ISSN 1995-0802, Lobachevskii Journal of Mathematics, 2020, Vol. 41, No. 12, pp. 2614–2620. © Pleiades Publishing, Ltd., 2020.

Features of Dataflow Processor Emulator Implementing

B. M. Shabanov[1*], E. A. Kuznetsova[1], and A. A. Rybakov[1***]**

(Submitted by A. M. Elizarov)

[1] *Joint Supercomputer Center, Scientific Research Institute for System Analysis of the Russian Academy of Sciences, Moscow, 119334 Russia*

Received May 1, 2020; revised May 30, 2020; accepted June 5, 2020

Abstract—The development of dataflow processor architecture is a promising direction for improving the performance of computing systems. The dataflow processors have several advantages in contrast with the von Neumann architecture processors. These advantages are express in explicit parallelism of calculations, since in the presentation of programs for the dataflow processors the execution of instructions is limited solely by the fact of input data's ready state. The concept of using tokens as data storage for instructions allows you to remove many problems associated with data conflicts and simplifies the program logic. One of the most important elements in the implementation of the logic of the dataflow processor is the associative token memory, which should provide storage and quick search for ready-made data for instructions that can be transferred for execution. This article discusses the functionality of the dataflow processor emulator and its parts including associative token memory and the logic of its operation using the example of a simple dataflow graph.

DOI: 10.1134/S1995080220120379

Keywords and phrases: *dataflow processor emulator, dataflow graph, token, token state, associative token memory, control flow graph, def-use graph.*

1. INTRODUCTION

A processor with dataflow architecture (dataflow processor) has the potential to provide the highest performance among processors due to the fact that the parallelism of command execution is taken into account when drawing up a program graph. The program execution is not required to be performed by sequentional translating of the sequence of instructions, as is in the case of a processor with the von Neumann architecture. A program in the dataflow processor is a directed graph where nodes are instructions, and information transfers by the edges as tokens, which contains the data field and context field of transferring data. This context identifies where token should be sent to (the identifier of command and number of its argument) and also contains the state, which allows executing different iterations of nested loops and different executions of procedures at the same program graph at parallel. And regardless of the place of the instruction with many inputs in the graph, it is passed for execution when the last of the operands (tokens) with the same state arrived. After calculating the result of the instruction, new tokens with the result value are sent to the inputs of subsequent commands according to the program graph, and the used operand tokens are destroyed.

At the same time, in contrast to the von Neumann architecture, the dataflow processor has not a central control device (and instruction counter), and the parallelism of the executing instructions is determined in dynamics by the arrival of operands to the instruction inputs in the decentralized scheme. Namely, the device for searching ready-to-execute commands—the memory of searching arguments sets (the token assiciative memory) can be executed such as many parallel-running modules, which

*E-mail: shabanov@jscc.com
**E-mail: mrallis@jscc.com
***E-mail: rybakov.aax@gmail.com

provides the same large numbers of arithmetic, logic or vector execution units. Since the execution of an instruction is dependent upon the arrivals of operands, the management of token storage and instruction scheduling are intimately related in any dataflow processor [1, 2].

Unlike the implementation of a multiprocessor system, direct hardware implementation allows you to fully implement the calculation graph in the form of a computing pipeline, thereby eliminating the need to create a module that implements the execution of the calculation schedule [3].

The difficulties noted above for the hardware implementation of the dataflow processor led to the fact that none of the many projects for its creation was possible to achieve higher performance compared to the von Neumann processor, and by the end of the 2000s, there were practically no such projects. This refers to the development of universal dataflow processors, but not specialized ones, since, for example, dataflow processors of digital signal processing were not only developed, but also produced [4].

The Joint Supercomputer Center of the Russian Academy of Sciences (JSCC RAS) is developing a vector processor with the dataflow architecture [5]. This paper introduces the development of the emulator of the model of dataflow processors family.

