Procedural Code Representation in a Flow Graph

Michal Brabec and David Bednárek

Parallel Architectures/Algorithms/Applications Research Group
Department of Software Engineering
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 25, Prague, Czech Republic
{brabec,bednarek}@ksi.mff.cuni.cz

Abstract. Modern scientific computing often combines extensive calculation with complex structure of data; however, the programming methodologies and languages of high-performance computing significantly differ from those of databases. This impedance mismatch leads many projects to the use of either primitive (like JSON) or overly general (like distributed file systems) methods of data access, ignoring the decades of development in database technology. In this paper, we investigate the possibility to represent procedural code fragments using a network of operators similar to query plans used in relational database systems. Such a unified representation forms the necessary step towards an integrated computational-database platform. We propose a flow graph representation that allows us to analyze, transform and optimize applications more efficiently and without additional data. Along with the graph, we designed an algorithm that transforms a procedural code into the graph.

Keywords: compiler, graph, optimization, parallelism

1 Introduction

Modern data processing often combines complex data layouts with intensive calculations. Despite the ongoing effort in the area of no-SQL databases, the traditional relational paradigm, especially in its column-based version [2], still offers unmatched maturity and efficiency up to multi-terabyte ranges. However, the database systems were not designed with general computing in mind.

Systems based on the MapReduce paradigm allow the programmer to integrate general procedural code with a distributed data storage more easily. Despite of their popularity, MapReduce implementations may still be outperformed by parallel databases even in brute-force tasks where the sophisticated database approach has seemingly no advantage [20]. However, the same experiments also show that the performance dominance of parallel databases is limited to workloads implemented by built-in functions; as soon as user-defined functions are required, the performance falls rapidly.

This observation shows that the runtime stages of modern parallel database systems are extremely efficient even under brute-force computing load. However, this efficiency is degraded by the inability of the relational optimizers to

M. Nečaský, J. Pokorný, P. Moravec (Eds.): Dateso 2015, pp. 89–100, CEUR-WS.org/Vol-1343.

efficiently handle procedural code fragments contained in user-defined functions [15]. Nevertheless, with a different front-end, a parallel database system may become a suitable runtime for parallel computing.

Such a front-end would have to compile procedural code into physical execution plans used in database systems. Modern databases, as well as streaming systems, use graph-based execution plans whose nodes are not limited to relational algebra operators, as shown by the successful adaptation of many relational engines to XML or RDF [17].

In this paper, we present *flow graphs* – an intermediate code capable to represent procedural code with its complex control-flow. Unlike typical intermediate representations used in compilers, the flow graphs are designed to be similar to pipeline-based models used in many database, streaming, and general parallel platforms.

Besides the introduction of flow graphs, we also describe the key algorithms which take part in the transformation of procedural code into flow graphs. The algorithms described here are applied after language-specific phases like parsing and semantic analysis and they also make use of analytical algorithms which are already frequently used in compilers.

The rest of the paper is organized as follows: After reviewing the related work, flow graphs are defined in Section 3. Section 4 presents the transformation algorithms; in Section 5, we revise possible optimizations of the flow graph.

2 Related Work

The flow graph described in this paper is similar but not identical to other modeling languages, like Petri nets [21] or Kahn process networks [14]. The main difference is that the flow graph was designed for automatic generation from the source code, where the other languages are generally used to model the application prior to implementation [16] or to verify a finished system [9]. The flow graph is similar to the graph transformation system [8], which can be used to design and analyze applications, but it is not convenient for execution. There are frameworks that generate GTS from procedural code like Java [7], though the produced graphs are difficult to optimize. The flow graph has similar traits to frameworks that allow applications to be generated from graphs, like UML diagrams [12] [4], but we concentrate both on graph extraction and execution.

The flow graph is closely related to graphs used in compilers, mainly the dependence and control flow graphs [3], where the flow graph merges the information from both. The construction of the flow graph and its subsequent optimization relies on compiler techniques, mainly *points-to* analysis [23], *dependence* testing [18] and *control-flow* analysis [22]. In compilers, graphs resulting from these techniques are typically used as additional annotation over intermediate code.

