DETECTING DEPENDENCIES IN DATAFLOW KERNELS

Nenad Korolija, PhD[1]
School of Electrical Engineering, University of Belgrade, Serbia

Anton Kos, PhD
Faculty of Electrical Engineering, University of Ljubljana, Slovenia

PURPOSE

The importance of artificial intelligence in detecting crime cannot be overstated. It is used in detecting social engineering attacks (Brkić, 2023), speech recognition and enhancement (Perić, Bogdanoski & Maček, 2022; Popović, 2023), but it is also capable of detecting persons (Hangaragi, Singh & Neelima, 2023; Naik, Kaur, Yathiraju, Das & Pant, 2023), tracking them (Han et al., 2023), and performing complex tasks necessary in detecting crimes (William et al., 2023). The need for artificial intelligence algorithms is constantly rising (Bharadiya, Thomas, & Ahmed, 2023). There are many artificial intelligence algorithms. The common feature of all of them is that they are known to be computation demanding (Zhang, Zhu & Su, 2023). As a result, the execution of artificial intelligence algorithms is often performed in modern cloud infrastructures using dedicated hardware.

Control-flow (CF) architectures are based on the von Neumann computing paradigm. Control-flow architectures execute program instructions in a sequential manner based on the program written by a programmer or programmers. Program usually includes control structures such as conditionals and loops, as well as function calls. These structures allow programs to make decisions, repeat tasks, and organize code into modular units, enabling more complex and flexible behavior. In control-flow architectures, instructions are executed one after the other, following the linear order specified by the program. This sequential execution model is based on the notion of a program counter that points to the next instruction to be executed. Control-flow architectures include branching and jumping mechanisms, allowing programs to alter the sequential flow of execution. Conditional branches allow making decisions in run time based on certain conditions, directing the program's flow. Jump instructions, such as 'goto' in low-level languages, enable non-sequential transfers of control. Many programming languages, such as C, Pascal, and Python, adhere closely to control-flow architectures, emphasizing procedural programming styles where instructions are organized into procedures or functions. In this approach, functions serve as building blocks for modular and reusable code. Control-flow architectures typically exhibit sequential consistency. In other words, instructions are typically executed in the order they appear in the program code, ensuring deterministic behavior. However, many modern architectures employ optimizations that can alter the order of execution of instructions. Also, pipelining is introduced to improve performance while maintaining the illusion of sequential execution from the programmer's point of view.

The main characteristic of such architectures is that they are capable of executing all instructions that the architecture is designed for in any order. Control-flow architectures are therefore flexible, allowing a programmer to write a program with no constraints except the available memory. However, this brings multiple constraints (Lam & Wilson, 1992). The source of one of the main disadvantages of control-flow is in its ability to execute any instruction after any instruction. As a result, executing

1 nenadko@etf.rs

each instruction requires fetching operands, executing the instruction, storing results, and interrupt handling. Many of these utilize the same common bus, resulting in it becoming the bottleneck of the system. Further, loading operands from the memory and storing them presents a burden that is not always necessary. For example, if an instruction calculates a value that is used in the following instruction and nowhere else, it would be beneficial if the semi-result could be forwarded directly from one instruction to another.

Dataflow architectures are designed to process data by defining the transformations of information while it flows through a system (Milutinovic, Salom, Veljovic, Korolija, Markovic & Petrovic, 2017a). Dataflow architecture models computation as a directed graph. Nodes represent processing units that are responsible for executing instructions. Edges represent the flow of semi-results between processing units. The key advantage of dataflow architectures is their ability to exploit parallelism, enabling concurrent execution of independent instructions. By distributing data and computations across multiple processing units, dataflow systems can achieve high throughput and scalability, making them well-suited for handling large volumes of data (Babović et al., 2023a; Kos, Tomažič, Salom, Trifunovic, Valero & Milutinovic, 2015).

Dataflow architectures are often implemented using FPGAs (Field-Programmable Gate Arrays). Unlike traditional integrated circuits (ICs), such as microprocessors or memory chips, which are designed for specific functions and fixed at the time of manufacturing, FPGAs are integrated circuits that can be reconfigured (reprogrammed) by a user. This flexibility allows users to customize the functionality of the FPGA for each algorithm the FPGA should execute. FPGAs are composed of an array of programmable logic and input/output blocks and interconnects. Logic blocks are capable of executing logic functions, such as *AND* and *OR*. The interconnects connect logic blocks together in various configurations, enabling the implementation of complex logic functions. FPGAs find applications in a wide range of fields. They are often used in digital signal processing, telecommunications, and programming. They are also used for lowering the price of chip production by allowing tasks such as simulating ASICs (Application-Specific Integrated Circuits) before they are implemented. However, there are other technologies available for implementing dataflow architectures that can even have better performance but are not often used due to the limitations in technology or funding, e.g., GaAs (Milutinovic, 1986).

Dataflow architectures and control-flow architectures each offer distinct advantages and are suited to different types of computational tasks (Fan et al., 2023). Dataflow architectures have garnered recognition for their ability to accelerate a greater number of instructions per second while simultaneously exhibiting enhanced energy efficiency (Trifunovic et al., 2015). They excel in handling parallelism and are highly efficient for tasks where data dependencies are well-defined and can be executed concurrently (Milutinović, Furht, Obradović & Korolija, 2016). They also consume less power per instruction execution. Many high-performance computing algorithms can be accelerated using dataflow architectures (Korolija, Djukic, Milutinovic & Filipovic, 2013; Cvijetić, Korolija & Vojinović, 2022). On the other hand, control-flow architectures are more versatile and adept at handling unpredictable conditional branching, making them suitable for tasks with complex control structures. Combining these architectures allows for leveraging the strengths of each approach in a complementary manner. This hybrid approach enables efficient execution of both parallelizable and sequential tasks within the same system. Additionally, combining dataflow and control-flow architectures can lead to improved performance and energy efficiency by effectively utilizing available resources and adapting to the varying computational requirements of different tasks (Kos, Ranković, & Tomažič, 2015; Umek & Kos, 2016). Dataflow hardware architectures combined with conventional processors based on control-flow are also capable of accelerating artificial intelligence algorithms.

