Procedia Computer Science

Volume 80, 2016, Pages 2287–2291

ICCS 2016. The International Conference on Computational Science

A Partition Scheduler Model for Dynamic Dataflow Programs

M. Michalska, E. Bezati, S. Casale-Brunet, and M. Mattavelli

EPFL STI-SCI-MM, École Polytechnique Fédérale de Lausanne, Switzerland

Abstract

The definition of an efficient scheduling policy is an important, difficult and open design problem for the implementation of applications based on dynamic dataflow programs for which optimal closed-form solutions do not exist. This paper describes an approach based on the study of the execution of a dynamic dataflow program on a target architecture with different scheduling policies. The method is based on a representation of the execution of a dataflow program with the associated dependencies, and on the cost of using scheduling policy, expressed as a number of conditions that need to be verified to have a successful execution within each partition. The relation between the potential gain of the overall execution satisfying intrinsic data dependencies and the runtime cost of finding an admissible schedule is a key issue to find close-to-optimal solutions for the scheduling problem of dynamic dataflow applications.

Keywords: scheduler, design space exploration, dynamic dataflow, multi-core

1 Introduction

In the last years, there has been a renewed interest in the field of dataflow programming. This fact has been originated by the limitation of the frequency increases of deep submicron CMOS silicon technology which has directed the evolution of processing platforms into concurrent systems composed of homogeneous or heterogeneous arrays of processors. The emergence of these new many/multi-core architectures poses some new problems and challenges for developing efficient application designs on them. Dataflow programming, in its various different models of computation *(MoCs)*, possesses nice properties that can be used to efficiently solve the new system implementation challenges consisting, essentially, in the portability of applications by themselves and in the portability of parallelism of application design [3, 6, 9, 10]. A dataflow program is defined as a directed graph in which each node represents a computational kernel, called *actor*, and each edge a first-in first-out *(FIFO)* lossless queue, called *buffer*. The communication between actors is permitted only by the exchange of atomic data packets (called *tokens*), by means of the interconnected buffers. Starting from this high-level and platform-independent representation, a software and/or hardware specific implementation of the design is then generated. However, the effectiveness of this process can greatly benefit from an accurate evaluation and exploration of different design alternatives (or *design points*) aiming at

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.415

satisfying general and/or design specific objectives and constraints (e.g. data throughput, energy consumption or any other implementation related optimizations). This process is known in literature as *design space exploration (DSE)*. The objective of *DSE* in the case of many-core and multi-core platforms is to compute: (a) how the application is partitioned (i.e., the mapping of actors to the available processing units), (b) which scheduling policy is assigned to each partition (i.e., what is the execution order of actors contained in the same partition), (c) how the buffers are dimensioned (i.e., what is the maximum number of tokens that can be stored). The problem is that the number of design points that need to be explored can rapidly explode in size even for very simple design instances. In fact, it can be shown that the three configurations mentioned above constitute NP-complete problems even taking each of them separately [3, 8].

In literature, a wide variety of *DSE* methodologies have been studied for the restricted classes of static *(SDF)* and cyclo-static *(CSDF)* dataflow programs [15]. For such classes of *MoCs* it is possible to analyze and optimize the design space at compile-time. However, both these classes have the disadvantage that the *MoC* expressiveness is very limited and most of interesting and useful applications (e.g., video codecs, packet switching networks algorithms, ...) cannot be efficiently and completely implemented by them. In these cases, the more expressive, but more hardly analyzable, dynamic dataflow programs *(DDF)* [2] are generally adopted. Different approaches need to be used for exploring the design space of *DDF* programs. The approaches based on the analysis of the execution trace [4, 6] which rely on a graph-structured data representations of *DDF* program executions have demonstrated their effectiveness for providing high quality solutions to partitioning and buffer dimensioning problems [5, 12]. The work developed in this paper aiming at searching for efficient scheduling policies, a problem which remains open for *DDF* designs, is also based on the same execution representation. The main idea is to define a systematic approach to model different scheduling policies by computing specific metrics that are used by *DSE* optimization algorithms.

2 Modeling

This section briefly describes the models used for the analysis of *DDF* programs. They include a model of a platform-independent program execution, a model of the target architecture, a performance estimation module and its underlying partition scheduler model.

2.1 Program execution

The model relies on the abstract representation of a program execution expressed as a directed, acyclic *Execution Trace Graph - ETG*. The *ETG* consists of a collection of atomic executions *(firings)* and the dependencies of various types imposing the precedence order between the *firings*. It is generated by a set of input stimuli which is sufficiently large and statistically meaningful in order to capture the whole dynamic behaviour of an application program. A detailed description of the properties of an *ETG* is given in Chapter 5 of [4].

