflow simulator based on the TALM execution model. The architecture of the simulator consists of a set of processing elements (PEs) connected by an interconnection network. For simplicity, and since hierarchical architectures are not the main focus of this current work, the simulated network topology is a complete graph. We thus assume that the communication latency between all pairs of PEs is the same. Each processing element is comprised by an Input Buffer, an Instruction List, a Matching Table, a Ready Queue and an Execution Unit.

The Input Buffer is a FIFO queue used to receive operands from other PEs. The Instruction List is the in-memory program of the PE and contains all instructions that were mapped to the PE, along with information about the number of operands each particular instruction must receive to execute and the set of instructions that will receive operands produced by the instruction. The Matching Table is a store where the operands retrieved from the Input Buffer are placed. Operands that are designated to the same instruction and have the same *iteration tag* are stored together. Once the number of stored operands with the same destination instruction and iteration tag is equal to the number of necessary operands for that destination instruction, the instruction is dispatched to execution along with the operands. The Ready Queue is a queue that stores the instructions whose input operands of the same iteration have all been received (in the Matching Table). Each cycle, if the Ready Queue is not empty, the PE retrieves one instruction from the queue and dispatches it to the Execution Unit, where *ready instructions* are actually executed.

The simulator receives as input a file containing the following: (*i*)the dataflow graph to be executed, (*ii*) the instruction-to-PE mapping configuration, (*iii*) architectural parameters such as communication latency time (the number of cycles it takes to send inter-PE operands) and (*iv*)the execution time (in cycles) of each instruction of the instruction set.

III. Related Work

Instruction mapping is a major problem in dataflow architectures. We will refer to recent work next. In [7], Mercaldi et al. proposed eight different instruction mapping strategies for the WaveScalar Architecture [9]. Since WaveScalar is a grid of Processing Elements organized hierarchically, the focus of the strategies in [7] is to take advantage of the communication hierarchy of the grid. Since communication costs may vary between PEs, these strategies aim at keeping instructions that exchange data (i.e. dependent instructions) the closest possible on the grid. In our model, the communication costs are equal among all PEs, so this is not one of our main concerns, that being the reason for us to diverge from this spatially oriented approach.

In [5], an heuristic to enhance list scheduling was presented for interconnection-constrained architectures. The priority each attributed to each task during the scheduling receives a dynamic level, which is updated at each step of the algorithm, depending on the tasks that were already mapped to the processors. This way, the decision making of the algorithm is refined, since it is not only based on characteristics of the graph, but also on the current estimated state of the processing and communication resources.

In [8] an approach that is similar to the dynamic-level approach of [5] is presented. The algorithm is also based on two phases of prioritizing: *(i)* static, based on characteristics of the graph itself, and *(ii)* dynamic, depending on the state of the resources at the moment of each iteration. However, one key advantage of this work is that it tries to find slots to allocate tasks between slots of instructions that have already been scheduled. That is, if you have two tasks scheduled to a processor and an idle time between them (typically because the second one has to wait for data from its predecessors) the algorithm will try to schedule a third task between them.

A model of computation that is closer to ours is the focus of the algorithm proposed by Boyer in [6], which also takes

advantage of dynamic priorities. The main difference between the problem addressed by Boyer and the dataflow instruction mapping problem is that in [6] programs are only described by DAGs (direct acyclic graphs) and in the dataflow model we may encounter cycles in the graphs, which correspond to loops in the program. Therefore, this approach is also not entirely adequate for our target systems.

IV. SCC-Map: Scheduling Tasks in Dataflow Graphs

As explained in the previous sections, algorithms such as the one proposed in [6] do not directly apply to dataflow systems, as dataflow graphs may have cycles. Therefore, we needed to develop a way to deal with cycles if we were to adopt a variation of such algorithms.

