UDC 004.4

SPACE-TIME PARALLELISM EXPLORATION ON MULTI-CORE SYSTEMS

Prihozhy A. A.
Belarusian National Technical University, Minsk, Belarus,
prihozhy@yahoo.com

The modern multi-processor system architectures realize three types of parallelism depending on the structure of input data [1]: space parallelism on one data set, time parallelism on data flow (sequence of data sets) and mixed space-time parallelism. The high-performance parallel computing is impossible without the time parallelism that is implemented by means of pipelining [2–11]. This paper aims at the analysis of the three types of parallelism in task graphs to find efficient implementations of the system on a multi-core processor.

In computing, a pipeline is a set of data processing elements (stages) connected in series, so that the output of one stage is the input of the next stage [9]. The stages of a pipeline operate in a time-sliced fashion. To do this, pipeline buffers are inserted in between the stages. The pipeline stage time has to be larger than the longest time delay between two neighbor stages. A pipelined system requires more resources than the system that executes one batch at a time, because any stage cannot reuse the computational resources of previous stages. Pipelining is a natural technique of the development of streaming applications, which organize data as a sequence of data sets over all parts of the design.

Figure 1 shows that the system specification to be implemented as a pipeline includes an input data flow, a high-level behavioral description to be pipelined, and an output data flow. The following languages and intermediate representations have been developed and used for describing pipeline specifications and modelling the pipeline at all steps of synthesis and optimization [3–6, 8]: the programming language C, data flow graphs (DFGs), signal flow graphs (SFGs), transactional specifications, dataflow description languages and other notations. The actor-based algorithmic language, CAL has been developed for representing pipelined networks [12–14].

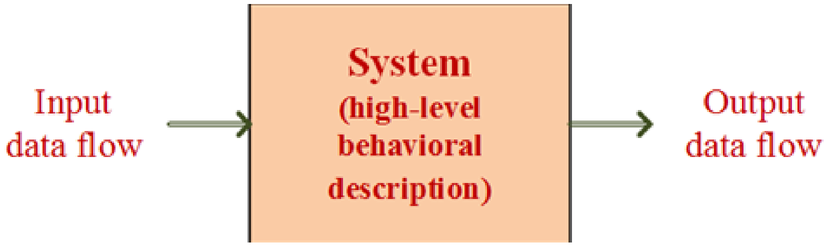

Figure 1 – System specification for dataflow implementation on multi-core processor

A pipelined system is characterized by several parameters such as the number of pipeline stages, the pipeline buffers size, the stage cycle time, the latency, the data initiation interval, the throughput, etc. The optimization can improve the pipeline parameters significantly.

Pipelining is a certain type of transformation of a digital system behavioral specification into a set of partitions that represent pipeline stages and execute in time-sliced fashion on the input data flow. Pipelining increases the operating frequency and throughput of data-intensive digital systems with long critical paths.

Computational pipelines are classified into hardware and software pipelined systems. The parallelism in a hardware system exists at the level of logical micro and macro elements, while the parallelism in a software system exists at the level of algorithms, threads and tasks. There are differences between the synthesis techniques that target parallel hardware systems and parallel software systems.

The pipeline synthesis problem aims either at minimizing the throughput given a constraint on the implementation cost, or at minimizing the implementation cost given a constraint on the throughput.

Since a multi-processor system exploits three types of parallelism, one of the key problems is to find efficient balancing between the space and time parallelism. We recognize three cases of implementing a data flow system on a multi-core processor:

1. All cores realize the space parallelism.
2. All cores realize the time parallelism.
3. A part of cores realizes the space parallelism and other cores realize the time parallelism.

In the paper, we assume that all cores in a multi-core system have the same parameters. The hierarchical memory of the system includes one or more local caches for each core, a cache that is shared among all cores, and a main memory. The data access time is largest for the main memory, and it is smallest for the local caches. The access time for the shared cache is smaller than that of the main memory and is larger than that of the local cache. The paper considers asynchronous system implementations [15]. This is because the execution time of a task is variable on a multi-core processor, which runs a multi-task operating system. The data transfer time among cores is also variable in a multi-core system.