2.2 Performance estimation

The simulation of a program execution is performed by a estimation module which is built upon a discrete event system specification formalism *(DEVS)* [14]. The *ETG*, whose firings are considered as *events* in the system, is weighted according to the target architecture model and profiling weights obtained by performing the measurements of a program on the target platform. The components of the module include the models of actors, buffers and their partitions, which

correspond to a physical assignment of the actors and buffers to the elements of the platform (i.e., different processing units). The focus of this work is concentrated on the model of a processing unit (actor partition) and its scheduler.

2.3 Partition scheduler

Given a partitioning configuration (i.e., a static assignment of each actor to a processing unit), each partition model contains a list of its members (actors), whereas each member contains its own list of firings, extracted directly from the *ETG*. The relative order of execution for the firings of each actor is determined by the order of appearance in the *ETG*. Furthermore, an eligibility of a firing to be executed is subject to the satisfaction of firing conditions: (1) input: if all necessary tokens are already available, and (2) output: if there is space available in all outgoing buffers. Since, in general, only one actor can be executed at a time, if multiple actors in a partition have *executable* firings (with both, input and output conditions satisfied), the partition must make a choice of an actor to execute. This choice is made by the *scheduler* according to its internal *scheduling policy*. The scheduling policy defines a set of rules basing on which the actors are (dynamically) chosen. The supported policies include: *Non-Preemptive (NP)*, *Round Robin (RR)*, and four different policies developed by the authors with the attempt of minimizing the execution time, described and discussed in detail in [13]. Depending on the policy, the scheduler can either search for the first executable actor on the list or iterate over the list in order to find the most convenient one (i.e., according to the given priorities). An important logging function inside the scheduler model keeps track on the number of conditions that are checked inside each partition between the two consecutive successful firings. It monitors also the cases where one or more conditions are not satisfied *(failed)*.

3 Experimental results

The experimental results reported in this section have been obtained in two phases. In the first one, the execution times simulated by the performance estimation module have been compared with the real values of execution times obtained on the platform. The level of accuracy of the results for different scheduling policies lead to an observation of a scheduling cost not considered in the performance estimation module. Then, in the second phase, the partition scheduler model described earlier has been employed to provide an estimation of such cost.

The experiments were performed with an MPEG-4 SP video decoder [7] consisting of 17 actors. The used input stimulus lead to an *ETG* consisting of about 1 million of firings and over 9 millions of dependencies. The target architecture was an array of (1 to 5) *Transport Triggered Architecture* processors [1]. The processing platform presents some interesting properties, such as cacheless construction which yields a deterministic profiling of the execution times. For the high accuracy of the performance estimation stage, already confirmed in previous experiments and reported in [11], the effect of the scheduling stage on the platform execution could be properly analysed and evaluated.

The experiments addressed the two, mentioned earlier, scheduling policies: *NP* and *RR* applied to a set of partitioning configurations generated with the attempt of balancing the workload between the partitions. The real execution times measured on a *TTA* platform using a cycle-accurate simulator [16] were compared with the simulated values generated by the performance estimation module *(PE)*. Tables 1 and 2 present the normalized numbers of clock cycles obtained in both cases. A positive values of difference (expressed in %) indicates that the real value is larger than the estimated value. For each scheduling policy, the numbers of

checked and failed conditions (as described in Section 2.3) were counted and the normalized values are presented in Tables 3 and 4.

Table 1: Execution times *NP*

Proc.	TTA [clk]	PE [clk]	diff [%]
1	19.04	19.01	0.15
2	10.27	10.67	-3.81
3	8.15	8.36	-2.52
4	6.40	7.00	-9.37
5	5.12	6.05	-18.22

Table 2: Execution times *RR*

Proc.	TTA [clk]	PE [clk]	diff [%]
1	21.31	19.01	10.81
2	10.64	9.57	10.10
3	8.18	6.69	18.23
4	6.39	5.33	16.65
5	4.42	4.53	-2.45

Table 3: Conditions checked

Proc.	NP	RR
1	33.051	143.934
2	54.445	180.484
3	76.641	192.344
4	112.795	222.847
5	119.455	233.804

Table 4: Conditions failed

Proc.	NP	RR
1	0.017	47.968
2	1.052	48.096
3	0.317	51.297
4	2.763	55.267
5	1.842	54.590

4 Discussion

It can be observed that for the *RR* policy the simulation discrepancy is much higher than for the *NP* (11.65% vs 6.82% considering the average absolute value). Furthermore, for the estimated values the *RR* is *always* better than the *NP*. It is, however, not reflected in the real values, where for 1 - 3 processors it is the *NP* that provides with shorter execution times. As a result, using the *RR* instead of the *NP* is only beneficial for the case of 4 and 5 processors. The gain is, however, much smaller than expected according to the simulation. For instance, for the case of 5 processors, the *RR* is 25% faster than the *NP*, while according to the platform execution it is only 14%. Hence, since the partition scheduler does not model the scheduler cost, it can be concluded that for the *NP* this cost is rather negligible, while for the *RR* it cannot be considered negligible.

