2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems

DFGenTool: A Dataflow Graph Generation Tool for Coarse Grain Reconfigurable Architectures

Manideepa Mukherjee, Alexander Fell, Apala Guha
IIIT-Delhi
New Delhi, India
Email: {manideepam, alex, apala}@iiitd.ac.in

***Abstract*—In this paper, DFGenTool, a dataflow graph (DFG) generation tool, is presented, which converts loops in a sequential program given in a high-level language such as C, into a DFG. DFGenTool adapts DFGs for mapping to Coarse Grain Reconfigurable Architectures (CGRA) to enable a variety of CGRA implementations and compilers to be benchmarked against a standard set of DFGs. Several kernels have been converted and are presented in this paper as case studies. The output of DFGenTool is in DOT, a popular graph description standard which could be used with a variety of CGRA compilers. Furthermore, DFGenTool has been released as open-source.**

I. Introduction

In System on Chips (SoC) the increasing demand for high computation capacities resulted in advancements of General Purpose Processors (GPP) in terms of high clock frequencies, complex instruction execution units and distributed computing. In addition the aspect of mobility emerged, in which battery operated devices are also equipped with GPPs. To provide sufficient computation capabilities and to prolong the battery life of mobile devices, these processors are surrounded by Application Specific Integrated Circuits (ASIC) destined to perform specific tasks. The manufacture of ASICs inflicts high Non-Recurring Engineering (NRE) costs and long time to market periods which require high volumes to remain profitable. Coarse-Grained Reconfigurable Architectures (CGRA) is an emerging class of reconfigurable architecture. They are similar to FPGAs, however the LUTs have been replaced by Processing Units (PU) and the interconnect by a packet-based segmented bus system called Network-on-Chip (NoC). The PU consists of a local memory, an Arithmetic Logic Unit (ALU) and an interface to the router establishing connectivity to other PUs through the NoC. The set of PUs and the interconnect form the *Execution Fabric* providing the required computational capacities. Compared to FPGAs, CGRAs are not programmed on the bit-level, but on the level of instructions due to the presence of ALUs. Therefore a powerful high level language such as C can be used to implement algorithms which are then compiled into instructions understood by the CGRA. To execute a sequential C program on a CGRA, a Data Flow Graph (DFG) is generated in which the vertices are instructions and the edges represent the dependencies among these instructions.

A DFG visualizes the Instruction Level Parallelism (ILP) in CGRAs. The DFG needs to be transformed to match the structure given by the Execution Fabric. Scheduling instructions temporally and mapping them spatially onto PUs has been in the focus of research and many heuristics exist such as [7], [8], [9], [10]. On the architectural side many CGRAs have been proposed till date such as ADRES[14], KressArray[11], Layered CGRA[19], [20], PACT XPP[5], [17] and REDEFINE[3]. For testing and benchmarking, algorithms have been adopted, compiled and executed for each of the platforms by the respective authors. However, it is difficult to compare performance in terms of execution time and power consumption across CGRAs because the code used by each research group is not available to the rest of the community. To alleviate the situation, we propose a DFG generator, which is able to convert an algorithm written in C/C++ into a DFG, that can be used by the community for benchmarking their respective platforms.

We implemented our tool in LLVM [12], which is a popular, open-source compiler platform. We provide detailed documentation and maintain the tool at http://github.com/manideepam/DFGenTool so that it can be used and extended by the community. Furthermore, our tool produces the DFGs based on LLVM Intermediate Representation (IR) code, which has two advantages. The first advantage is that IR code is not tied to a particular front-end or high-level language, rather IR code can be produced from any high-level language that LLVM already provides a front-end for. Since the LLVM front-end covers several popular high-level languages and will add more in the future, the tool can work with a large set of languages. The second advantage is that IR code is not tied to a particular back-end, and yet resembles the instruction set architecture of most hardware platforms i.e. there are analogous hardware instructions for most IR instructions in common architectures. In addition IR lacks register bindings thereby providing a higher-level abstraction for data. Finally, we output the DFGs in the `dot` format [2] which is amenable to visulization as well as automated processing by popular graph processing software such as `Boost` C++ Library [1] and python networkx.