In the paper, we represent a high-level behavioral description with the task graph model [1]. In a task graph, a vertex is a task and its weight is the task execution time on a core. An edge represents a data transfer between tasks, and its weight is the data transfer time between cores through the main memory. Two tasks assigned to one core have a reduced data transfer time since the data are transferred through the local cache. If the tasks are assigned to different cores, and the amount of data transferred from one core to another through the shared cache is not high, the data transfer time is reduced against the transferring through the main memory.

Time-parallel (pipelined) implementation. It is preferable for a system whose behavioral description has a long critical path. Figure 2 depicts the pipeline components and their assignment to the processor cores and hierarchical memory components. The cores implement the pipeline stages, which operate using local caches. The data transfer between stages is carried out through the local and shared caches, and through the main memory in case of big data.

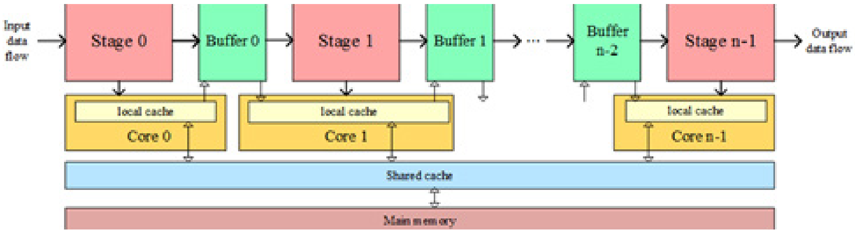

Figure 2 – Pipeline implementation on a multi-core system

Let a weighted task graph $G = (T, E)$ represents the system behavioral specification. Let $T = \{t_1,\ldots,t_n\}$ be a set of graph task-vertices and $w(t_i)$ be a weight of vertex t_i (execution time of the corresponding task). Let L be a sum of tasks execution time, $w(t_i)$ over all tasks $i = 1\ldots n$. Let the multi-core processor has m cores. Then, each core must carry out the load of $l \geq L / m$ during the stage time period, while processing one data set.

Our goal is to search for *m* sub-graphs of the task graph, each assigned to a separate core. The computational load of each core must be about the same. To reduce the cost of data transfer between cores, the sum of edge weights must be minimal over all edges between task-vertices assigned to different cores. In this case, the overall size of pipeline buffers is reduced.

Figure 3 depicts an example task graph consisting of n = 10 task-vertices. The overall computational load of the tasks is L = 389 ms. Let the processor has two cores (m = 2). Then, the core load per data set should be l ≥ 194.5. Figure 3a and Figure 3b depict two decompositions of the task graph and two architectures of pipeline for the 2-core processor. To estimate the architecture parameters, we assume that the data transfer time between pipeline stages is 4 times lower through the local cache over the transfer through the main memory:

Architecture a)

Stage 0 includes tasks 0, 1, 2, 3 and 4. The overall execution time of tasks is 156 ms.

The data transfer time between the tasks through the local cache is 22 / 4 = 5.5 ms.

Stage 1 includes tasks 5, 6, 7, 8 and 9. The overall execution time of tasks is 233 ms.

The data transfer time between the tasks through the local cache is 18 / 4 = 4.5 ms.

The pipeline stage time is *max* (156 + 5.5, 233 + 4.5) = 237.5 ms.

The data transfer time between the stages is 40 ms.

Architecture b)

Stage 0 includes tasks 0, 1, 3, 4 and 9. The overall execution time of tasks is 182 ms.

The data transfer time between the tasks through the local cache is 29 / 4 = 7.25 ms.

Stage 1 includes tasks 2, 5, 6, 7 and 8. The overall execution time of tasks is 207 ms.

The data transfer time between the tasks through the local cache is 32 / 4 = 8 ms.

The pipeline stage time is *max* (182 + 7.25, 207 + 8) = 215 ms.

The data transfer time between the stages is 19 ms.