The flow graph is not only a compiler data representation, it is a processing model as well, similar to KPN graphs [13]. It can be used as a source code for specialized processing environments, where frameworks for pipeline parallelism

are the best target, since these frameworks use similar models for applications [19]. One such a system is the Bobox framework [10], where the flow graph can be used to generate the execution plan similarly to the way Bobox is used to execute SPARQL queries [11].

3 Flow Graph

The *Flow graph* was designed as a compact format for representation of procedural code. Once constructed, it contains the code along with the information about its structure, including control flow and data flow, and it can be transformed back into procedural code. In this section, we define the flow graph and we explain its relation to other processing models.

A *flow graph* is a directed graph, where nodes represent operations and edges represent data exchange among the operations. The direction of the edge indicates the direction of data flow (source and sink operations). Figure 1 shows an unoptimized flow graph for a function without branches (see Listing 1 for source code). The gray nodes denote dead code and they will be removed during later optimization steps. The flow graph can become complicated once control flow is introduced – see Figure 5 for an example of a more complex graph that implements a program with a loop.

```
void Statements() {
    int a = 3;
    int b = 5;
    int c = 0;
    c = a + b;
    print(c);
}
```

Listing 1. Simple function without branches

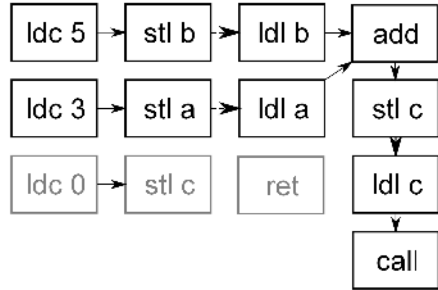

Fig. 1. Simple function transformed to an unoptimized flow graph

For a particular domain of application, a *specific flow graph language* is defined that contains a set of *basic operations* and a set of *data types*. We construct flow graphs based on such platform specific language. Both nodes and edges contain information about the represented operation or data type respectively. Each edge is connected to a specific input or output of the node according to the represented operation (the data type must be compatible).

As our research is focused on C#, we use CIL [1] instructions to specify node operations. In this paper, there are four most common instruction types: *ldl x* (load variable or constant), *stl x* (write to a variable), *ble* (conditional branch),

add etc. (mathematical operations). We omit data types for the edges, because they are not important for the graph construction.

3.1 Execution Model

The *flow graph execution model* defines the way nodes process data and communicate. Nodes represent operations and edges represent unbounded queues (FIFO).

Operations have three states - *waiting for input*, *processing* and *inactive*. Each operation starts waiting for input, it *fires* once input is available, processes the input and produces an output. Once the input is processed, the operation again waits for data. Nodes without input (loading constants) fire at the beginning of the execution, produce data and then they become inactive.

The queues transport single values of the assigned data type (based on the edge). Nodes must always consume their input, they cannot simply check the data and leave them in the queue. For example a simple node, which adds two numbers, fires when there are data in both its incoming queues, it consumes both numbers and then is stores the result in its outgoing queue.

3.2 Special Nodes

Loops and branches of the source language are transformed into a graph of platform independent *special nodes* which interact with the data-flow carried through the *basic nodes* that perform the basic operations.

An *extended primary node* is a special version of any basic operation without parameters, like load constant value. The extended version has a single input and it restarts whenever the input contains data.

A *broadcast node* has a single input and a variable number of outputs. It represents a simple operation that creates a copy of the input for each output. This node is created whenever an operation must pass its result to multiple operations and the number of receivers defines the number of outputs. A *loop feedback node* is a special type broadcast node that distributes the positive result of a conditional branch to all extended primary nodes.

A *merge node* has a single output and three inputs. It represents an operation that accepts data from two sources and passes them to a single operation and it is used to merge data flow after a conditional branch. The node has an extra input used to get feedback from another node, generally a conditional branch. The node fires when all three inputs contain data.