Nevertheless, in contrast to the development of control-flow algorithms, the development of dataflow algorithms typically demands a deeper knowledge of computer architectures, primarily attributed to the constraints of dataflow hardware. Besides that, additional knowledge is needed to utilize the dataflow hardware controlled by the control-flow program, which inherently necessitates significantly augmented efforts in terms of programmer tasks (Popovic, Bojic & Korolija, 2015), particularly when working on non-functional requirements (Popović, Korolija, Marković & Bojić, 2017). The increased effort stems from the need to adapt programming methodologies to accommodate the operational characteristics of dataflow architectures since the development phase for dataflow applications is predisposed to extended debugging, along with more time needed to optimize performance within the prescribed hardware limitations. Certain methods for automation of translation of control-flow algorithms to the dataflow paradigm have been developed to reduce the effort needed for programming dataflow architectures (Korolija, 2023).

The additional constraint to combining control-flow and dataflow architectures is the scheduling. Researchers have developed relatively fast scheduling algorithms due to the nature of high-performance computing tasks running on dataflow architectures that are usually executed during a considerable amount of time compared to the relatively fast tasks that are often executed using control-flow hardware (Korolija, Bojic, Hurson & Milutinovic, 2022).

Another important aspect of the combined control-flow and dataflow architectures is the expected lifetime and counterfeit detection. The duration is expected to be around the shortest, lasting from these two incorporated architectures, and the possible counterfeit processor chips could be identified using existing methods (Huang, Liu, Korolija, Carulli & Makris, 2015).

Researchers and the programming community have noticed the need for providing dataflow algorithms accessible in the open literature. As a result, the application gallery has been developed (Trifunovic, Milutinovic, Korolija & Gaydadjiev, 2016; Milutinovic et al., 2017), allowing users to view the source code of certain dataflow algorithms.

The aim of this research is to investigate the possibilities of applying existing research in the field of computer science on advancing dataflow architectures to allow faster artificial intelligence algorithm execution (Babović et al., 2023b). The existing research includes the research in the domain of compilers and in the domain of parallelizing the execution of algorithms. The presentation is founded upon the proposed method (Milutinovic, 1996) and has been evaluated by the authors in alignment with the proposed evaluation criteria (Banković et al., 2020).

DESIGN/METHODS/APPROACH

This article introduces a novel approach to automatically identify dependencies within and between dataflow kernels, essential for optimizing parallel processing tasks on dataflow hardware (Korolija, Popović, Cvetanović & Bojović, 2017). The method outlined entails constructing a dependency graph using a designed tool, facilitating a visual representation of the relationships between different data elements and operations within the kernel, but also letting the tool detect parallelizable portions of source code. By utilizing this tool, developers can efficiently analyze and comprehend the intricate dependencies present in their dataflow kernel code, thus enabling them to fine-tune performance and enhance parallelism. The article not only highlights the theoretical framework behind the proposed method but also offers practical insights through the inclusion of example code, illustrating the step-by-step process of dependency detection. The versatility of the proposed methodology extends

to hybrid processors, which integrate both control-flow and dataflow hardware components within the same chip die (Korolija, Jelisavčić, Minchev & Milutinović, 2022; Milutinović et al., 2021; Milutinović, Trifunović, Korolija, Popović & Bojić, 2017; Miladinović, Bojović, Jelisavčić & Korolija, 2022; Korolija & Štrbac-Savić, 2024). This adaptability may be crucial in contemporary computing architectures where heterogeneous processing units coexist to exploit the advantages of both paradigms (Milfeld, Korolija, 2022). By accommodating hybrid processors, the method becomes applicable across a broader spectrum of computing systems, including heterogeneous architectures, cluster, and cloud computing, ensuring its relevance and utility in diverse hardware environments. Ultimately, this article serves as guidance for developers seeking efficient ways to optimize dataflow kernels and leverage the capabilities of dataflow hardware for enhanced computational performance.

Compilers are software tools designed to translate source code written in a high-level programming language that is suitable for programmers into machine code so that it can be executed directly by a computer's processor. The process begins with the call to transform the input of the source code. The compiler begins by performing lexical analysis, thus breaking down the source code into tokens such as keywords, identifiers, operators, and literals. This step is important in recognizing the basic building blocks of the language and grouping them into meaningful units for easier processing. The following step is the syntax analysis, or parsing these tokens. The compiler performs syntax analysis, so that the structure of the code is checked against the rules of the programming language's grammar. In order to do this, the compiler constructs a parse tree or abstract syntax tree that represents the hierarchical structure of the code and verifies that it conforms to the language syntax. After syntax analysis, the compiler performs semantic analysis to check the meaning of the code. This involves type checking, symbol resolution, and other checks to ensure that the code adheres to the language's semantics and constraints. If the source code is analyzed and validated, the compiler can generate an intermediate representation of the code, which is a platform-independent and optimized representation that serves as an intermediate step before generating the final machine code. Many compilers include the possibility to optimize the code by applying various transformations to the intermediate representation to improve the efficiency and performance of the generated code. In this step, the unreachable or dead code can be eliminated, loop can be optimized, etc. Finally, the compiler translates the resulting intermediate representation of the code into the corresponding machine code suitable for execution on the computer architecture. This requires selecting appropriate instructions from the target instruction set and arranging the execution in a sequential manner to implement equal or optimized behavior. The output is typically an executable binary code that can be run directly on the computer architecture.

The crucial aspect of the compilers for this research is the ability to generate an intermediate representation of the code, serving as a bridge between the original source code and the subsequent analysis processes needed for detecting dependencies between instructions. By providing a structured and abstracted view of the code, this representation enables the readability of the algorithm for detecting dependencies and reduces the debugging time.