Observing the parameters of two pipeline architectures, we conclude that architecture 1 is better against architecture 0 regarding both the stage time and data transfer time.

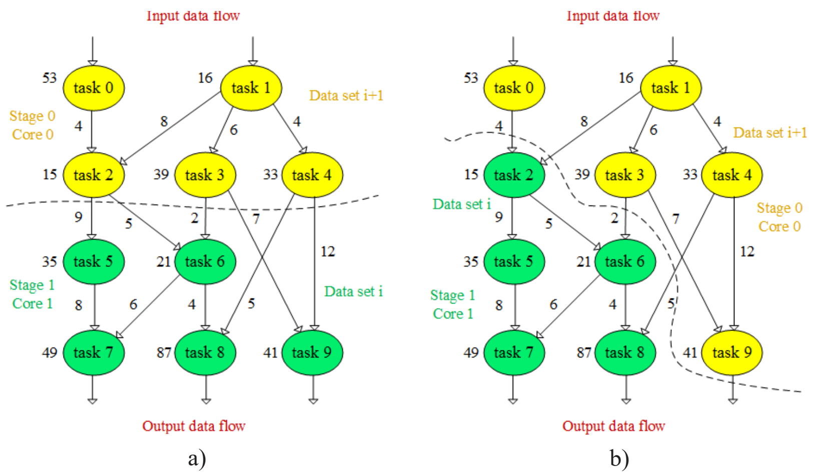

Figure 3 – Task graph pipelining:
a) architecture 1 of two-stage pipeline with stage time period of 237.5 ms;
b) architecture 2 of two-stage pipeline with stage time period of 215 ms

Space-parallel implementation. An alternative to the time-parallel multi-core implementation is a space-parallel multi-core implementation. The selection of the implementation architecture depends on properties of the task graph. If the amount of the space parallelism is sufficient for the given number of cores, the space-parallel architecture is preferable since it does not require any pipeline buffers. Figure 4a depicts a task graph, which has large enough amount of space parallelism for two cores. Allocating tasks 0, 1 and 2 to core 0, and tasks 3, 4 and 5 to core 1 (figure 4b) balances the cores' load. The cores process a data set *i* with the time delay of *max* (129 + 8 / 4, 112 + (7 + 5) / 4) = *max* (131, 115) = 131 ms, which is composed of the tasks execution time and data transfer time through the local cache. No need in pipelining in this case.

Mixed time-space-parallel asynchronous implementation. A system implementation is time-space-parallel, if the tasks are assigned to cores in such a way that some pairs of cores operate in the time-parallel mode, some of them operate in the space-parallel mode, and other pairs operate in the mixed time-space-parallel mode. Figure 5a depicts a task graph whose set of tasks is divided into five subsets, each implemented on a separate core. The task set decomposition aims at balancing the load of cores.

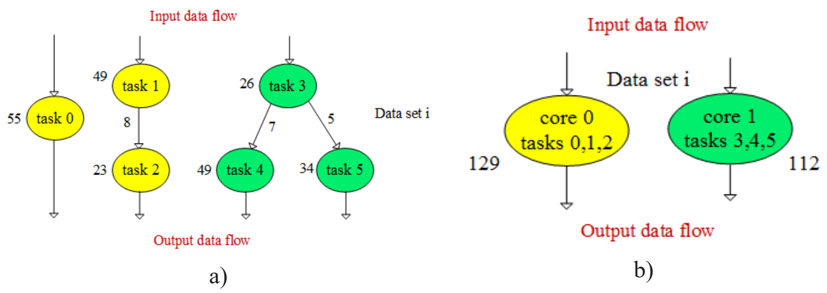

Figure 4 – Space-parallel system: a) task graph partitioning; b) space-parallel implementation on two cores