A *loop merge node* is a special version of a merge node with two inputs for conditional branches, it is used to merge data in loops. The input is split into two pairs, where each contains one data input and one branch input. The node fires, whenever both inputs in a pair contain data. The node is either positive or negative, where the conditional inputs are required to be either positive or negative, for passing data into a loop or outside a loop.

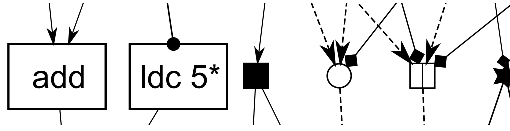

Fig. 2. Node types: basic, extended primary, broadcast, merge, loop merge, feedback

4 Flow Graph Construction

In this section, we present a two-phase algorithm that transforms a procedural code to a flow graph. The input to the algorithm is an intermediate code; in our case, the CIL. For simplicity, we assume that the CIL code was compiled from a C# source without unsafe code and goto. We also assume that the code is first subjected to a points-to analysis [23] which resolves potential aliasing problems.

The basic idea of the algorithm is that each CIL instruction is transformed into a node. Edges are generated according to the data used by each instruction. Edges correspond to the inputs / outputs of the instructions. When the basic graph is ready, we add special nodes for broadcasting and merging data according to branches and loops in the code. For simplicity, we ignore function calls in this description – see Section 4.1 for more information on function handling.

Basic Graph Construction The first step of the algorithm is to create basic nodes according to the instructions of the source code. We use basic operations in the first part, the special operations are introduced in the second part. Algorithm 1 contains all the necessary steps and it produces the basic nodes N and edges E of the graph.

We can create a node for every CIL instruction, because we defined the operations based on the instruction set (lines 1 to 3 in Algorithm 1).

Next, we analyze how instructions exchange data, either using registers or stack, and we create edges that connect the source and sink instruction. CIL instructions communicate using virtual stack and we use a stack simulator to analyze the way the instructions exchange data. Then we create edges between nodes representing instructions that exchange data (lines 4 to 6 in Algorithm 1), along with the appropriate data type. The analysis is similar when the instructions communicate through registers, only instead of stack simulator, we have to connect instructions that use the same register.

Then, we have to take into account the data passing through variables. We have to create edges between nodes representing variable access, from write to read. This step is more complex, because we have to connect every variable read with the nearest write, along all the possible execution paths, which means analyzing the control flow.

We generate basic blocks for the input code. In every basic block, we locate all the instruction that access any variable (lines 7 to 11 in Algorithm 1). First we create edges inside every block, where we connect variable writes to reads if

```
.method void Statements() cil
  ldc.i4.3
  stloc.0
  ldc.i4.5
  stloc.1
  ldloc.0
  ldloc.1
  add
  stloc.2
  ldloc.2
  call    void print(int32)
  ret
```

Listing 2. CIL code of Statements function

```
void BranchElse() {
  int a = 3;
  int c = 0;
  if (a > 0)
    c = 4;
  else
    c = -4;
  print(c);
}
```

Listing 3. Simple branch

both access the same variable but only if the read is after the write and there is no other write between them (lines 12 to 16 in Algorithm 1). Results of this step are illustrated in Figure 1 which shows the intermediate flow graph based on the CIL code in Listing 2, generated from the source code in Listing 1.

Next, we have to connect variables between basic blocks. We locate all reads before the first write in every block, for every variable. We call them accessible reads. We connect the last write in a basic block to all the accessible reads in the blocks that follow the original, until one of them contains an instruction that writes to the same variable (lines 17 to 26 in Algorithm 1). Basically, we perform an exhaustive search through the block graph, where we stop at nodes that change the studied variable. This way the data is propagated through the control flow.

Figure 3 shows a flow graph generated from the function in Listing 3. There is one conditional jump that produced the two edges that lead to the node *ldl c*. The branch instruction must decide which input is used (Section 4). It is important that the initialization of c to 0 is identified as unused code.

Control Flow Management In this section, we present algorithms necessary to make the graph produced by the Algorithm 1 deterministic and compliant to the flow graph definition. We must make sure that the nodes have a proper number of inputs and outputs and that the branching conditions deterministically decide what value is used at every time, especially in loops. Algorithm 2 contains all the transformations for handling control flow.