Detecting dependencies in source code in the general case requires identifying relationships between different components within software. In the case of dataflow architectures, these components may represent kernels but also instructions. Dependencies in source code can be of various types, such as variable references, function calls, etc. One common type of dependency in source code is when one function calls another function. In the case of dataflow programming, the same logic might be applied to detect dependencies between kernels. This logic involves analyzing the source code to identify function calls and determining the relationship between each pair of functions where one function calls the other. Within a single dataflow kernel, one could detect variable references, as dependencies that arise when two instructions use the same variable. Generally, if a variable is declared in one por-

tion of the source code and used in another as well, there is a dependency between those two portions of source code. Detecting variable references involves tracking the declaration and usage of variables throughout the source code.

In object-oriented programming languages, classes can inherit behavior from other classes, optionally overriding methods defined in parent classes. Detecting these dependencies involves understanding the inheritance hierarchy and determining dependencies between classes. A similar problem can be found in dataflow hardware programming, but this work doesn't cover these aspects of detecting dependencies. The same applies for dependencies between file imports. In modular programming, code is often organized into separate files or modules that can depend on each other. However, dataflow kernels are usually defined each in its file, without the need for importing a relatively considerable number of files.

Various static analysis tools can be used to detect dependencies in source code automatically. These tools parse the source code in order to build an internal representation of the program's structure and then analyze it to identify dependencies. Once dependencies are detected, they can be represented using dependency graphs. Dependency graphs provide a visual representation of the relationships within the source code, making it easier to understand and manage complex dependencies. Overall, detecting dependencies in source code can be done for many reasons. Some of them include understanding the structure and behavior of software, identifying bottlenecks, and improving the code by refactoring and optimizing. In this research, the dependency graph is created to support parallelizing the dataflow programs automatically.

Detecting dependencies between variables in C++ in the general case can be challenging due to the language's flexibility and the possibility of indirect relationships through function calls and pointers to data. However, even a relatively simple tool can detect direct dependencies between variables in source code and enable parallelization (Korolija, Furht, & Milutinović, 2023). The process or steps involved in creating and using the software for detecting dependencies will be clearly outlined. This is essential so that anyone following these steps can achieve the same results and outcomes.

The code from Algorithm 1 represents *dependency_graph.h* file. The following classes and functions are introduced. The Variable class represents variables in the input source code. For each variable, ID, type, and name are stored. Constructors are responsible for variable initialization, with a default name or a specified one. Method *print()* displays relevant information about the variable. The Dependency class represents a dependency relationship between two variables (Variable *variable* whose name is stored in the field *variable.name* depends on *dependsOn* variable). Method *print()* prints the details of the dependency in a clear format. The *DependencyGraph* class is in this example rather a mock class that holds only the most important information for the graph. It manages a collection of dependencies (dependencies vector). The *addDependency()* method adds a new dependency to the graph. Dependencies are stored as pairs of variables where the first one depends on the second one. The *printGraph()* method prints all dependencies stored in the graph.

```
#ifndef DEPENDENCY_GRAPH_H
#define DEPENDENCY_GRAPH_H
#include <iostream>
#include <vector>
#include <fstream>
#include <string>
```

```
// Enum defining types of variables in the dependency graph
enum VariableType { CONSTANT, SCALAR, ARRAY };
// Represents a variable in the dependency graph
class Variable {
public:
   Variable();                        // Default constructor
   Variable(std::string varName); // Constructor with variable name
   void print();                      // Prints information about the variable
private:
   static int nextId;                 // Static member to generate unique IDs
   int id;                           // Unique identifier for the variable
   VariableType type;                 // Type of the variable (constant, scalar, array)
   std::string name;                  // Name of the variable
};
// Represents a dependency relationship between two variables
class Dependency {
public:
   // Constructor with two variables:
   Dependency(Variable &variable, Variable &dependsOn);
   void print();                      // Prints information about the dependency
private:
   Variable variable;                 // Variable that depends on another
   Variable dependsOn;                // Variable on which another depends
};
// Manages the dependency graph of variables and their relationships
class DependencyGraph {
public:
   DependencyGraph();                 // Constructor for initializing the graph
   // Adds a dependency between variables:
   void addDependency(Variable &var, Variable &dependsOn);
   void printGraph();                 // Prints the entire dependency graph
private:
   std::vector<Dependency> dependencies; // List of dependencies in the graph
};
#endif // DEPENDENCY_GRAPH_H
```

Algorithm 1. *The Declaration of Classes for Detecting Dependencies*

The following code represents only the implementation of function *main()* from the file *dependency_graph.cpp*. Implementation of explained functions is trivial and more complex function implementation will be explained in this article.

```
#include “dependency_graph.h”
int main() {
   DependencyGraph graph;
   std::ifstream inFile(“test_program.cpp”);
   std::ofstream outFile(“resulting_graph.txt”);
   std::cout << “Reading test program...” << std::endl;
   std::string line;
   getline(inFile, line); // Read first variable name
   Variable varA(line); // Construct the variable
   getline(inFile, line); // Read “=” sign
   getline(inFile, line); // Read second variable name
   Variable varB(line); // Construct the variable
   getline(inFile, line); // Read “+” sign
   getline(inFile, line); // Read third variable name
   Variable varC(line); // Construct the variable
   outFile.close();
   inFile.close();
   // Claim dependencies between the first and the second variable:
   graph.addDependency(varA, varB);
   // Claim dependencies between the first and the third variable:
   graph.addDependency(varA, varC);
   graph.printGraph();
   return 0;
}
```

Algorithm 2. *The Implementation of the Main Function*

The function *main()* reads c++ statements from *test_program.cpp* to extract variable names and create instances of variables. Further, it adds dependencies (*varA* depends on *varB* and *varC*) and stores them in the DependencyGraph. Finally, it prints out the dependency graph in a structured format. This C++ program illustrates the construction and visualization of a dependency graph based on variable dependencies parsed from a hypothetical C-like source file (*test_program.cpp*).

The presented structured approach is invaluable for tasks such as code optimization, refactoring, and understanding program logic. It provides clarity and insight into the inter-dependencies within statements of a C++ program or between dataflow kernels.