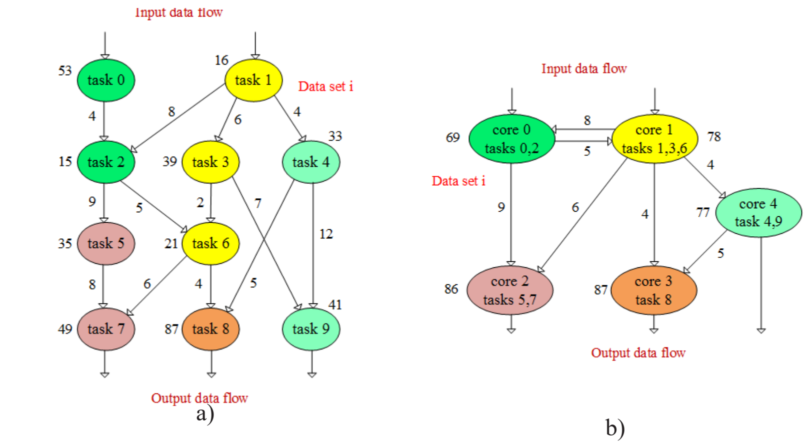

Figure 5 – Mixed time-space-parallel asynchronous system implementation: a) task graph partitioning for five-cores; b) space-time-parallel implementation on five cores

Figure 5b shows a five-core time-space-parallel asynchronous threaded implementation architecture. It consumes a sequence of data sets at input and produces a response sequence of data sets at output. The load of cores on one data set varies from 69 ms to 87 ms. Pair 0 – 2 of cores as well as pair 1 – 3 operate in time-parallel (pipelined) mode. There are data dependences between the cores 0 and 2, and between 1 and 3, which require pipeline buffers inserted in between the cores. The cores communicate and transfer data asynchronously. While core 2 processes the data set *i*, core 0 processes the data set *i* + 1. The same concerns cores 1 and 3.

Cores 2 and 3 as well as cores 2 and 4 operate in space-parallel mode on one or two neighbor data sets, as there are no data dependencies between cores 2 and 3 as well as between 2 and 4. Core pairs 0 – 1 and 1 – 4 operate partially sequentially, partially in space-parallel mode and partially in time-parallel mode. For instance, core 0 realizes tasks 0 and 2, and core 1 realizes tasks 1, 3 and 6, which can communicate in different modes. In particular, task 0 of core 0 is executed in parallel with tasks 1 and 3 of core 1. Task 2 is executed in series with tasks 1 and 6. Task 2 is executed in parallel with task 3. Task 0 can be executed in the time-parallel mode with task 6 because the tasks may process different data sets. Cores 3 and 4 operate partially sequentially and partially in space-parallel mode.

It is interesting to note that while cores 2, 3 and 4 process the *i* data set, cores 0 and 1 may begin processing the $i + 1$ data set. The throughput of the mixed-parallel five-core architecture is more than twice higher over the purely pipelined two-core architectures presented in Figure 3.

The task execution times and the data transfer times significantly depend on the input data flow. Changes in input data can infer changes in the execution time of tasks and in amount of data transferred from one task to another. The weights of vertices and edges of the task graph are modified, although the graph structure is the same. In its turn, the optimal allocation of tasks to cores and the parameters of the time-space-parallel architecture depend on the vertex and edge weights of the task graph. In this case, it is reasonable to apply the reconfiguration methodology [14, 16–17] and develop techniques for synthesis of reconfigurable space-time-parallel implementations, which can tune to the input data flow.

Conclusion

The parallelism in a hardware system exists at the level of logical micro and macro elements, while the parallelism in a software system exists at the level of algorithms, threads and tasks. This difference infers different methods of the parallel system synthesis and optimization. The paper has given an analysis of three types of parallelism in a software system having a data flow at input: space, time and mixed parallelism. The goal is to generate efficient implementation on multi-core processor from a task-graph model. The preferable usage of a particular parallelism type depends on the task graph configuration and the core count. If the graph has long critical paths and the number of cores is limited, the system implementation should be time-parallel or pipelined. If the graph has many independent branches and the number of cores is limited, the system implementation should be space-parallel. In other cases, the system implementation is more efficient if it is decomposed into subsystems, which operate pair-wisely in mixed space-time-parallel mode. The throughput of such implementations increases over the purely time-parallel and purely space-parallel implementations.