We start this algorithm by locating loops in the basic block graph. We locate all the blocks L of the inner-most loop and we locate the block L_b that contains the backward conditional jump b that restarts the loop (it is sufficient to locate the backward jump, because we consider a restricted C#). We duplicate the entire block L_b as L_{b_i}, the copy drives the first iteration which is different, since it uses data initialized before the loop started (initialization of variables). See

Algorithm 1. Basic graph construction

Require: I – set of instructions i
B – basic blocks
M – set containing all memory locations (variables)
C_i – all instructions consuming the output of the instruction i
R_i – variables read by instruction i
W_i – variables read by instruction i
Ensure: N – nodes of the flow graph
E – edges of the flow graph
1: **for all** $i \in I$ **do**
2: $N = N \cup \{N_i : operation(N_i) = O_i\}$ – nodes based on instructions
3: **end for**
4: **for all** $N_i \in N$ **do**
5: $E = E \cup \left\{E_{N_i N_j} : j \in C_i\right\}$ – edges based on instruction communication (stack)
6: **end for**
7: **for all** $b \in B$ **do**
8: $Load_{bm} = \{i \in b : m \in R_i \wedge w \in b < i \implies m \notin W_w\}$ – read before update
9: $Load_{bmj} = \{i \in b : m \in R_i \wedge j < i \wedge w \in [j, i] \implies m \notin W_w\}$ – read after write
10: $Store_{bm} = \{i \in b : m \in W_i \wedge w \in b > i \implies m \notin W_w\}$ – last update in block
11: **end for**
12: **for all** $b \in B$ **do**
13: **for all** $m \in M$ **do**
14: $E = E \cup \left\{E_{N_j N_i} : i \in Load_{bmj}\right\}$ – edges based on variable access
15: **end for**
16: **end for**
17: **for all** $m \in M$ **do**
18: **for all** $b \in B$ **do**
19: **for all** $n_0 \in B : next(b, n_0)$ **do**
20: $E = E \cup \left\{E_{N_j N_i} : j \in Store_{bm} \wedge i \in Load_{bn_0}\right\}$ – edges between blocks
21: **if** $Store_{bn} = \emptyset$ **then**
22: recursion for $\{n_1 \in B : next(n_0, n_1)\}$
23: **end if**
24: **end for**
25: **end for**
26: **end for**

Listing 4 where i is first compared while it still has the value of 1. We locate all nodes ni inside the loop with more incoming edges than inputs (line 1 in Algorithm 2). We create loop merge nodes m for all ni (line 2 in Algorithm 2), redirect the incoming edges to the merge nodes (line 3 in Algorithm 2). We add edge E_{mni}. Finally we connect the merge nodes to the conditional branches and we pair the input coming from outside the loop to the duplicate branch in b_i in L_{b_i} and the other with the branch b in L_b (line 4 in Algorithm 2). This ensures that the first iteration takes the data from outside and all the rest take the internal data. See Figure 5 for complete loop with merge nodes.

When the loops are fitted with merge nodes, we add a loop feedback node that restarts all the nodes without input. We replace all nodes without input in

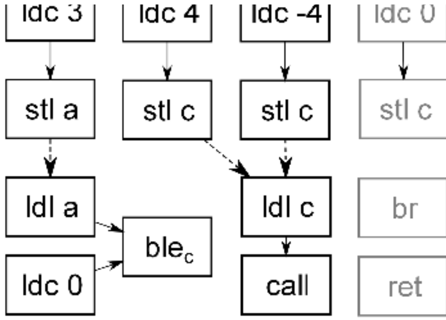

Fig. 3. Basic graph of a conditional branch

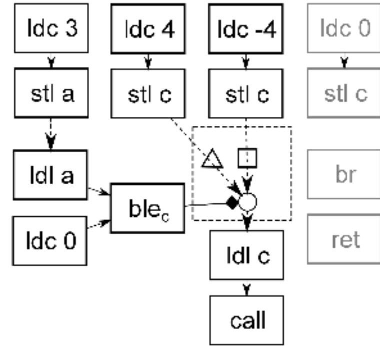

Fig. 4. Completely transformed branch

```
void SingleLoop()
{
    for (int i = 1;
         i <= 5;
         i++)
    {
        print();
    }
}
```