2. EMULATION OF PROCESSOR WITH THE VON NEUMANN ARCHITECTURE

One of the basic principles of the von Neumann architecture is the principle of control flow. According to it, the executable code of the program is stored in memory, and the processor can execute its instructions sequentially, one by one. The next instruction address submitted to the processor for execution is stored in a special register called instruction pointer. By default, it always shifts to the next instruction in memory. This address can be forcedly changed by special transition instructions, and in this case, we can talk about transferring control to another part of the program.

Let's introduce the concept of a linear section (or program basic block). The linear section is a sequence of instructions with the next two properties. First, control could only be transfered to the first instruction of a given section. In other words, a linear section has only one input, which is the section's first instruction. Second, if control was transferred to the linear section, then all instructions of this section must be executed (i.e., there can be no transition instructions inside the linear section). The transition instruction can only be at the end of the linear section. Sometimes, instead of linear sections, it is convenient to view so-called quasilinear sections, which can have an arbitrary amount of transition instructions inside for transitions to other linear sections. We will not consider quasilinear sections, because any such section can be divided into a sequence of ordinary linear sections.

An important concept widely used in the theory of optimizing compilers is the control flow graph (CFG) [6]. A control flow graph is a directed graph whose nodes are linear sections of the program, and edges indicate the transfer of control between them. The control flow graph is a convenient and demonstrative structure for describing program behavior and used by optimizing compilers to implement global program optimizations, i.e. optimizations that operate with linear sections.

Consider the logic of intermediate program representation inside one linear section. Consider for this a simple example of finding the roots of a quadratic equation. We will use listing in the C programming language to write examples and pseudocode. Thus, we have a linear section (program code block) at the input of which we have the variables a, b, c containing the coefficients of the quadratic equation $ax^2 + bx + c = 0$. At the linear section output it is required to get the values of the roots, which are calculated by the common formulas $x_{1,2} = (-b \pm \sqrt{b^2 - 4ac})/(2a)$.

Listing 1. Code block for calculating the roots of quadratic equation

```
1 {
2     float t = sqrt(b * b - 4.0 * a * c);
3
4     x1 = (-b + t) / (2.0 * a);
5     x2 = (-b - t) / (2.0 * a);
6 }
```

In Listing 1, the linear section is highlighted by curly brackets and temporary variable t is declared to understand that this variable is not used outside the linear section. For convenience, we assume that the processor, for which the executable code will be created, supports the instruction for calculating

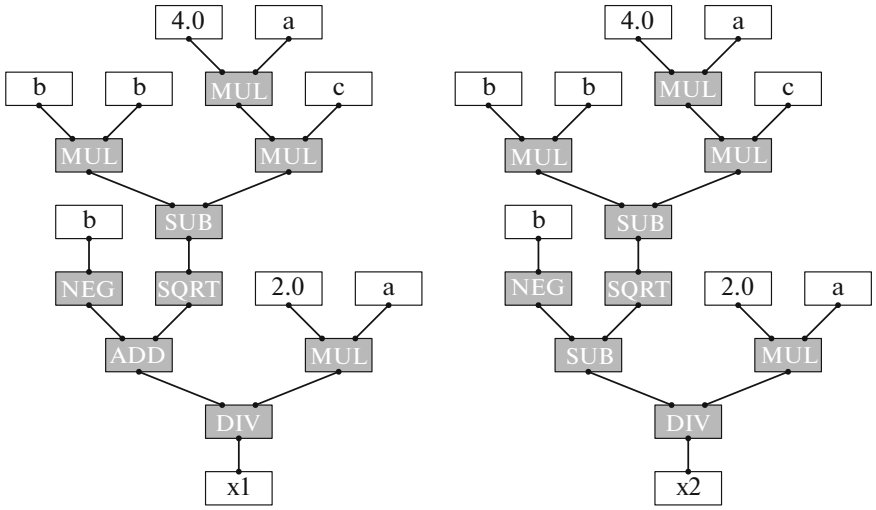

Fig. 1. Trees for calculating the roots of quadratic equation.