A. Strongly Connected Components

As a preliminary investigation of the properties of mappings, we selected some graphs with up to 12 nodes (which yields 4.213.597 possible mappings) and ran all of the possible mappings for each of these graphs on our simulator. The observation of the optimal mappings obtained in this investigation suggested that *the nodes in strongly connected components (SCC) of the dataflow graph should be mapped to the same PE*. Intuitively, nodes in an SCC represent a set of instructions such that each one has dependencies on all of the other instructions of the set, so there is a limitation for the amount of parallelism between them. Based on this heuristic we made the decision to change the focus to mapping the SCCs of the graph to PEs, instead of mapping each task individually. It follows from this decision that the graph used as input for SCC-Map would be $G_{SCC}(I', E')$ obtained from the original graph $G(I, E)$ as follows. First, we derive I' as the set of SCCs present in G and E' as the set of edges between them such that $(A, B) \in E'$ if $A, B\} \subseteq I'$ and $\exists(i, j) \in E$ such that $(i \in A \wedge j \in B)$. It is important to notice that the decision to map SCCs instead of mapping each instruction individually also solves the obstacle we had for using DAG-targeted scheduling algorithms on dataflow graphs, since G_{SCC} has no cycles (i.e. is a DAG). In the next sections we refer to specific SCCs by listing the set of tasks that belong to it, e.g., $\{1, 2, 4\}$ is a SCC comprised by tasks 1, 2 and 4.

B. Task priority

In the algorithm proposed in [6], the next task to be mapped to a PE is randomly chosen from the *ready list* (the list of tasks whose predecessors have all been already mapped). In order to obtain better mappings, the algorithm is executed multiple times (according to a stop criteria) and the solution with the minimum makespan is chosen. On the other hand, in [5], the next task is chosen based on a priority level, defined as the size of the longest path from the target task to another task that is a leaf in the DAG. For efficiency reasons and in accordance with our preliminary investigations, we chose to follow the path of [5] and adopt priority heuristics for selecting the next instruction to be mapped.

The simulations with small graphs described in IV-A hinted three heuristics which we adopted as the tasks priorities (in the order they are presented here): task level, task *fan-out* and task *fan-in*. The task level is the size of the longest path from the task to another task that is a leaf in the graph, the task *fan-out* is the number of edges coming out of the task and the task *fan-in*: the number of edges going into the task.

C. Custom SCC finish time

In order to estimate the earliest start time of a task in a certain mapping it is necessary to obtain an estimation of the maximum finish time between its predecessors, since a task can only execute after receiving all of its input operands. If we consider the entire execution time of the parent SCCs when mapping a task, we may end up overestimating the time when the target SCC may start executing. This effect can be observed in the example of Fig. 1. Notice that, since we are going to map the elements of an SCC to the same PE, the execution time of each SCC is going to be the sum of the estimated execution times of all tasks in the SCC. In the graph of Fig. 1, task 5 can start execution as soon as it receives the operand from task 2, so, in this case, using the execution time of the entire SCC $\{1, 2, 3, 4\}$ to estimate the start time of 5 would be an overestimation.

Consequently, in SCC-Map, when we calculate the start time of an SCC that we want to map, the estimated execution time of its predecessors must be customized. In other words, the estimated execution time of an SCC A may have one value when we are mapping an SCC B that depends on A and a different value when we are mapping another SCC C that also depends on A. We shall represent with T_{AB} the estimated execution time of an SCC A customized for mapping the SCC B. T_{AB} is defined as the maximum path between an *entrypoint* in A and a task in B that receives an edge from A, where an *entrypoint* is a task in the SCC that has an input edge coming from another SCC. For instance, in the example of Fig. 1, task 1 is the only *entrypoint* in $\{1, 2, 3, 4\}$ and $T_{\{1,2,3,4\},\{5,6,7\}} = 2$, since $distance(1, 5) = 2$.

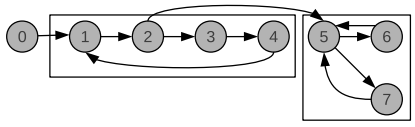

Fig. 1. Example of finish time overestimation.