The improved version of the algorithm introduces in Variable class *operator* ==, necessary for comparing variables so that there are no two variables with the same name, and methods *setRead()* and *setWrite()*. The Dependency class is also appended by the *operator* ==, so that no two dependencies exist with the same pair and order of variables. However, the biggest change is in the function main, where the most important portion is shown in Algorithm 3.

```
int fileLineNumber = 0;
while (getline(inFile, line)) {
    fileLineNumber++; // Increment line number

    std::cout << std::endl << "Parsing file line: " << line << std::endl << std::endl;
    std::vector<std::string> dependsOnList;
    // Parse expression using parser
    parse(line, dependsOnList);
    std::cout << "Lhs token detected: " << Parser::_lhsToken << std::endl;
    std::string lhsTokenString = Parser::_lhsToken;
    Variable *lhsVar = VarManager::find(lhsTokenString);
    if (!lhsVar) {
        Variable temp(lhsTokenString);                  // Create temporary variable
        lhsVar = VarManager::find(lhsTokenString); // Find variable again to get its reference
    }
    lhsVar->setWrite();
    std::cout << "List of variables that lhs depends on: " << std::endl;
    for (auto &dependsOn : dependsOnList) {
        std::cout << dependsOn << std::endl;
        Variable *rhsVar = VarManager::find(dependsOn);
        if (!rhsVar) {
            Variable temp(dependsOn);               // Create temporary variable
            rhsVar = &globalVars.back();            // Get reference to the last added variable
            lhsVar = VarManager::find(lhsToken); // Find variable again to get its reference
        }
        rhsVar->setRead();
        graph.addDependency(*lhsVar, *rhsVar, fileLineNumber);
    }
}
//...
VarManager::printDetailed();
graph.printGraph();
```

Algorithm 3. *Improvements in the Implementation of the Main Function*

The updated version of the *main()* function orchestrates the creation of a dependency graph. Here, each line of the input test program is read and processed to identify dependencies between variables. The most important feature is that variable and dependency handling is implemented to support multiple dependencies. For each input file line (i.e., expression), it parses the line using a *parse()* function, whose implementation is available in the open source, except for a few changes needed to support detecting dependencies. The left-hand side (LHS) variable of the expression is identified (*lhsToken*), and its dependency relationships are determined by parsing the expression. While parsing a statement, square brackets are also allowed to support vectors. Vector *_dependsOnList* is updated whenever a dependency is detected in a statement.

While the *Variable* class represents variables, maintaining their names and read/write status, *VarManager* is introduced to provide static methods *printAll()*, *printDetailed()*, and *find()*. It is responsible for managing and accessing all variables created during the process of detecting dependencies and printing results. The improved version of the algorithm displays the list of variables found on the LHS, the list of all variables along with flags showing whether each of them is read from, written to, or both, and the constructed dependency graph where each dependency is assigned a unique statement number. This algorithm can help software developers understand how variables depend on each other within a program, which is crucial for tasks such as debugging, optimization, and restructuring of their code.

Improving the parsing of loops in the source code is implemented using a modified lexer available in the parser source code. This involves refining the process of tokenizing and interpreting user code. A lexer, or lexical analyzer, breaks down C++ statements into tokens, which are the smallest units of meaning in the C++ language. To optimize this process, several strategies can be employed, including optimizing the lexer's implementation for efficiency, leveraging efficient data structures, and optimizing regular expression patterns used for token recognition.

Parsing multiline loops is implemented by interpreting and understanding consecutive statements that span across multiple lines. To effectively parse multiline loops, recognizing the beginning and end of the loop construct and identifying loop control statements (e.g., conditions for loops and iteration parameters) is needed. This is implemented in our source code, without modifying the parser, so that the scope of the parser is limited to parsing a single C++ statement.

In compilers, loop parsers typically use recursion or similar techniques to navigate the hierarchical structure of nested loops. These methods allow parsers to accurately associate loop bodies with their respective loop headers. Error handling is also needed in compilers for parsing multiline loops. However, in our algorithm, parsing is limited to parsing loops that are not nested, without error handling mechanisms.

Further, the parsing algorithm is improved by implementing functions to associate identifiers to statements where read or write occurred. This will be explained using the *setRead()* function. The Variable class represents variables in a program and tracks where they are read from (read) and written to (write). The *setRead()* function is responsible for recording when a variable is read by pushing a specific statement identified into the vector of statement identifiers associated with the variable.

The statement identifier is defined as an integer representing the unique identifier of the statement where the variable is read. The *setRead()* adds a statement identifier to the read vector associated with the current instance of *Variable*. This vector keeps track of all statement IDs where this variable has been read. In the user C++ source code, whenever an expression or assignment involving variables is parsed, the program identifies which variables are read and written to within a statement. As an example, when parsing a line of source code where a variable *v* is read in a statement with identifier 5, the *setRead()* function call for the variable *v* would append number 5 to its read vector, indicating that variable *v* is read in a statement with identifier 5.

The parsing algorithm is designed to build a dependency graph between variables or kernel arguments and outputs based on their read and write operations across multiple statements or kernels. This graph helps analyze how variables depend on each other throughout the code, enabling automatic parallelization of the code using the dataflow paradigm.

By tracking all statements of the user source code and tracking reads using the function *setRead()* and writes using *setWrite()* for each variable, the program constructs a comprehensive picture of data dependencies, essential for tasks like parallelization of dataflow kernels or understanding data flow in complex algorithms.

Further improvement of our parsing algorithm includes detecting kernel calls in order to be able to detect dependencies between dataflow kernels. The same algorithm is also applicable to detecting function calls.

The *parseKernel()* function is designed to parse a line of the user source code that potentially contains a function call or a call to the dataflow kernel. The *parseKernel()* function also determines if the function call involves a return value assignment and extracts details about the function and its arguments. This function takes the reference to the dependency graph where dependencies between variables are stored, the statement user file line where a kernel or function call may occur, and an integer representing the current maximum statement identifier. The function first initializes an input string stream with a given file line, enabling the lexer to tokenize this line. The value of the return variable is initially assigned the first token, presuming it represents the variable to which the function's return value is assigned.