The key contributions that we make in this paper are the following:

- A tool to build DFGs from LLVM IR.
- Documentation and maintenance of the tool in a GitHub repository.
- Optimizations that adapt DFGs for CGRAs.
- An output module that stores graphs in the `dot` format.
- A set of DFGs of kernels in the Polybench 3.2 linear algebra [18] benchmark suite.

2380-6923/16 $31.00 © 2016 IEEE
DOI 10.1109/VLSID.2017.62

The rest of the paper is organized as follows: Section II discusses related works, then continues with the implementation process and optimizations tailoring DFGs for use in CGRAs in section III. Results are shown in section IV followed by conclusions.

II. Related Work

The paper in [4] describes a DFG generation method to transform a C code into a dataflow graph. The GNU Compiler Collection (GCC) is used to create an intermediate representation called GIMPLE[15] which is a reduced subset of intermediate code GENERIC[15]. GIMPLE creates blocks of functions connected through control edges and the DFGs are created for each block. Although this method can be used to transform a C code into a dataflow graph, this will not directly work for CGRAs as the optimizations related to CGRAs (such as Φ node removal) has not been addressed here. In addition user is forced to use GCC compiler if GIMPLE is used where as DOT does not have this limitation. Paper[13] gives an algorithm for automatic extraction of coarse grained data-flow threads from imperative programs. This is a thread based data flow graph generation algorithm targeted for general purpose processors. Although, this would be interesting to map these coarse grain data flow graphs onto CGRAs, choosing the optimum thread size that can be fitted in the memory of a PU needs further research. Moreover the algorithm is limited to scalar data only. Turbine[6] and DFTtools[16] are DFG generation tools available online as open source. They can generate graphs by setting parameters such as the number of nodes required, input and output edges, their weight etc. which are used to generate a DFG randomly. The tools cannot generate a DFG based on an algorithm and are therefore not suitable for benchmarking specific applications.

The purpose of this paper is to provide an open source DFG generation tool that can be used to generate DFGs for analysis, mapping, transformation and comparison of results.

III. Implementation

The goal of this work is to design a DFG generator tool that is 1) open-source, 2) works for multiple high-level languages, 3) adapted to CGRAs, 4) independent of the characteristics of a particular CGRA chip, 5) outputs results in a popular stand-alone format, and, 6) is easily expandable.

Figure 1 shows the schematic diagram of the DFG generation tool from high level C/C++ code to the dataflow graph output. The tool is composed of four parts. The first part generates intermediate code from high level C/C++ code using the Clang compiler and it is briefly discussed in section III-A. Secondly, the DFG generation tool consists of stages including inner loop extraction and initial graph formation followed by two optimizations specific to CGRAs (refer to sections III-B and III-C). The last part described in section III-D, converts the graph to a dot file as output.

A. Intermediate Code Generation

LLVM has a frontend compiler called Clang, to translate high level C/C++ code to Intermediate Representation (IR) code (.ll/.bc). We build our tool at the IR code level to make it platform independent. Listing 1 is an example that will be used to demonstrate DFGenTool.

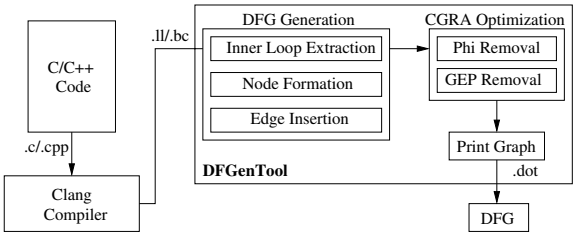

Fig. 1: Schematic diagram of DFG generation tool

```
1 for(i=0; i<n; i++) {
2     if(i%2 == 0)
3         data[i] = i;
4     else
5         data[i] = 2*i;
6     sum = sum + data[i];
7 }
```

Listing 1: A simple C code

Listing 2 shows the generated intermediate code for listing 1. It consists of four basic blocks with unique identifiers (labels) 7, 10, 13 and 17. The first basic block contains the intermediate code for the `if` condition. The second block assigns values to `data[i]`, if the condition is true, while block 13 handles the assignment for the else branch. Block 17 is the intermediate code for the final summation. The generated DFG for this intermediate code is shown in figure 2.