Listing 4. Simple loop function

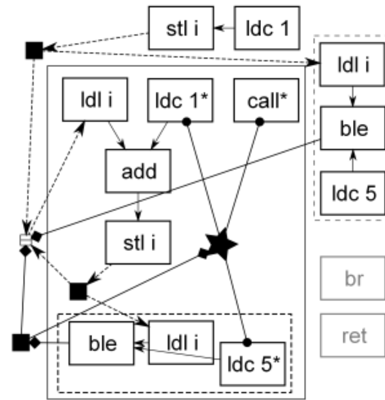

Fig. 5. Flow graph of a loop

a loop by their extended version, see Section 3.2. We connect the feedback node to the branch in Br_l and to all the extended nodes in the loop (lines 6 to 10 in Algorithm 2). Figure 5 shows a complete loop, where the Br_l and Brc_l are outlined by a dashed rectangle and the entire loop by a solid rectangle.

When all loops are transformed, we must locate all remaining nodes n with more incoming edges than inputs (line 11 in Algorithm 2), the multiple inputs are the result of conditional branches. Figure 3 shows the situation where two edges lead to a node with a single input (*ldl c*). We solve this situation by introducing a merge node m along with the edge E_{mn} (lines 12 to 13 in Algorithm 2). Then we redirect the two input edges to the m (line 14 in Algorithm 2). Finally, we have to locate the conditional branch responsible for the merge and connect it to the merge node. We can do this by following the paths from source nodes, where we locate the branch just before the paths join, ble_c (branch if $a \leq 0$) in Figure 3. The result of this algorithm is in Figure 4, where the edge with a square is the positive input and the triangle is negative.

The final step is to locate all nodes n with more outgoing edges than outputs. This is solved simply by using a broadcast node. We create a new broadcast node b, we add an edge E_{nb} and we change the outgoing edges to start in b (lines 16 to 20 in Algorithm 2).

Algorithm 2. Iteration over stripes

Require: B – basic blocks
Ensure: N – nodes of the flow graph
E – edges of the flow graph
1: **for all** $\{ni \in N : \exists L \subset B \wedge ni \in L \wedge |E_{xn}| > inputs(ni)\}$ **do**
2: $N = N \cup \{m\}$ – add loop merge node
3: $\forall E_{xni} : x \in N \implies E = (E \setminus \{E_{xni}\}) \cup \{E_{xm}\}$
4: $E = E \cup \{E_{mni}\} \cup \{E_{bm}\} \cup \{E_{b_i m}\}$ – b and b_i are conditional jumps of the loop
5: **end for**
6: **for all** $\{\forall\ L \subset B : loop(L)\}$ **do**
7: $N = N \cup \{f\}$ – add loop feedback node
8: $E = E \cup \{E_{bf}\}$ – where b is conditional jump of the loop
9: $\forall e \in L : extended(e) \implies E = E \cup \{E_{fe}\}$
10: **end for**
11: **for all** $\{n \in N : |E_{xn}| > inputs(n)\}$ **do**
12: $N = N \cup \{m\}$ – add merge node
13: $E = E \cup \{E_{mn}\} \cup \{E_{bm}\}$ – where b is the conditional jump
14: $\forall E_{xni} : x \in N \implies E = (E \setminus \{E_{xn}\}) \cup \{E_{xm}\}$
15: **end for**
16: **for all** $\{n \in N : |E_{nx}| > outputs(n)\}$ **do**
17: $N = N \cup \{b\}$ – add broadcast node
18: $E = E \cup \{E_{nb}\}$
19: $\forall E_{nx} : x \in N \implies E = (E \setminus \{E_{nx}\}) \cup \{E_{bx}\}$
20: **end for**

4.1 Functions and Methods

Methods are analyzed starting with the main function and then the graph is constructed as additional methods are called. Whenever a method call is located, we create a flow graph for the called method, treating its parameters as local variables. We connect the parameters to their actual values (source variables or constants). This approach is equivalent to complete procedure integration and it is applicable only for programs whose call graph is acyclic and contains reasonable number of paths. Using additional special nodes, any program might be transformed; however, at the cost of additional operations which correspond to the call-return pairs in conventional program execution. In the supposed application domain, the general approach is probably unnecessary.