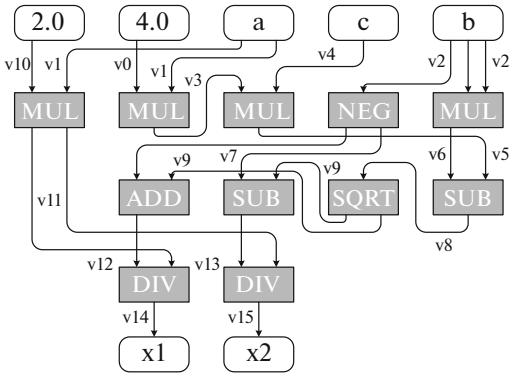

Fig. 2. Dependency graph of a linear section which calculates the roots of quadratic equation.

the square root. Thus, the following instructions for working with real numbers appear in the listing above: MUL—multiplication of two numbers, SUB—subtraction of two numbers, SQRT—the square root of a number, NEG—unary negative, ADD—addition of two numbers, DIV—division of two numbers.

Figure 1 shows two trees of evaluating the expressions x_1 and x_2. The compiler, of course, will simplify the calculation data by collecting common subexpressions, and the final version of the linear section representation will be close to the dependency graph shown in Fig. 2 (def-use graph, DFG).

Nodes of this directed graph are separate operations (shown as dark rectangles), and edges are def-use dependencies between data operations (the B command depends on the A command if uses the data generated by the A command). It can be other kinds of dependencies between operations besides data dependency (control dependency, anti-dependency), but these types of dependencies are missing in our simple example.

To form the resulting executable code the presence of a def-use graph is not enough. It is necessary to line up all commands included in the def-use graph in a single sequence, which will subsequently be written to a linear section of memory. In this case, when aligning a sequence of commands, it is necessary to observe the requirement that if the B command depends on the A command according to the data, then it should be later than the A command in the final sequence. This process called code planning [7]. In the result of code planning in the considered case, we will get a pseudocode for calculating the roots of quadratic equation similar to the code in Listing 2.

Listing 2. Pseudocode for calculating the roots of quadratic equation

```
1  MOV 4.0       ->  v0
2  MOV   a       ->  v1
3  MOV   b       ->  v2
4  MUL  v0,  v1  ->  v3
5  MOV   c       ->  v4
6  MUL  v3,  v4  ->  v5
7  MUL  v2,  v2  ->  v6
8  NEG  v2       ->  v7
9  SUB  v6,  v5  ->  v8
10 SQRT v8       ->  v9
11 MOV 2.0       -> v10
12 MUL v10,  v1  -> v11
13 ADD  v7,  v9  -> v12
14 SUB  v7,  v9  -> v13
15 DIV v12, v11  -> v14
16 DIV v13, v11  -> v15
17 MOV v14       ->  x1
18 MOV v15       ->  x2
```

In Listing 2, all instructions except transmission operation operate on virtual registers (v0-v15). The order of the commands is important in this form. At any time after the execution of any instruction, we can talk about some state of the processor which is characterized by the content of memory and of all the registers used.

The main part of the processor emulator is the realization of instruction semantic, executed by the processor. Every instruction is considered as a function, which transfers the computer from one state to another.

3. DATAFLOW PROCESSOR EMULATION

The dataflow processor emulator, which considered in this article, has a dynamic model. It permits to pass the commands for execution when their tokens are ready with the simultaneous execution possibility of inner cycle' iterations and different procedure launches on the same program graph.

The dynamic model of the dataflow processor allows revealing more parallelism in program due to not only the coincidence of command number, of the operand tokens, but also of the tokens state [8].

In contrast with processors with the von Neumann architecture, dataflow processors do not need complete orderliness in command execution. I.e. if two instructions can be executed in parallel (this is possible in the absence of dependence between them) then the order of their execution is not determined. We will consider the same example of the linear section, inside which the roots of the quadratic equation are calculated. The program for the dataflow processor is the graph just like the def-use graph showed in Fig. 2. Dataflow graph, which is a program for dataflow processor, is also directed. The nodes of this graph are also the instructions, but edges have a completely different meaning. In the case of def-use graph, edges were fixed fact of the dependence between two operations (indicating by which variables, by which resource), in case of dataflow graph the edge is a unique data carrier that names token. Therefore, the token is a data structure that stores information about transferring data and about the destination point of this data (the identifier consuming this instruction data as well as the argument number). Besides, the token contains additional information (token state) allowing to distinguish instruction, which relates to different function calls or to different loop iterations, but for this example, this is not important, therefore, the description of token state will be omitted.