B. DFG Generation Tool

In this section the DFG generation tool is described in detail. As most of the scientific and linear algebra kernels spend a significant amount of their execution time in loops, different loop optimizations are used to improve the loop performance[18]. Since the innermost loop has the largest number of iterations, any optimization applied to this loop level has a high impact on the overall performance of the kernel. First it extracts the innermost loop or function from the intermediate code using LLVM after which it forms the nodes and edges. However, the DFGenTool is not limited to loops only and it can also be used to generate DFGs for a complete function in a program.

1) Node Formation: Each instruction in the inner loop is represented by a node in the graph. Most instructions have two source operands and one destination operand. Since LLVM intermediate code is in static single assignment (SSA) form, the destination operand (if present) uniquely identifies the instruction. Therefore, the source operand of an instruction itself could be an instruction node. If the source operands are not instruction nodes, new operand nodes are created as data nodes.

To increase the readability of the graph, different types of nodes are given distinguishable shapes and border lines. The different shapes and borders used for DFG nodes, are shown in table I.

```
1  <label>:7   ; preds = %.lr.ph , %17
2    %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %17 ]
3    %sum.02 = phi i32 [ 0, %.lr.ph ], [ %20, %17 ]
4    %8 = and i64 %indvars.iv, 1
5    %9 = icmp eq i64 %8, 0
6    br i1 %9, label %10, label %13
7  <label>:10 ; preds = %7
8    %11 = getelementptr inbounds [1000 x i32]* %data, i64 0, i64 %indvars.iv
9    %12 = trunc i64 %indvars.iv to i32
10   store i32 %12, i32* %11, align 4, !tbaa !1
11   br label %17
12 <label>:13 ; preds = %7
13   %14 = trunc i64 %indvars.iv to i32
14   %15 = shl nsw i32 %14, 1
15   %16 = getelementptr inbounds [1000 x i32]* %data, i64 0, i64 %indvars.iv
16   store i32 %15, i32* %16, align 4, !tbaa !1
17   br label %17
18 <label>:17 ; preds = %13, %10
19   %18 = getelementptr inbounds [1000 x i32]* %data, i64 0, i64 %indvars.iv
20   %19 = load i32* %18, align 4, !tbaa !1
21   %20 = add nsw i32 %19, %sum.02
22   %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
23   %21 = trunc i64 %indvars.iv.next to i32
24   %22 = icmp slt i32 %21, %6
25   br i1 %22, label %7, label %._crit_edge
```

Listing 2: Intermediate code generated for listing 1

Node Type	Node Shape	Border Line		
		Integer	Float	Vector
Compute	Circular	Single	Double	Triple
Load/store	Octagon			
Data	Rectangle			

TABLE I: Node types with their shapes and border lines

2) Edge Insertion: Instructions in the IR code can be dependent on other instructions. Data edges are added between the instruction that generates a result (producer), and the instruction which uses that result (consumer). Dependences between load and store instructions are discovered using the LLVM alias analysis module. In addition control edges are inserted between branch instructions and the instructions that are control-dependent on these branches. In the DFG generated by our tool, true dependencies are denoted by dotted lines, while false dependencies are depicted by dashed lines.

C. Optimizations for CGRAs

In this section we describe two CGRA targeted optimizations: Φ node modification and GEP instruction expansion.

1) Φ Node Modification: LLVM uses a static single assignment (SSA) based intermediate representation. SSA representation requires that each variable is defined before use and assigned exactly only once. Therefore, for existing variables in the original high level code multiple copies are created by divergent control paths. At control merge points, computation may have to choose between values that were computed by different paths leading to the merge point. ϕ nodes represent such merging of values. The phi-nodes cannot be mapped onto the CGRA as there is no corresponding instruction present in hardware for the phi-operator. Therefore, this node has to be modified in the DFG in such a way that all the original data dependencies are preserved. The attribute of the ϕ has been modified as a zero latency no operation node (NOP) which gives output if any of the input is available.

2) GetElementPtr (GEP) Instruction Expansion: LLVM uses `getelementptr` (GEP) instructions to compute the address of a sub element of an aggregated data structure. The GEP instruction can be arbitrarily large with an undefined number of dereferences. However, a corresponding hardware instruction to represent a GEP is nonexistent and therefore it needs to be broken up into a set of known operations. The first argument of `getelementptr` is the data type, while the second argument is the base pointer. All the remaining arguments are indices to be used by the respective dereference operations.