Identifying kernel or function call is implemented in the parsing algorithm, apart from the existing C++ code for parsing a single statement. First, the equal sign is searched for, which represents a statement where a function has a return value. This is not used when parsing kernel calls. No expressions are allowed where the argument can be replaced with an expression containing the equal sign. Next, the kernel name is extracted using the lexer from the existing C++ code for parsing a single statement. In order to claim that the statement represents a kernel of function call, the following token to be parsed has to be the open bracket token. At this stage, it is assumed that the statement is a kernel of a function call.

Parsing arguments are implemented similarly. First, a token is extracted, representing a variable name. Following, the arguments separated with the comma sign are recognized. This process is repeated until the closing bracket token is found.

The *parseKernel()* function plays a crucial role in identifying possibilities for reorganizing and parallelizing kernel calls. It also contributes to constructing a dependency graph that tracks relationships between variables based on function calls and assignments.

In the improved version of the parsing algorithm, the *Kernels* class is implemented to manage dataflow kernels and their relationships. It also supports managing function calls within the user source code. It supports initializing a kernel, finding the index of a function by name, adding a relationship between two kernels, etc.

Rearranging kernel calls and code execution based on dependencies involves ensuring that code is executed in the correct order to satisfy dependencies between different parts of the program. The first step in this process is the identification of dependencies. Dependencies can be of various types, such as data dependencies, control dependencies, or kernel dependencies. For example, a statement that uses the value of a variable initialized elsewhere represents a data dependency. We are mostly interested in dataflow kernel dependencies. The analysis of dependencies determines the correct order of execution. This may involve examining the flow of data and control within the program to understand how different statements and kernel and function calls depend on each other. After this step, the rearranging of code execution can be done. Based on the analysis of dependencies, for each statement or kernel of function call, it can be determined how far it can be moved up or down without affecting the

result of the program execution. This may involve moving blocks of statements or even restructuring the program logic if the programmer finds a useful guide in the rearranged code, which can further lead to the performance optimization by minimizing unnecessary dependencies and maximizing parallelism where possible.

In the existing algorithm for the automatic parallelization of the source code, the synchronization mechanisms are not applied. In cases where dependencies between different parts of the code cannot be resolved through rearrangement alone, a mechanism can be implemented to coordinate the execution of dependent parts of the code using locks, mutexes, semaphores, or barriers to ensure that certain tasks are completed before others begin.

Overall, detecting the dependencies between statements and kernel and function calls in the source code is a critical aspect of improving the performance of a dataflow program. As the dataflow programming model is inherently parallel, the incorrect execution order can lead to race conditions, deadlocks, or other synchronization issues. By automatically analyzing dependencies and organizing code accordingly, it can be ensured that the user program executes correctly and efficiently.

FINDINGS

The execution of the proposed algorithm is depicted using the C++ source code given in Algorithm 4. The comments in the source code are set to be equal to the statement identifier of each statement. As a result, the first statement in the *for* loop has three identifiers, equal to the number of *for* loop iterations.

```
using namespace std; // 1
#include <vector> // 2
#include <iostream> // 3
unsigned int foo(unsigned int a1, unsigned int a2){ // 4
   return a1 * 10 + a2; //5
} // 6
int main(){ // 7
   unsigned int i, a, b, c, d, e, v[10]; // 8
   a = 10; // 9
   b = 5; // 10
   c = a + b; // 11
   d=1+(b  *(c+   6))^2; // 12
   e = b; // 13
   v[0] = 1; // 14
   for(i = 0; i < 3; i++){ // 15
      v[i] = a; // 16, 18, 20
      c = v[i] + 1; // 17, 19, 21
   } // 22
   v[0] = a * e; // 23
   c = foo(a, b); // 24
}
```

Algorithm 4. *Example of User Source Code*

The resulting dependency graph for the function *main()* of the example algorithm is given in Table 1. One may note that the dependency table for the function is purposely omitted, as it contains no dependencies.

Table 1. *Dependency Graph of the Parsed Source Code*

	a	b	c	d	e	v[0]	v[1]	v[2]
a								
b								
c	11, 24	11, 24				17	19	21
d		12	12					
e		13						
v[0]	16, 23				23			
v[1]	18							
v[2]	20							

The presented algorithm can also detect function calls and dependencies among them. The same logic is applicable to kernel calls. The results of executing the presented algorithm with multiple kernels detect dependencies between kernels, enabling automatic parallelization of kernel execution. Exiting frameworks for transforming the code from the control-flow to the dataflow paradigm can generate the VHDL files and further configure the dataflow hardware based on the user definition of kernels implemented in a high-level programming language (e.g., MaxJ).

The execution flow of the Maxeler framework is as follows. The code written in a high-level language, e.g., *C*, prepares scalar inputs used in kernels and signals for multiplexers and demultiplexers using a conventional approach by calling a function defined in the library. The same convention is used for calling the dataflow hardware to execute the most time-consuming portion of the algorithm using the dataflow hardware and return results. Multiple kernels can be initiated directly from the *C* code but also using the manager. In this scenario, the manager for kernels acts as a kernel itself, connecting input signals to the manager to appropriate inputs of kernels it controls. After the manager code is executed, the kernels will be connected by the framework. In this research, authors didn't consider parsing the manager itself, as it is specific to the company providing the framework. On the other hand, calling kernels directly from a high-level language, e.g., *C*, is common for all libraries provided by various hardware vendors. The detection of dependencies between the execution of functions is depicted using the C++ source code given in Algorithm 5.

```
int aa(int a1, int a2){
    return a1 * 10 + a2;
}
int bb(int b1, int b2){
    return b1 * 10 + b2;
}
int cc(int c1, int c2){
    int d;
    d = bb(c1, c2) * aa(c1, c2);
    return d;
}
int main(){
    int a, b, c, d, e;
    a = 10;
    b = 5;
    d = 3;
    e = 4;
    c = aa(a, b);
    b = cc(e, d);
}
```

Algorithm 5. *Example of Function Call Dependencies*

The result or parsing function calls of the source code is given in Table 2. This is a fairly simple example with the purpose of showing the possibility of parsing function definitions and function calls, forming the graph of function calls. In order to be able to determine the possibility of moving the execution of a function before or after the line where it is executed, dependencies between arguments have to be taken into account as well. This is achieved by treating function calls in the same manner as expressions. For example, if a result of a call to function *aa* with arguments *a* and *b* is stored in a variable *c*, as shown in Algorithm 5, the dependency graph would be the same as if there was a statement $b = e + d$ in the same line of the source file. For each variable, the proposed algorithm keeps track of statement identifiers where it is read or written to so that the statement execution could be moved without affecting the result of the execution of user source code.