Figure 3 shows the dataflow graph in the initial time when none of the instructions have been completed yet. In this figure, on the edges of the graph, you can see those tokens that are ready and transferred to the relevant instructions. For example, for the instruction 9 MUL two tokens are ready: $\{9, 0, 2.0\}$ (9—number of instruction, 0—number of argument, 2.0—real number), $\{9, 1, a\}$ (9—number of instruction, 1—number of argument, a—real number). Both arguments for the instruction 9 MUL are ready and instruction can be executed. After execution, tokens containing arguments are deleted,

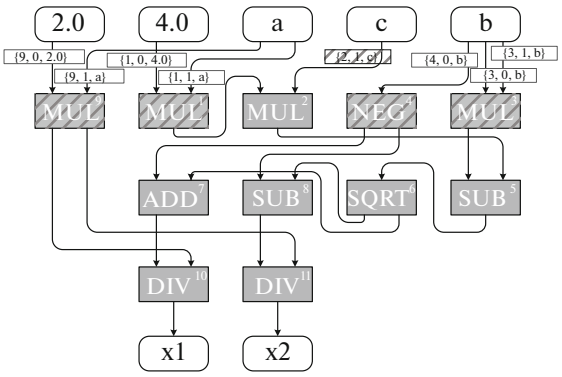

Fig. 3. Dataflow graph of a linear section which calculates the roots of a quadratic equation.

and tokens containing the result of instruction 9 MUL will be sent by the output edges to instructions 10 DIV and 11 DIV.

If we consider another instruction 2 MUL, we can see that only the second argument is ready for it (token $\{2, 1, c\}$). The first argument will only be generated after executing the 1 MUL instruction.

Thus, all instructions in the presented graph will be processed until the required values are obtained in tokens on the output edges from instructions 10 DIV and 11 DIV.

The implementation of instruction semantics in a dataflow processor emulator is no different from the same functionality in a traditional processor emulator.

The central functional element for ensuring the operability of the dataflow processor is so-called associative token memory, which is designed to store and search tokens for ready-to-execute commands.

Associative token memory is a repository of tokens that are currently created and are being executed in the program. Moreover, if it turns out that in the associative memory there are all the necessary arguments for a command, they are immediately removed from the associative memory and sent for execution to the command. Thus, at any moment in time in the associative memory there are only those arguments that wait for the missing arguments for some instructions. That is, at the conditionally first moment in time, shown in Fig. 3, in the associative memory of tokens there is only one token $\{2, 1, c\}$, waiting for its pair, while the commands are 9 MUL, 1 MUL, 4 NEG, 3 MUL are already ready for execution and are in the command buffer along with sets of their arguments.

The dataflow processors approach resolves any threads of control into separate instructions that are ready to execute as soon as all required operands available [9]. In Fig. 3 the token located in the associative memory is shaded, as well as the commands ready to be executed.

After execution of the instructions 9 MUL, 1 MUL, 4 NEG and 3 MUL, the only instruction ready for execution is 2 MUL. The state of the computer before executing this instruction is shown in Fig. 4. In this figure, the 2 MUL instruction ready for execution is visible, as well as several tokens that have been passed into associative memory. Further execution of the program is similar and at the end we come to the formation of two tokens on the output edges of operations 10 DIV and 11 DIV, which will then be sent to their consumer instructions.

In the dataflow processors, overflow of the associative memory is unacceptable, that is why the associative memory must have high capacity and be fast at the same time, which is difficult to realize in practice. That is the point to implement the emulator, which will help to get closer to creating such the dataflow processor while revealing all possible errors and shortcomings.