Algorithm 1 shows a pseudo code to expand the GEP instruction by inserting additions and a multiplication for static two dimensional arrays. The function `getAddress` returns a set of source nodes for the base address of the structure to be accessed, while `getOffset` and `getElementIndex` return the nodes computing the offset and the element index respectively. The multiplication calculates the address of the element to be accessed by multiplying the width of the data type with the desired index. The nodes $add2$ and $add1$ add the offset to the base address whose sum is added to the product to obtain the absolute address of the element. The edges are attached to the newly created compute nodes accordingly. Finally the succeeding nodes of the GEP node are reattached to node $add1$, before the GEP node is removed. Figure 3 shows the generated DFG after GEP instructions expansion of listing 2. The shaded nodes are the newly added addition and multiplication nodes.

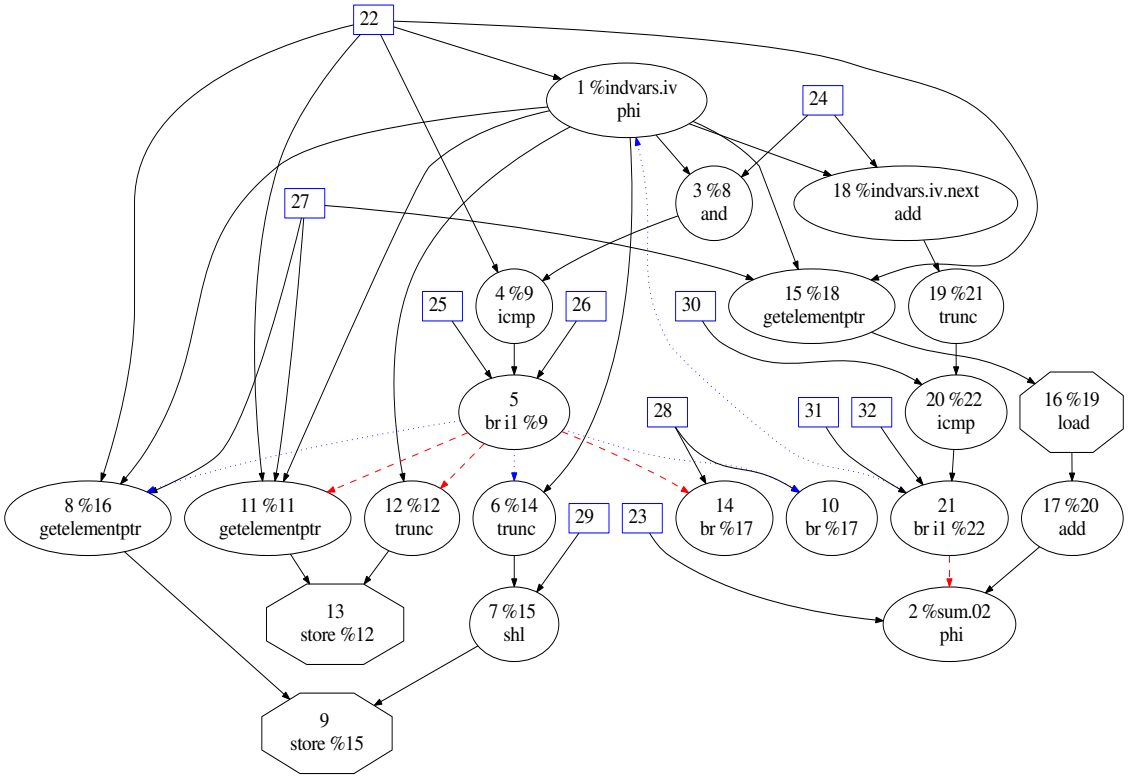

Fig. 2: The DFG generated by the tool for the intermediate code listing 2

D. Print Graph

The output of DFGenTool is in the dot format which is human readable and can be used to process the DFG further to adapt it to the desired architecture.