D. Algorithm description

In this section we will describe the final form of our algorithm, which, as mentioned, is a variation of the one presented in [6] with the adaptations presented to make it suitable for dataflow graphs with cycles. The algorithm receives as input the dataflow graph $G(I, E)$, where I is the set of tasks/instructions and E is the set of direct dependencies, the set of processors P and the estimated execution time t_i of each instruction i. The first step in the algorithm is to obtain the

G_{SCC} graph, isolating the SCCs of the input G graph. Next, we calculate the priority levels of each SCC according to the heuristics presented in IV-B. The final step of the algorithm's initialization is to populate the *ready list* with all the SCCs that have no *entrypoints* and, therefore, can be executed from start, without needing to receive input operands.

Input: Task Dependency Graph $G(I, E)$, the set of processors P and the estimated execution times t_i of each task i
Output: A mapping of $E \rightarrow P$
  Calculate the priorities of all tasks in G
  Create the graph of SCCs $G_{SCC}(I', E')$
  Initialize *ready* list
  while $ready \neq \emptyset$ **do**
    remove the SCC A with the highest priority from *ready*
    MAP(A)
    Add to *ready* the SCCs that have become ready after this
  end while
  procedure MAP(A)
    $minmakespan \leftarrow \infty$
    for all PE $p \in P$ **do**
      $S_A \leftarrow max(S_B + T_{BA} + L_{BA})$
      $F_A \leftarrow max(S_A, max(F_Z(p))) + T_A$, where Z was mapped to p
      if $F_A < minmakespan$ **then**
        $minmakespan \leftarrow F_A$
        $chosenproc \leftarrow p$
      end if
    end for
    if $minmakespan > makespan$ **then**
      $makespan \leftarrow minmakespan$
    end if
    map A to $chosenproc$
  end procedure

Fig. 2. Algorithm for Scheduling Task Graphs with Cycles

The core of the algorithm consists of a loop that iterates over the *ready list* removing from it the SCC with the highest priority (following the criteria chosen in IV-B), maps it to the PE that minimizes the resulting *makespan* and pushes into the list the SCCs that have become ready (the ones whose predecessors have all been mapped) in this iteration. The resulting *makespan* if the SCC is mapped to a PE p is calculated as follows: *(i)* define S_A, the start time of the SCC A, as the maximum value of $S_B + T_B A + L_{BA}$, where B is a predecessor of A and L_{BA} is the latency time between the PE to which B is mapped and p ($L_{ji} = 0$ if j was mapped to p); *(ii)* if $S_A < max(F_Z(p))$, where $F_Z(p)$ is the finish time of an SCC Z mapped to p, define $F_A := max(F_Z(p)) + T_A$, otherwise, make $F_A := S_A + T_A$; *(iii)* if $F_A > makespan$, the resulting *makespan* will be F_A

The PE chosen to map the SCC is the one that yields the minimal resulting *makespan*. In Fig. 2 we present the complete pseudo-code form of the algorithm.

V. Experiments and Results

In this section we describe the methodology adopted to validate our task mapping algorithm (SCC-Map) and compare it with others.

A. Benchmarks

Since our focus are dependency graphs with cycles, namely dataflow programs with loops, we developed a set of kernels containing independent loops, nested loops and a combination of both. In order to demonstrate that our algorithm would not perform poorly with acyclic graphs, in comparison with the algorithms designed for those types of graphs, we also implemented kernels that had no loops. All kernels were written in C language and compiled to dataflow graphs with the Couillard C Compiler [1].

The proposed kernels are composed by smaller blocks of codes with different characteristics. Acyclic blocks (A) perform the calculation of a big mathematical expression (which contains no loops) and would comprise the acyclic portions of the benchmarks. Cycle blocks (C) are a loop that calculates a summation. Nested cycle blocks (NC) comprise two nested loops.

These blocks were connected in different ways to build our benchmarks. Basic (B) means just one building block. Serial connection (S) means we have four blocks of the same type connected serially. In a Parallel connection (P) four blocks of the same type placed in parallel with their results being joined hierarchically, therefore their execution is independent. A Serial/Parallel connection (SP) is a combination of the Serial and Parallel categories, where two groups of blocks are connected in parallel, and each group has two blocks serially connected.