Table 2. *Dependency Between Functions*

	main	aa	bb	cc
main	-	+	-	+
aa	-	-	-	-
bb	-	-	-	-
cc	-	+	+	-

Along with the dependency graph between statements of the user code generated with the proposed algorithm, this further supports automatic translation of multiple statements of the user source code into the dataflow architecture by choosing the most suitable set of instructions based on the acceleration using the dataflow hardware for a given user input. Although user input is not known before running the application in a general case, the algorithms that are executed using the dataflow hardware often have the number of iterations known in advance, before executing the algorithm.

Executing algorithms in a cloud environment assumes sharing multiple computing resources by multiple algorithms. In the case of dataflow programming, this means sharing dataflow hardware units by multiple kernels, i.e., a single dataflow card can be attached to the multicore processor using the PCIe bus (this is only one of the possible configurations). Sharing a single dataflow card among multiple algorithms implies that there will be multiple kernels executing simultaneously. If the scheduler recognizes that the same dataflow algorithm is required to be executed in the cloud, these can be executed by sharing the dataflow hardware in such a manner that a separate kernel can exist for different instances of the algorithm, and partially occupied kernels can be shared among these instances. This can lead to an even higher acceleration factor when using dataflow hardware in a cluster or a cloud for the execution of artificial intelligence algorithms on the same type of input files (e.g., frames from a video surveillance system). Example applications can be found in many machine learning algorithms that already exploit the benefits of reconfigurable dataflow architectures (Filippas et al., 2024; Yu et al., 2024).

Figure 1 depicts the acceleration possibility in the case where there are multiple instances of the Lattice Boltzmann algorithm running on a single dataflow card (i.e., dataflow jobs), where each instance consists of five kernels, four for processing edge elements of all matrices, and one for processing other elements of matrices.

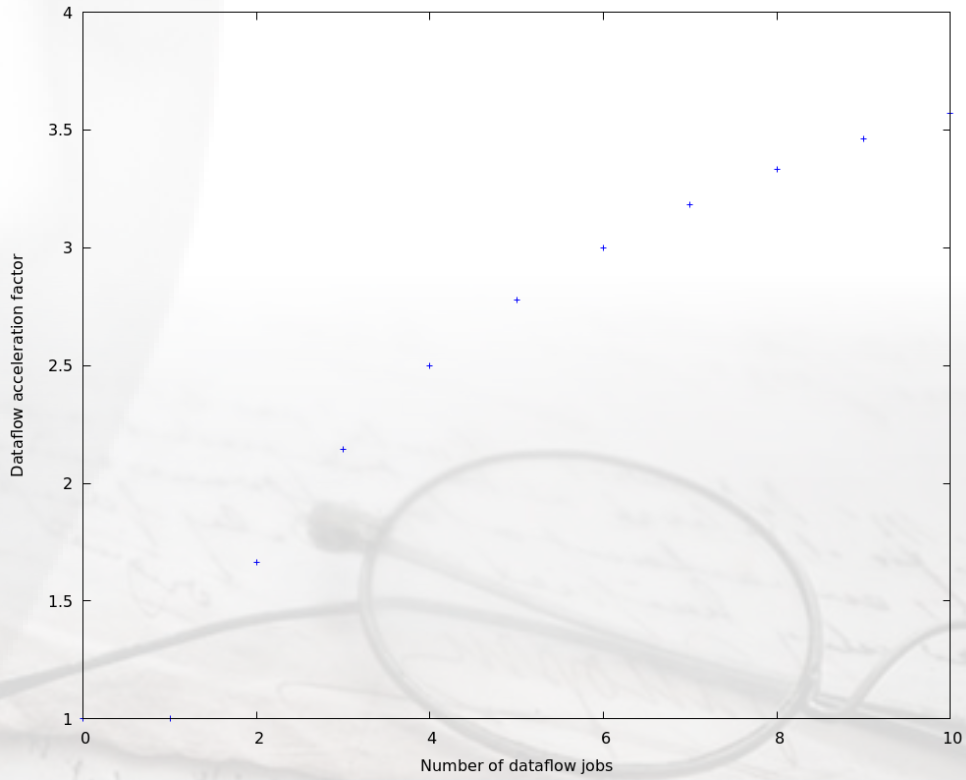

Figure 1. *Dataflow Hardware Acceleration When Combining Dataflow Kernels of Lattice Boltzmann Method*

In the case of executing the method on matrices that have reasonable sizes, the processing time of kernels that process edge elements is neglectable compared to the execution time of the kernel that processes other elements. In this scenario, four kernels processing edge elements can be shared among all dataflow jobs. We can assume that kernels occupy approximately the same chip die surface. This way, the acceleration factor that can be achieved using the same chip area is equal to the ratio between the number of kernels needed if all instances would have to use five of their kernels divided by the number of kernels in the case of sharing edge kernels among all dataflow jobs.

ORIGINALITY/VALUE

While the acceleration possibilities of parallelizing the code execution using the dataflow hardware exist, not much work is performed to automatically transform control-flow algorithms into dataflow algorithms. This can be inferred from the relative scarcity of automated tools and frameworks specifically designed for this purpose compared to those available for control-flow paradigm, such as parallelization or optimization. There are multiple reasons why this is the case. In this chapter, several possible reasons will be explained.