IV. Results

In this section we present results obtained by generating DFGs for common kernels taken from the Polybench3.2 linear algebra benchmark [18] and a case study of the benchmark kernel *2mm*. Table II shows the statistics in terms of number of inner loops, size of the DFGs in nodes, size of the largest DFG, number of GEP nodes and Φ nodes of the generated DFGs for selected kernels. The source code of each kernel has been compiled by the Clang compiler with O2 optimization level. The table reveals that for these kernels, the number of loops and size of the DFGs are quite large making it tedious to create the corresponding DFGs manually and perform any transformation on them. Therefore by using the our tool, DFGs for a numerous amount of kernels can be generated automatically.

A. Case study: 2mm

In the case study, the generated DFG for kernel *2mm* is discussed in detail. It targets two matrix multiplications, $D = A.B$ and $E = C.D$, in which the second multiplication depends on the result of the first one. However due to clarity we discuss only the first inner loop for the DFG generation further (see listing 3). Figure 4 depicts the resulting DFG after the GEP node has been expanded for the intermediate code in listing 3.

Bench-mark	Inner Loops	Total size of DFGs	Largest DFG	Total GEP nodes	Total Φ nodes
atax	3	71	29	4	6
cholesky	4	92	33	6	6
gemm	5	115	33	6	6
gemver	8	212	39	18	10
trmm	3	81	33	5	4
symm	4	107	33	7	5
3mm	8	182	33	11	11
2mm	7	161	33	9	8

TABLE II: Comparison of generated DFGs for various kernels

As seen from this case study that for this small kernel in only one loop there are two Φ nodes and two `getelementptr` nodes. The complexity of the edges to be handled increases with the number of these nodes. It is tedious to create the modification manually when the number of these nodes are large.

V. Conclusion

In this paper we presented an open source DFG generation tool which converts C code into DFGs automatically. The tool addresses researchers working in the domain of CGRAs, who like to compare inter-CGRA performances of various kernels easily. Due to the presentation of the DFG in the standard, human readable dot format, these dependencies can be addressed further by scheduling and mapping algorithms.