TABLE I
Benchmarks detailed information

Bmark	Inst. #	SCC #	max(SCC)	avg(SCC)	var(SCC)
A-B	135	135	1	1	0
A-P	540	540	1	1	0
A-S	537	537	1	1	0
A-SP	538	538	1	1	0
C-B	10	5	4	2	2
C-P	40	20	4	2	1.68
C-S	61	29	4	2.1	1.02
C-SP	46	22	4	2.09	1.41
NC-B	28	10	15	2.8	19.95
NC-P	112	40	15	2.8	18.42
NC-S	228	85	15	2.69	9.76
NC-SP	150	54	15	2.77	14.13
MIX	175	152	15	1.15	1.46

Each benchmark is named by concatenating its building block type with its connection type (and the same is done with its acronym), e.g., "Nested Cycle Basic" (NC-B). The "Mixed" benchmark (MIX) is a combination of all building blocks, using one of each. Table I shows more detailed information about each benchmark, namely, the instruction count (Inst #), the number of SCCs (SCC #), the number of instructions (or size) of the biggest SCC (max(SCC)), the average SCC size

(avg(SCC)) and the variance of the SCCs sizes (var(SCC)). Notice that the number of instructions is an upper-bound of the number of PEs that can be used by a benchmark (each instruction could be mapped to a different PE). Moreover, for our algorithm, as the mapping unit is a SCC, the number of PEs is limited by the number of SCCs.

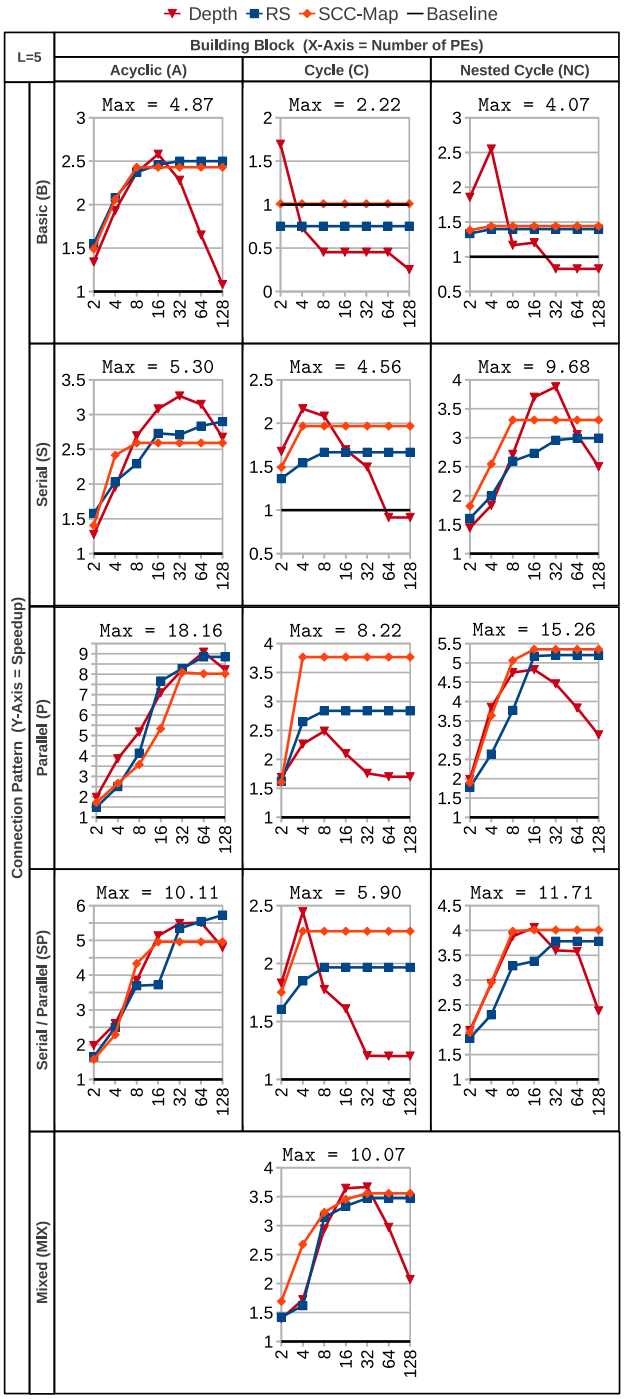

Fig. 3. Results (Communication Latency=5 cycles).