4.2 Objects and Arrays

A variable of complex data type (object or array) can contain a number of memory locations (members or elements) that can be accessed using special instructions. We treat member data of objects as separate variables where the same members of two independent objects are different variables. Arrays can be viewed as objects with a single member - data, where the data contains multiple independent values. Arrays are treated this way by many compiler algorithms [23].

5 Graph Optimizations

A flow graph produced by the algorithm presented in Section 4 is generally very big, since we transform every instruction into a single node, which is not very convenient for execution, but it can be efficiently analyzed and optimized. In this section, we shortly introduce optimizations that can produce more compact graphs.

Each node in a flow graph, as defined in Section 3, represents a basic or special operation. We introduce the *merge rules*, to allow optimizations of the graph. The merge rules define the way the operations are combined to produce complex operations. They define the behavior (source code) of the complex operation and its inputs and output along with their data types. The merge rules are added to the definition of the specific flow graph language.

A complex operation is created by merging other operations, either basic or complex, according to the merge rules defined along with the graph. The merge rules for CIL instructions contain for instance chaining of the instructions. For example, when merging the addition instr. in $A + B + C$, we create a complex operation that is equal to $\sum_1^3 In_i$.

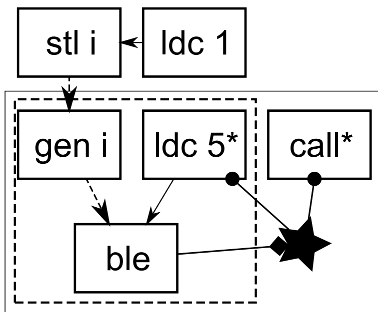

Fig. 6. Merged simple loop

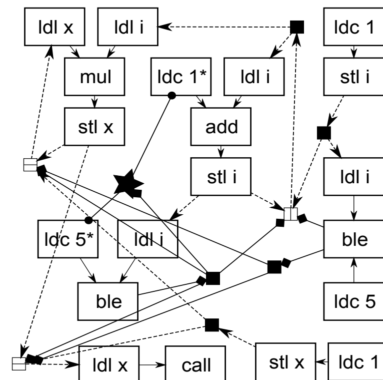

Fig. 7. Factorial computation graph

Another transformation is aimed at simple loops controlled by a single variable. Figure 5 shows a very simple loop that is controlled by the variable i, updated in every iteration. The loop creates many unnecessary dependences.

When we locate such a simple situation, we can merge the entire loop into a single node that just generates appropriate values in this case $\{1, 2, 3, 4\}$. We can utilize algorithms used in compilers [18] to locate the loops, because the graph contains all the information found in the original source code. Figure 6 shows how the loop from Figure 5 is optimized. This optimization is essential for improving the efficiency of the flow graph, compare the optimized graph to an unoptimized graph implementing the computation of a factorial, Figure 7.

6 Conclusions

We designed the flow graph to represent a procedural code along with important information about its structure and behavior. We designed an algorithm that allows us to create a flow graph for an application implemented in a subset of C# and compiled to CIL. This transformation becomes a part of a toolchain that allows the transformation of C# programs into a stream-based parallel computing platform [5]. The algorithm can be modified for other languages, like Java bytecode [6].

The flow graph is a powerful tool for application analysis and optimization. Besides generating pipeline-based execution plans, the flow graph can be used for automatic parallelization. For such use, the original flow graph may be too fine-grained – in this case, it has to be transformed using a set of merge rules to make the final parallel application efficient.

Acknowledgements

This paper was supported by Czech Science Foundation (GAČR) project P103-13-08195, and by the Grant Agency of Charles University Grant Agency (GAUK) project 122214.