Converting control-flow algorithms into dataflow representation is inherently challenging due to the fundamental differences in the execution models of these paradigms. Control-flow algorithms rely on explicit execution of statements in sequences, allowing branching of instructions, whereas dataflow algorithms express computations as a flow of data between operations, allowing no changes in the execution unless using special conditional statements that assume that both scenarios are implemented in the dataflow hardware during the program execution, but that only one path will be useful. Transforming control-flow to dataflow programs requires significant restructuring of the code and often introducing new data dependencies and parallelism, making it considerably complex but also error-prone (Milutinović, Trifunović, Korolija, Popović & Bojić, 2017; Milutinovic, Salom, Veljovic, Korolija, Markovic & Petrovic, 2017b).

Unlike loop parallelization or optimization and similar transformations, there is no widely accepted generalized solution or methodology for automatically transforming control-flow algorithms into dataflow counterparts. While some research has been done in this area, much of it is experimental and focused on specific domains or problem types, rather than providing general-purpose solutions applicable across a wide range of applications. There are some frameworks, but with limited functionalities, e.g., for creating a dataflow kernel written in a programming language specially designed for programming dataflow hardware using a similar language to Java.

The transformation from control-flow to dataflow is highly dependent on the specific characteristics and requirements of the application or problem domain. Different applications may require different dataflow structures or optimizations, making it difficult to develop generalized transformation techniques that work effectively in all cases. Also, the performance of the dataflow hardware implementation of a control-flow algorithm depends on the input data the dataflow hardware will work with. Since the frequencies of control-flow hardware are an order of magnitude higher than those of dataflow hardware, if there is not much computation, it is more justifiable to execute the code using the control-flow hardware.

Automating the transformation process may require significant computational resources. Analyzing control-flow algorithms, identifying data dependencies, and restructuring the code to automatically produce a dataflow algorithm is computationally intensive and may require sophisticated static anal-

ysis techniques and optimization algorithms. In the case of the debugging, the resource requirements are even higher, especially if the simulation of the transformed algorithm should be performed on a low level.

Dataflow programming models are not suitable for applications usually executed on personal computers but rather only for high-performance algorithms executed using computer clouds or clusters. Most of the existing software is designed for control-flow computing architectures and may not benefit from being transformed into a dataflow paradigm.

In the conclusion, while there has been some research and experimentation in transforming control-flow algorithms into dataflow hardware automatically, the lack of widely adopted tools and frameworks suggests that this area remains relatively underexplored compared to tools and experimentation in the field of control-flow hardware. The importance of executing artificial intelligence algorithms faster and with less power consumption justifies the effort in automating the translation of control-flow algorithms to their dataflow counterparts. The manual transformation loses in importance over time, as high-performance computing architectures become more complex and capable. As a result, managing many kernels in a cloud in real time can only be done automatically. Further development of the presented algorithm should take into account dependencies in the streams on the element-level, and not only by treating streams as parameters.

REFERENCES

Brkić, M. (2023). A system architecture for preventing social engineering attacks via e-mail. Journal of Computer and Forensic Sciences, 2(1), 43–52.

Perić, D., Bogdanoski, M., & Maček, N. (2022). Application of convolutional neural networks to spoken words evaluation based on lip movements without accompanying sound signal. Journal of Computer and Forensic Sciences, 1(1), 7–16.

Popović, B. (2023). Speech enhancement by CycleGAN using feature map regularization. Journal of Computer and Forensic Sciences, 2(1), 19–28.

Hangaragi, S., Singh, T., & Neelima, N. (2023). Face Detection and Recognition using Face Mesh and Deep Neural Network. Procedia Computer Science, 218, 741–749.

Naik, M. N., Kaur, A., Yathiraju, N., Das, S., & Pant, K. (2023, May). Improved and Accurate Face Mask Detection Using Machine Learning in the Crowded Places. In 2023 3rd International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE) (pp. 572–576). IEEE.

Han, X., You, Q., Wang, C., Zhang, Z., Chu, P., Hu, H., ... & Liu, Z. (2023). Mmptrack: Large-scale densely annotated multi-camera multiple people tracking benchmark. In Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (pp. 4860–4869).

William, P., Shrivastava, A., Shunmuga Karpagam, N., Mohanaprakash, T. A., Tongkachok, K., & Kumar, K. (2023). Crime analysis using computer vision approach with machine learning. In Mobile Radio Communications and 5G Networks: Proceedings of Third MRCN 2022 (pp. 297–315). Singapore: Springer Nature Singapore.

Bharadiya, J. P., Thomas, R. K., & Ahmed, F. (2023). Rise of Artificial Intelligence in Business and Industry. Journal of Engineering Research and Reports, 25(3), 85–103.

Zhang, B., Zhu, J., & Su, H. (2023). Toward the third generation artificial intelligence. Science China Information Sciences, 66(2), 121101.

Lam, M. S., & Wilson, R. P. (1992). Limits of control flow on parallelism. ACM SIGARCH Computer Architecture News, 20(2), 46–57.

Milutinovic, V., Salom, J., Veljovic, D., Korolija, N., Markovic, D., & Petrovic, L. (2017). DataFlow supercomputing essentials: research, development and education. Springer.

Babović, Z., Bajat, B., Đokić, V., Đorđević, F., Drašković, D., Filipović, N., ... & Zak, S. (2023). Research in computing-intensive simulations for nature-oriented civil-engineering and related scientific fields, using machine learning and big data: an overview of open problems. Journal of Big Data, 10(1), 73. Springer.

Kos, A., Tomažič, S., Salom, J., Trifunovic, N., Valero, M., & Milutinovic, V. (2015). New benchmarking methodology and programming model for big data processing. International Journal of Distributed Sensor Networks, 11(8), 271752.

Milutinovic, V. (1986). Guest Editor's Introduction GaAs Microprocessor Technology. Computer, 19(10), 10–13.

Fan, Z., Li, W., Tang, S., An, X., Ye, X., & Fan, D. (2023, August). Improving utilization of dataflow architectures through software and hardware co-design. In European Conference on Parallel Processing (pp. 245–259). Cham: Springer Nature Switzerland.