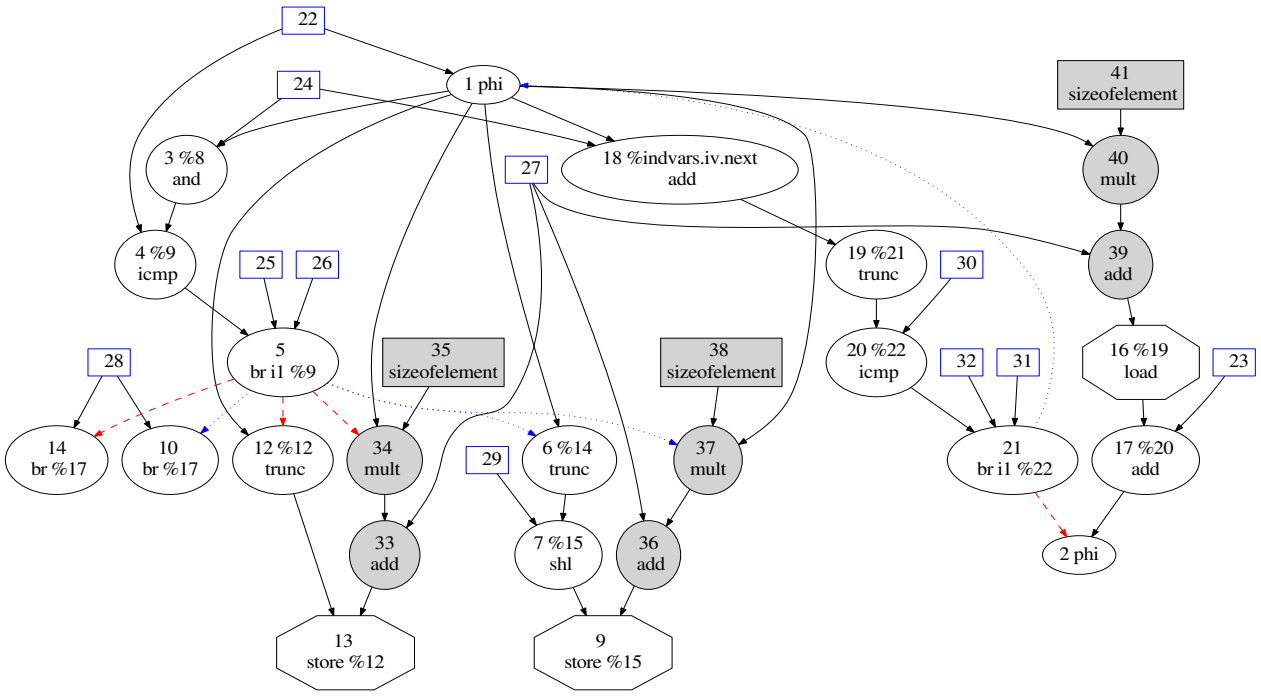

Fig. 3: Final DFG after GEP node expanded

```
; <label>:51; preds = %51, %49
  %52 = phi double [ 0.000000e+00, %49 ], [ %59, %51 ]
  %indvars.iv15.i11 = phi i64 [ 0, %49 ], [ %indvars.iv.next16.i12, %51 ]
  %53 = getelementptr inbounds [1024 x double]*%6, i64 %indvars.iv22.i, i64 %indvars.iv15.i11
  %54 = load double* %53, align 8, !tbaa !1
  %55 = fmul double %54, 3.241200e+04
  %56 = getelementptr inbounds [1024 x double]*%17, i64 %indvars.iv15.i11, i64 % ↩
      indvars.iv18.i10
  %57 = load double* %56, align 8, !tbaa !1
  %58 = fmul double %55, %57
  %59 = fadd double %52, %58
  store double %59, double* %50, align 8, !tbaa !1
  %indvars.iv.next16.i12 = add nuw nsw i64 %indvars.iv15.i11, 1
  %exitcond17.i13 = icmp eq i64 %indvars.iv.next16.i12, 1024
  br i1 %exitcond17.i13, label %60, label %51
```

Listing 3: Generated intermediate code for 2mm.c

Therefore the DFG generation tool not only allows to compare the performance among several CGRA implementations, but also to estimate the impact of various scheduling and mapping methods for a specific CGRA.

References

[1] Boost C++ Libraries. online: http://www.boost.org/.

[2] Graphviz - Graph Visualization Software. online: http://www.graphviz.org/.

[3] Mythri Alle, Keshavan Varadarajan, Alexander Fell, C. Ramesh Reddy, Nimmy Joseph, Saptarsi Das, Prasenjit Biswas, Jugantor Chetia, Adarsh Rao, S. K. Nandy, and Ranjani Narayan. REDEFINE: Runtime Reconfigurable Polymorphic ASIC. *ACM Transactions on Embedded Computing Systems*, 2009.

[4] Péter Arató and Gergely Suba. A data flow graph generation method starting from c description by handling loop nest hierarchy. In *9th IEEE International Symposium on Applied Computational Intelligence and Informatics, SACI 2014, Timisoara, Romania, May 15-17, 2014*, pages 269–274, 2014.

[5] Volker Baumgarten, G. Ehlers, Frank May, Armin Nückel, Martin Vorbach, and Markus Weinhardt. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *The Journal of Supercomputing*, 26(2):167–184, 2003.

[6] Bruno Bodin, Youen Lesparre, Jean-Marc Delosme, and Alix Munier Kordon. Fast and efficient dataflow graph generation. In *17th International Workshop on Software and Compilers for Embedded Systems, SCOPES '14, Sankt Goar, Germany, June 10-11, 2014*, pages 40–49, 2014.

[7] A. Fell, Z.E. Rakossy, and A. Chattopadhyay. Force-directed scheduling for Data Flow Graph mapping on Coarse-Grained Reconfigurable Architectures. In *ReConFigurable Computing and FPGAs (ReConFig),*

Algorithm 1 Algorithm for GEP Node Expansion

for all $node \in DFG$ **do**
  if $node = GEPnode$ **then**
    $P \leftarrow$ Parent($node$)
    $C \leftarrow$ Child($node$)
    addNode($add1$)
    addNode($add2$)
    addNode($mult$)
    addNode($sizeofelement$)
    for all $p \in P$ **do**
      if $p \in$ getAddress($node$) **then**
        addEdge(p, $add1$)
      end if
      if $p \in$ getOffset($node$) **then**
        addEdge(p, $add2$)
      end if
      if $p \in$ getElementIndex($node$) **then**
        addEdge(p, $mult$)
      end if
      addEdge($sizeofelement$, $mult$)
      addEdge($mult$, $add2$)
      addEdge($add2$, $add1$)
      deleteEdge($node$, p)
    end for
    for all $c \in C$ **do**
      addEdge($Add1$, c)
      deleteEdge($node$, c)
    end for
    deleteNode($node$)
  end if
end for