References

1. TG3. Common Language Infrastructure (CLI). Standard ECMA-335, June 2005
2. Abadi, D., Boncz, P.A., Harizopoulos, S., Idreos, S., Madden, S.: The design and implementation of modern column-oriented database systems. Foundations and Trends in Databases 5(3), 197–280 (2013)
3. Allen, R., Kennedy, K.: Optimizing compilers for modern architectures. Morgan Kaufmann San Francisco (2002)
4. Balasubramanian, D., Narayanan, A., van Buskirk, C., Karsai, G.: The graph rewriting and transformation language: Great. Electronic Communications of the EASST 1 (2007)
5. Brabec, M., Bednárek, D., Malý, P.: Transformation of pipeline stage algorithms to event-driven code. In: Kurkova, V., Bajer, L., Svátek, V. (eds.) Proceedings of the 14th Conference on Information Technologies - Applications and Theory, Jasna, Slovakia, 2014. CEUR Workshop Proceedings, vol. 1214, pp. 13–20. CEUR-WS.org (2014), http://ceur-ws.org/Vol-1214

6. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: FlumeJava: easy, efficient data-parallel pipelines. In: ACM Sigplan Notices. vol. 45, pp. 363–375. ACM (2010)
7. Corradini, A., Dotti, F.L., Foss, L., Ribeiro, L.: Translating java code to graph transformation systems. In: Graph Transformations, pp. 383–398. Springer (2004)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Graph transformation systems. Fundamentals of Algebraic Graph Transformation pp. 37–71 (2006)
9. Ezpeleta, J., Colom, J.M., Martinez, J.: A Petri net based deadlock prevention policy for flexible manufacturing systems. Robotics and Automation, IEEE Transactions on 11(2), 173–184 (1995)
10. Falt, Z., Kruliš, M., Bednárek, D., Yaghob, J., Zavoral, F.: Locality aware task scheduling in parallel data stream processing. In: Camacho, D., Braubach, L., Venticinque, S., Badica, C. (eds.) Intelligent Distributed Computing VIII, Studies in Computational Intelligence, vol. 570, pp. 331–342. Springer International Publishing (2015)
11. Falt, Z., Čermák, M., Dokulil, J., Zavoral, F.: Parallel SPARQL query processing using Bobox. International Journal On Advances in Intelligent Systems 5(3 and 4), 302–314 (2012)
12. Geiger, L., Zündorf, A.: Graph based debugging with fujaba. Electr. Notes Theor. Comput. Sci. 72(2), 112 (2002)
13. Geilen, M., Basten, T.: Requirements on the execution of Kahn process networks. In: Programming languages and systems, pp. 319–334. Springer (2003)
14. Gilles, K.: The semantics of a simple language for parallel programming. In: Information Processing: Proceedings of the IFIP Congress. vol. 74, pp. 471–475 (1974)
15. Guravannavar, R., Sudarshan, S.: Rewriting procedures for batched bindings. Proceedings of the VLDB Endowment 1(1), 1107–1123 (2008)
16. Josephs, M.B.: Models for data-flow sequential processes. In: Communicating Sequential Processes. The First 25 Years, pp. 85–97. Springer (2005)
17. Mayer, S., Grust, T., Van Keulen, M., Teubner, J.: An injection with tree awareness: adding staircase join to postgresql. In: Proceedings of the Thirtieth international conference on Very large data bases-Volume 30. pp. 1305–1308. VLDB Endowment (2004)
18. Muchnick, S.S.: Advanced compiler design implementation. Morgan Kaufmann Publishers (1997)
19. Navarro, A., Asenjo, R., Tabik, S., Cascaval, C.: Analytical modeling of pipeline parallelism. In: Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on. pp. 281–290. IEEE (2009)
20. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. pp. 165–178. ACM (2009)
21. Peterson, J.L.: Petri nets. ACM Comput. Surv. 9(3), 223–252 (Sep 1977), http://doi.acm.org/10.1145/356698.356702
22. Reps, T.: Program analysis via graph reachability. Information and software technology 40(11), 701–726 (1998)
23. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. ACM SIGPLAN Notices 41(6), 387–400 (2006)