Trifunovic, N., Milutinovic, V., Salom, J., & Kos, A. (2015). Paradigm shift in big data supercomputing: dataflow vs. controlflow. Journal of Big Data, 2(1), 1–9. Springer.

Milutinović, V., Furht, B., Obradović, Z., & Korolija, N. (2016). Advances in high performance computing and related issues. Mathematical problems in engineering, 2016.

Korolija, N., Djukic, T., Milutinovic, V., & Filipovic, N. (2013). Accelerating Lattice-Boltzman method using Maxeler dataflow approach. The IPSI BgD Transactions on Internet Research, 34.

Cvijetić, D., Korolija, N., & Vojinović, M. (2022). Possibilities for Parallelizing Simplicial Complexes Simulation.

Kos, A., Ranković, V., & Tomažič, S. (2015). Sorting networks on Maxeler dataflow supercomputing systems. In Advances in computers (Vol. 96, pp. 139–186). Elsevier.

Umek, A., & Kos, A. (2016). The role of high performance computing and communication for real-time biofeedback in sport. Mathematical Problems in Engineering, 2016.

Popovic, J., Bojic, D., & Korolija, N. (2015). Analysis of task effort estimation accuracy based on use case point size. IET Software, 9(6), 166–173.

Popović, J., Korolija, N., Marković, Ž., & Bojić, D. (2017, November). The influence of non-functional requirements in UCP method on the accuracy of effort estimates. In 2017 25th Telecommunication Forum (TELFOR) (pp. 1–4). IEEE.

Korolija, N. (2023, May). Drawbacks of Programming Dataflow Architectures and Methods to Overcome Them. In Serbian International Conference on Applied Artificial Intelligence (pp. 57–70). Cham: Springer Nature Switzerland.

Korolija, N., Bojic, D., Hurson, A. R., & Milutinovic, V. (2022). A runtime job scheduling algorithm for cluster architectures with dataflow accelerators. Advances in Computers, 201.

Huang, K., Liu, Y., Korolija, N., Carulli, J. M., & Makris, Y. (2015). Recycled IC detection based on statistical methods. IEEE transactions on computer-aided design of integrated circuits and systems, 34(6), 947–960.

Trifunovic, N., Milutinovic, V., Korolija, N., & Gaydadjiev, G. (2016). An AppGallery for dataflow computing. Journal of Big Data, 3(1), 1–30. Springer.

Milutinovic, V., Salom, J., Veljovic, D., Korolija, N., Markovic, D., Petrovic, L., ... & Petrovic, L. (2017). Maxeler AppGallery Revisited. DataFlow Supercomputing Essentials: Research, Development and Education, 3–18.

Babović, Z., Bajat, B., Barac, D., Bengin, V., Đokić, V., Đorđević, F., ... & Zak, S. (2023). Teaching computing for complex problems in civil engineering and geosciences using big data and machine learning: synergizing four different computing paradigms and four different management domains. Journal of Big Data, 10(1), 89. Springer.

Milutinovic, V. (1996). The best method for presentation of research results. IEEE TCCA Newsletter, (9), 1–6.

Banković, M., Filipović, V., Graovac, J., Hadži-Purić, J., Hurson, A. R., Kartelj, A., ... & Živković, M. (2020). Teaching graduate students how to review research articles and respond to reviewer comments. In Advances in Computers (Vol. 116, No. 1, pp. 1–63). Elsevier.

Korolija, N., Jelisavčić, V., Minchev, Z., & Milutinović, V. (2022). Towards hybrid control-flow and dataflow architectures. Archibald Reiss Days, 12.

Milutinović, V., Azer, E. S., Yoshimoto, K., Klimeck, G., Djordjevic, M., Kotlar, M., ... & Ratkovic, I. (2021, June). The ultimate dataflow for ultimate supercomputers-on-a-chip, for scientific computing, geo physics, complex mathematics, and information processing. In 2021 10th Mediterranean Conference on Embedded Computing (MECO) (pp. 1–6). IEEE.

Korolija, N., Popović, J., Cvetanović, M., & Bojović, M. (2017). Dataflow-based parallelization of control-flow algorithms. In Advances in computers (Vol. 104, pp. 73–124). Elsevier.

Miladinović, D., Bojović, M., Jelisavčić, V., & Korolija, N. (2022, June). Hybrid manycore dataflow processor. In Proceedings, IX International Conference IcETRAN, Novi Pazar, Serbia (pp. 6–9).

Korolija, N., & Štrbac-Savić, S. (2024). Merging control-flow and dataflow architectures on a single chip. Journal of Computer and Forensic Sciences.

Milfeld, K. & Korolija, N. (2022). Towards hybrid supercomputing architectures. Journal of Computer and Forensic Sciences, 1(1), 47–54.

Korolija, N., Furht, B., & Milutinović, V. (2023). Fine Grain Algorithm Parallelization on a Hybrid Control-flow and Dataflow Processor.

Milutinović, V., Trifunović, N., Korolija, N., Popović, J., & Bojić, D. (2017, November). Accelerating program execution using hybrid control flow and dataflow architectures. In 2017 25th Telecommunication Forum (TELFOR) (pp. 1–4). IEEE.

Milutinovic, V., Salom, J., Veljovic, D., Korolija, N., Markovic, D., & Petrovic, L. (2017). Transforming applications from the control flow to the dataflow paradigm. In Dataflow supercomputing essentials (pp. 107–129). Springer, Cham.

Filippas, D., Peltekis, C., Titopoulos, V., Kansizoglou, I., Sirakoulis, G., Gasteratos, A., & Dimitrakopoulos, G. (2024). A High-Level Synthesis Library for Synthesizing Efficient and Functional-Safe CNN Dataflow Accelerators. IEEE Access.

Yu, Z., Sreeram, S., Agrawal, K., Wu, J., Montgomerie-Corcoran, A., Zhang, C., ... & Zhao, Y. (2024). HASS: Hardware-Aware Sparsity Search for Dataflow DNN Accelerator. arXiv preprint arXiv:2406.03088.