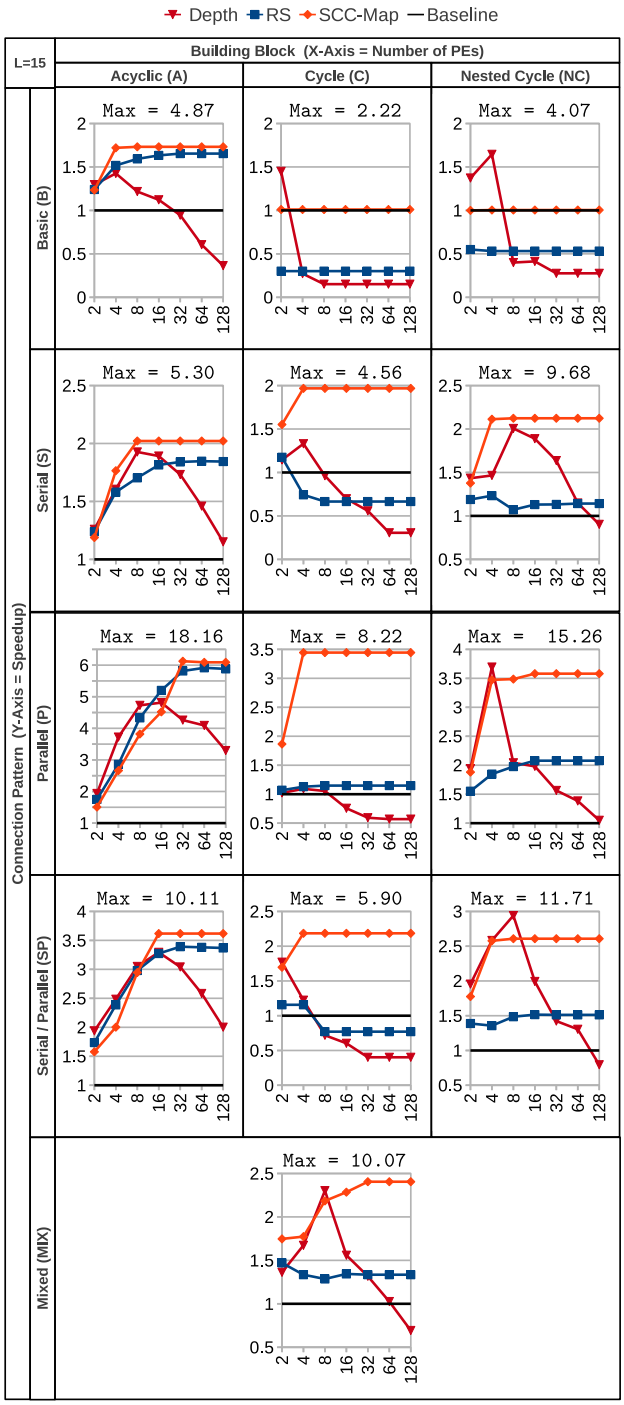

Fig. 4. Results (Communication Latency=15 cycles).

B. Comparison with other algorithms

The algorithms chosen for comparison with our work were the Random Search algorithm [6] and the Depth-first-snake algorithm [7]. In the results we refer to them simply as RS and Depth, respectively.

For all experiments we used our simulator to collect execution time (in clock cycles), for each benchmark (described in Section V-A), varying the inter-PE communication latency (5 and 15 clock cycles), the number of the available PEs in the system (2, 4, 8, 16, 32, 64 and 128) and the scheduling algorithm (Depth, RS and SCC-Map). The benchmarks are categorized according to the used building block and the block connection types (described in Section V-A).

In Fig. 3 we present the results collect with our simulator, after setting up inter-PE communication latency to 5 clock cycles. Each chart shows the speedups achieved for one of the benchmarks, using each of the aforementioned mapping methods, taking as baseline the serial execution (all instructions in the same PE). In each chart, the y-axis and x-axis are related, respectively, to the speedups and to the the number of PEs. Moreover the value labeled as "Max" is a theoretical maximum speedup that could be achieved for that application. This value was obtained by placing each instruction of the program in a distinct PE and setting the communication latency to 1 clock cycle, meaning that inter and intra-PE communication overheads are the same. Naturally, for these executions the number of PEs in the simulator was set to the number of instructions. We used this value as an upper-bound for comparison, since it is the best possible speedup for each application. Fig. 4 provides the results, when inter-PE communication latency is set to 15 clock cycles.