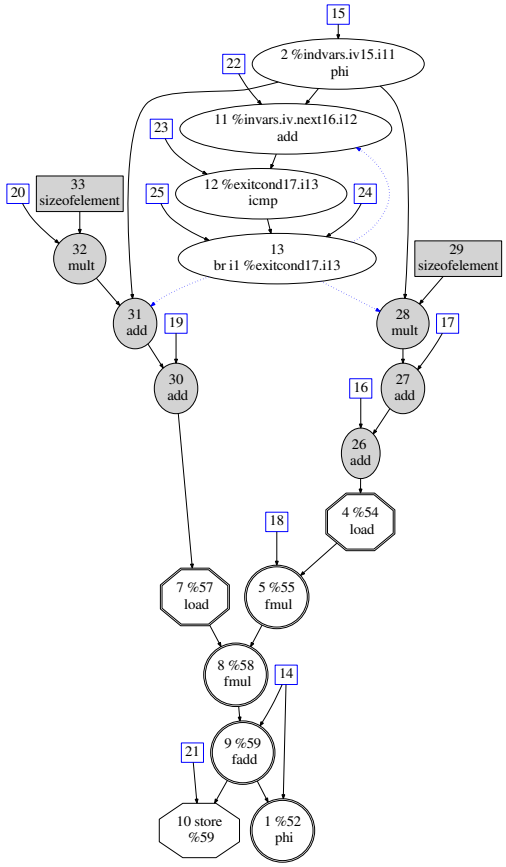

Fig. 4: Final DFG generated after expansion of GEP nodes for the IR code in listing 3

2014 International Conference on, pages 1–8, Dec 2014.

[8] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. SPR: an architecture-adaptive CGRA mapping tool. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 191–200, New York, NY, USA, 2009. ACM.

[9] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. EPIMap: Using Epimorphism to Map Applications on CGRAs. In *Proceedings of the 49th Design Automation Conference (DAC)*, 2012.

[10] Mahdi Hamzeh, Aviral Shrivastava, and Sarma B. K. Vrudhula. REGIMap: Register-Aware Application Mapping on Coarse-Grained Reconfigurable Architectures (CGRAs). In *The 50th Annual Design Automation Conference 2013, DAC '13*. ACM, May 29–June 7, 2013.

[11] R. Hartenstein, M. Herz, Th. Hoffmann, and U. Nageldinger. *Generation of Design Suggestions for Coarse-Grain Reconfigurable Architectures*, pages 389–399. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[12] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[13] F. Li, A. Pop, and A. Cohen. Automatic extraction of coarse-grained data-flow threads from imperative programs. *IEEE Micro*, 32(4):19–31, July 2012.

[14] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, volume 2778 of *Lecture Notes in Computer Science*, pages 61–70. Springer, 2003.

[15] Jason Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179. Citeseer, 2003.

[16] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 36–40, 2014.

[17] Mihail Petrov, Tudor Murgan, Frank May, Martin Vorbach, Peter Zipf, and Manfred Glesner. The XPP Architecture and Its Co-simulation Within the Simulink Environment. In *Field Programmable Logic and Application, 14th International Conference, FPL, Leuven, Belgium*, volume 3203 of *Lecture Notes in Computer Science*, pages 761–770. Springer, August 30 2004.

[18] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. online: http://www.cs.ucla.edu/~pouchet/software/polybench/, 2012.

[19] Zoltán Endre Rákossy, Tejas Naphade, and Anupam Chattopadhyay. Design and Analysis of Layered Coarse-Grained Reconfigurable Architecture. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 2012.

[20] Zoltán Endre Rákossy, Dominik Stengele, Axel Acosta-Aponte, Saumitra Chafekar, Paolo Bientinesi, and Anupam Chattopadhyay. Scalable and Efficient Linear Algebra Kernel Mapping for Low Energy Consumption on the Layers CGRA. *11th International Symposium on Applied Reconfigurable Computing, Bochum, Germany*, April 2015.