REFERENCES

1. Прихожий, А. А. Распределенная и параллельная обработка данных. – Минск: БНТУ, 2016. – 90 с.

2. M. Weinhardt and W. Luk, "Pipeline vectorization," Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 20, no. 2, pp. 234–248, Feb. 2001.

3. D. I. Ko and S. S. Bhattacharyya, "The pipeline decomposition tree: an analysis tool for multiprocessor implementation of image processing applications," in Proc. CODES+ISSS '06: 4th Int. Conf. on Hardware/software codesign and system synthesis, 2006, pp. 52–57.

4. S. Oh, T. G. Kim, J. Cho, and E. Bozorgzadeh, "Speculative loop pipelining in binary translation for hardware acceleration," Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 27, no. 3, pp. 409–422, March 2008.

5. H. Javaid, A. Ignjatovic, and S. Parameswaran, "Rapid design space exploration of application specific heterogeneous pipelined multiprocessor systems," Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 29, no. 11, pp. 1777–1789, November 2010.

6. E. Nurvitadhi, J. Hoe, T. Kam, and S. Lu, "Automatic pipelining from transactional datapath specifications," Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 30, no. 3, pp. 441–454, March 2011.

7. Z. Zhang, B. Liu. "SDC-Based Modulo Scheduling for Pipeline Synthesis," IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 211–218, November 2013.

8. A. Prihozhy, E. Bezati, A.-H. Ab Rahman, M. Mattavelli. "Synthesis and Optimization of Pipelines for HW Implementations of Dataflow Programs," IEEE Transactions on CAD, vol. 34, no. 10, pp. 1613–1626, 2015.

9. Прихожий, А. Эвристический генетический алгоритм оптимизации вычислительных конвейеров / А. А. Прихожий, А. М. Ждановский, О. Н. Карасик, М. Маттавелли // Доклады БГУИР, 2017, № 1, с. 34–41.

10. A. Prihozhy, S. Casale-Brunet, E. Bezati and M. Mattavelli, "Efficient Dynamic Optimisation Heuristics for Dataflow Pipelines," 2018 IEEE International Workshop on Signal Processing Systems (SiPS), 2018, pp. 1–6, doi: 10.1109/SiPS.2018.8598386.

11. Prihozhy, A., Casale-Brunet, S., Bezati, E., M. Mattavelli. Pipeline Synthesis and Optimization from Branched Feedback Dataflow Pro-

grams. J Sign Process Syst 92, 1091–1099 (2020). https://doi.org/10.1007/s11265-020-01568-5.

12. J. Eker and J. Janneck, CAL Language Report: Specification of the CAL Actor Language. University of California-Berkeley, December 2003.

13. M. Canale, S. Casale-Brunet, E. Bezati, M. Mattavelli, J. Janneck: “Dataflow Programs Analysis and Optimization Using Model Predictive Control Techniques”, Journal of Signal Processing Systems, 2016, Vol: 84, No. 3, Pages 371–381.

14. M. Mattavelli, I. Amer, M. Raulet, “The Reconfigurable Video Coding Standard“ [Standards in a Nutshell], Signal Processing Magazine, IEEE 27 (3) (2010) 159–167.

15. Prihozhy, A. A. Asynchronous scheduling and allocation / A. A. Prihozhy / Proceedings Design, Automation and Test in Europe. Paris, France. – IEEE, 1998, pp. 963–964.

16. Z. Gong, K. Qiu, W. Chen, Y. Ni, Y. Xu, J. Yang, Redesigning pipeline when architecting STT-RAM as registers in rad-hard environment, Sustainable Computing: Informatics and Systems, Volume 22, 2019, Pages 206–218, https://doi.org/10.1016/j.suscom.2018.09.001.

17. A. I. Dordopulo, I. I. Levin. Performance Reduction For Automatic Development of Parallel Applications For Reconfigurable Computer Systems. Supercomputing Frontiers and Innovations, Volume 7, No. 2, 2020, Pages 4–23, https://DOI:10.14529/js200201.