It is important to say that the Depth algorithm will always use the number of available PEs, even if this incurs larger communication overheads. Obviously, if there are more PEs than instructions, Depth will map each instruction to a PE, leaving some PEs idle. On the other hand, RS and SCC-Map will seek for better mappings, possibly with only a few PEs, as long as a better overall performance is achieved (better execution time, lower communication overheads and reasonable amount of parallelism).

Results show that, for most of the scenarios, SCC-Map outperforms RS and Depth, even in the absence of cycles. This is a good indication that our approach to treat cycles is not only effective, but also generic enough to present good performance in not so favorable scenarios. Moreover, we observed that RS and Depth are less effective than SCC-Map when inter-PE communication latency is higher.

VI. Conclusions and Future Work

We presented SCC-Map, a novel task scheduling algorithm for dependency graphs with cycles, such as dataflow programs. In order to validate our algorithm and verify its performance we built a cycle-accurate dataflow simulator, based on the TALM model [1], [2]. We designed a set of benchmarks and compiled them to run on the simulator using 3 different mapping algorithms: RS [6], Depth [7] and SCC-Map. For each benchmark we have also varied communication latency (5 and 15 cycles) and the number of available processing elements in the system (2, 4, 8, 16, 32, 64 and 128). We compared the speedups in all scenarios, taking as baseline the serial execution, where all instructions are mapped to the same PE. Moreover, for each benchmark, we provided a theoretical maximum speedup, which is achieved by placing each instruction of the program in a distinct PE and setting the communication latency to 1 clock cycle (minimum possible latency).

Our results show that, for the majority of the evaluated scenarios, our set of benchmarks presents greater performance when executed the scheduling provided by SCC-Map, even when the benchmark has no cycles. This is a good indication that our approach to treat cycles is not only effective, but also generic enough accelerate loop-free applications. We have also observed that SCC-Map is even more effective when the system has a higher communication latency.

In this first study, for simplicity we have considered that all instructions take the same time to execute. However, SCC-Map and our simulator support instructions with different execution times. Experiments that consider such non-uniform scenarios are part of our future work. Moreover, since SCC-Map is a generic placement algorithm for dependency task graphs with cycles, it could be implemented and evaluated in a series of systems, besides the Trebuchet, such as heterogeneous architectures and clusters.

Acknowledgments

To CAPES and CNPq for the financial support given to the authors of this work.

References

[1] L. A. J. Marzulo, "Explorando Linhas de Execução Paralelas com Programação Orientada por Fluxo de Dados," Ph.D. dissertation, Universidade Federal do Rio de Janeiro, Oct. 2011.

[2] 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.

[3] 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 '11. New York, NY, USA: ACM, 2011, pp. 59–70.

[4] P. Zipf, G. Sassatelli, N. Utlu, N. Saint-Jean, P. Benoit, and M. Glesner, "A Decentralised Task Mapping Approach for Homogeneous Multiprocessor Network-On-Chips," *International Journal of Reconfigurable Computing*, vol. 2009, pp. 1–14, 2009.

[5] G. C. Sih and E. A. Lee, "A Compile-Time Scheduling Heuristic Heterogeneous Processor Architectures," vol. 4, no. 2, pp. 175–187, 1993.

[6] 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.

[7] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers, "Modeling instruction placement on a spatial architecture," in *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*. New York, NY, USA: ACM, 2006, pp. 158–169.

[8] H. Topcuoglu, S. Hariri, and I. C. Society, "Performance-Effective and Low-Complexity," *Computer*, vol. 13, no. 3, pp. 260–274, 2002.

[9] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The WaveScalar architecture," *ACM Transactions on Computer Systems*, vol. 25, no. 2, pp. 4–es, May 2007.