It corresponds well with the statistics provided on the numbers of checked and failed conditions, because in all cases the *RR* policy is characterized with bigger values than the *NP*. In some cases the difference is quite large, for instance, for the mono-core configuration the fraction of failed conditions for *RR* is almost 3000 larger than for the *NP*. Averaging the values, it can be concluded that one successful firing with the *RR* requires checking 61% more conditions than *NP*.

These observations lead to interesting considerations of extensions and improvements. The opportunities of measuring and modeling the scheduling cost should be investigated. It can be performed only as a matter of approximation, since the real cost might be subject to multiple factors, such as the level of dynamism inside the actors in a certain partition, their complexity (i.e., the numbers of input/output conditions) or even their order of appearance. Nevertheless, the scheduling cost could be modeled as a function of checked/failed conditions, where each check/failure is assigned with a certain value. Furthermore, since the results confirm that an appropriate choice of the scheduling policy can provide not negligible performance improvements, further studies on the development of more sophisticated scheduling policies seem promising. Minimizing the numbers of conditions checked/failed, as provided by the simulation,

could be taken as an indicative optimization criterion. Finally, an extensive study of a dynamic dataflow execution should be performed in order to establish if it is possible to reduce the cost of condition checking at every execution by identifying static regions in the *ETG*.

Acknowledgment

This work is supported by the Fonds National Suisse pour la Recherche Scientifique under grant 200021.138214.

References

[1] TTA-Based Co-design Environment. http://http://tce.cs.tut.fi/tta.html, Last checked: April 2016.

[2] S. Bhattacharyya, E. Deprettere, and B. Theelen. Dynamic dataflow graphs. In *Handbook of Signal Processing Systems*, pages 905–944. Springer, 2013.

[3] S. Bhattacharyya, P. Murthy, and E. Lee. *Software synthesis from dataflow graphs*, volume 360. Springer Science & Business Media, 2012.

[4] S. Casale-Brunet. Analysis and optimization of dynamic dataflow programs. *PhD Thesis at EPFL, Switzerland*, 2015.

[5] S. Casale-Brunet, M. Mattavelli, and J. W. Janneck. Buffer optimization based on critical path analysis of a dataflow program design. In *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pages 1384–1387. IEEE, 2013.

[6] J. Castrillon, R. Velasquez, A. Stulova, W. Sheng, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr. Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms. *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2010.

[7] ISO. Information technology - MPEG systems technologies - Part 4: Codec configuration representation. ISO 23001-4:2011, International Organization for Standardization, 2011.

[8] E. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, C-36(1):24–35, January 1987.

[9] E. Lee and T. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, 1995.

[10] M. Mattavelli. MPEG reconfigurable video representation. In Leonardo Chiariglione, editor, *The MPEG Representation of Digital Media*, pages 231–247. Springer New York, 2012.

[11] M. Michalska, J. Boutellier, and M. Mattavelli. A methodology for profiling and partitioning stream programs on many-core architectures. In *International Conference on Computational Science (ICCS), Reykjavik, Iceland, June 1-3, Procedia Computer Science Ed.*, volume 51, pages 2962–2966, 2015.

[12] M. Michalska, S. Casale-Brunet, E. Bezati, and M. Mattavelli. Execution trace graph based multi-criteria partitioning of stream programs. In *Procedia Computer Science Ed.*, volume 51, pages 1443–1452, 2015.

[13] M. Michalska, N. Zufferey, J. Boutellier, E. Bezati, and M. Mattavelli. Efficient scheduling policies for dynamic dataflow programs executed on multi-core. *9th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG)*, 2016.

[14] J. Nutaro. *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Wiley Publishing, 2010.

[15] T. Parks, J. Pino, and E. Lee. A comparison of synchronous and cycle-static dataflow. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 1, pages 204–210. IEEE, 1995.

[16] H. Yviquel, A. Sanchez, P. Jaaskelainen, J. Takala, M. Raulet, and E. Casseau. Embedded multi-core systems dedicated to dynamic dataflow programs. *Journal of Signal Processing Systems*, pages 1–16, 2014.