4. ARCHITECTURE OF DATAFLOW PROCESSOR EMULATOR

Consider the architecture of the dataflow processor emulator in general, presented in Fig. 5.

The vector dataflow processor emulator is implemented using four infinite processes that exchange messages with each other. Each process is a separate processor entity and has its own functionality.

The main process *main* is the lead in the emulator. He is the host of three basic data structures that reflect the execution of the program. The *tokens* structure contains a list of tokens that currently exist during program execution. The structure of *graph* is a directed graph that implements the logic of the

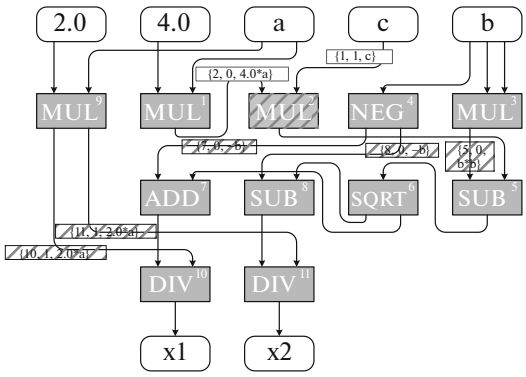

Fig. 4. Dataflow graph of a linear section which calculates the roots of quadratic equation before executing the 2 MUL instruction.

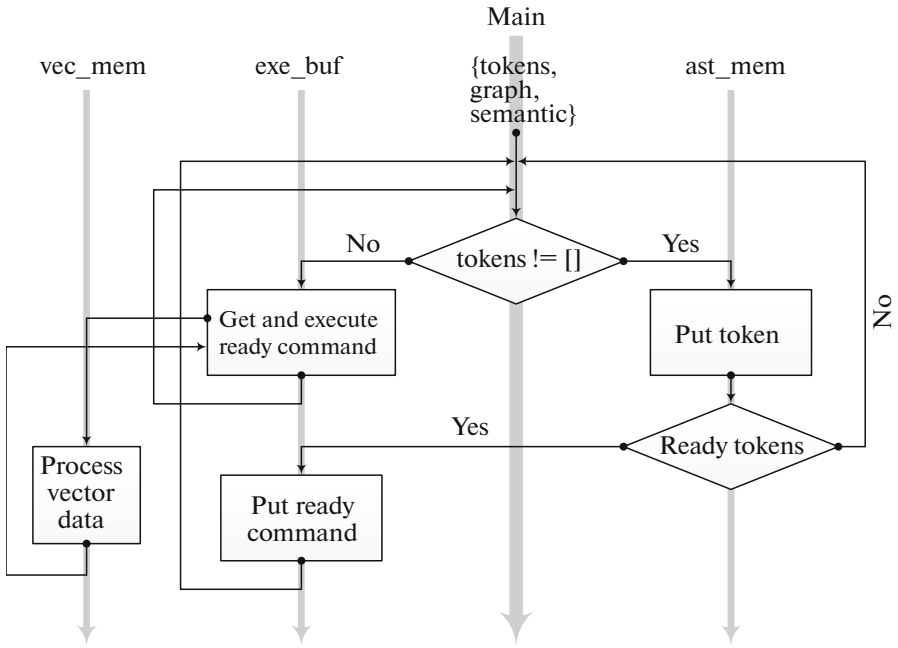

Fig. 5. General architecture of dataflow processor.

program. Finally, the *semantic* structure describes the semantics of processor instructions, that is, it contains the implementation of functions that, according to a set of input tokens, generate sets of output tokens for each instruction.

The next process under consideration is the *ast_mem* process, which implements the logic of associative token memory. The interface of this process consists of one single function—to put a token into associative memory. In the event that this token is the last missing element for executing any instruction, then the entire set of input tokens for this instruction is deleted from the associative token memory and transferred to form a command that is ready for execution.

For storage of ready-to-execute commands, the process *exe_buf* is intended. This process is a simple queue containing all currently ready commands. Accordingly, in this queue, you can add a command along with its ready token arguments, or you can get some command and remove it from the queue.

To implement the work with vector arguments of commands, a vector storage is intended, which is implemented using the *vec_mem* process. Work with this repository is carried out using the vector address. The vector memory interface consists of such requests as: allocate a new vector, get a vector from memory, get a vector element, write a specific value to a vector element, etc.

The logic of the emulator consists of an endless poll of the head process *main*. At each iteration of the survey, the following logic for processing data structures *tokens*, *graph* and *semantic* is performed. If the *tokens* structure contains tokens, then one of them is taken and sent to *ast_mem*. After that, depending on the response of the process *ast_mem*, the token can simply be added to the associative tokens memory, or a ready-to-execute command can be generated and placed in the buffer *exe_buf*. If the structure *tokens* is empty, then the head process receives the first (or arbitrary) command from the ready-made instructions buffer and executes it using the semantics storing in *semantic* structure. Next, the generated tokens of the result of the executed command with the help of the *graph* structure receive information about the destination commands and are added to the *tokens* structure. After that, the program execution cycle continues again until the target tokens are received and the buffer of ready-to-execute instructions is empty.

The described logic of the interaction of the processes of the emulator of a dataflow processor is implemented using the Erlang programming language [10, 11]. Erlang programming language is a functional language designed to implement separate parallel processes that exchange messages. Messaging between processes is the only way to ensure communication between processes (the language does not have the concept of shared memory) it is supported at the syntactic level. Parallel to the execution of processes is a natural behavior of the language; therefore, it is not necessary to implement additional tools to ensure the interaction of system elements.

5. CONCLUSION

The article discusses the differences in the implementation of an executable program designed for a processor with von Neumann architecture and for a dataflow processor. The differences between the def-use graph and the dataflow graph are shown, which reflect the program logic within the linear section. A dataflow processor emulator implementation is considered, including associative token memory, with which it is possible to store and quickly find ready-made data for commands that can be transferred for execution. This dataflow processor emulator architecture can be used for implementing dataflow processor with arbitrary logic and semantic of instructions. This approach to dataflow processor emulator is used in the emulator implementation of a vector dataflow processor which is being developed at the JSCC RAS.

FUNDING

The work has been done at the JSCC RAS as part of the state assignment for the topic 0065-2019-0016 (reg. no. AAAA-A19-119011590098-8). The supercomputer MVS-10P, located at the JSCC RAS, was used during the research.

REFERENCES

1. N. A. Dikarev, B. M. Shabanov, and A. S. Shmelev, "The use of fine-grained parallelism in dataflow processor," in *Proceedings of the Conference on Problems of Developing Promising Micro- and Nanoelectronic Systems (MES)* (2016), Vol. 2, pp. 144–150.
2. D. Culler, "Dataflow architecture," MIT/LCS/TM-294 (Massachusetts Inst. Technol., 1986).
3. R. I. Popov, "Application of stream computer models in designing specialized processors," J. Inform. Technol., Mech. Opt. **5** (75), 77–81 (2011).
4. H. Terada, "DDMP's: Self-timed super-pipelined data-driven multimedia processors," Proc. IEEE **2** (86), 282–296 (1999).
5. N. A. Dikarev and B. M. Shabanov, "Vector dataflow processor," Izv. SFedU **10** (54), 80–85 (2005).
6. S. S. Muchnick, *Advanced Compiler Design smf Implementation* (Academic, New York, 1997).
7. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers. Principles, Techniques, and Tools* (Addison-Wesley, Reading, MA, 2007).
8. B. War, *Dataflow Computers: Their History and Future,Wiley Encyclopedia of Computer Science and Engineering* (Wiley, New York, 2008).
9. J. Silc, B. Robic, and T. Unqerer, *Processor Architecture: From Dataflow to Superscalar and Beyond* (Springer Science, New York, 2012).
10. J. Armstrong, *Programming Erlang. Software for a Concurrent World* (Pragmatic Programmers, 2013).
11. F. Cesarini and S. Thompson, *Erlang Programming* (O'Reilly Media, USA, 2009).