Towards a Java bytecodes compiler for Nios II soft-core processor

Willian S. Lima[1], Renata S. Lobato[1], Aleardo Manacero Jr.[1] and Roberta Spolon[2]
[1]*DCCE, UNESP – São Paulo State University – Brazil*
[2]*DC, UNESP – São Paulo State University – Brazil*
willian.lima@sjrp.unesp.br, renata@ibilce.unesp.br, aleardo@ibilce.unesp.br and roberta@fc.unesp.br

Abstract

Reconfigurable computing is one of the most recent research topics in computer science. The Altera™ Nios II soft-core processor can be included in a large set of reconfigurable architectures, especially because it is designed in software, allowing it to be configured according to the application. The recent growth in applications that demand reconfigurable computing made necessary the building of compilers that translate high level languages source codes into reconfigurable devices instruction sets. In this paper we present a compiler that takes as input the bytecodes generated by a Java front-end compiler and generates a set of instructions that attends to the Nios II processor instruction set rules. Our work shows how we process Java bytecodes to the intermediate code, in the Nios II instructions format, and build the control flow and the control dependence graphs.

1. Introduction

Reconfigurable computing is an important current research topic in computer sciences [7] [10] [11]. Among its many applications we may cite the development of embedded systems, such as cell phones [14].

We say a device is reconfigurable (reconfigware) when it is possible to change its configurations to customize the execution of an application [8]. In other words, reconfigurable computing systems allow the execution of instructions in hardware and permit changes to these instructions via software.

Due to the growing need to build applications for reconfigwares, the development of a compiler that is able to translate programs written in a well-known language, such as Java, to the instruction set of a device that may be configured by the programmer, such as Altera™ Nios II processor, became a necessity in reconfigurable computing. Adopting Java as the primary high-level language is a good choice, as it is free and one of the most used programming languages at the moment; furthermore, its architecture allows us to use analysis phases from its front-end compiler.

In our work, we present a new compiler, called JaNi (*Java Compiler for Nios II*) that takes as input the intermediate representation of a source code compiled by a Java front-end compiler (bytecodes) and generates an ordered set of Nios II instructions. Our compilation process includes bytecodes conversion into compiler intermediate code, control graphs creation and target code generation. Boards' configuration is not part of this version of JaNi, as is postponed to a later stage, after compilation.

An overview of Java compilation process is presented in section 2, along with some important points to be considered about Nios II instruction set. In section 3 we describe our compiler structure and phases. We focus on control graphs creation techniques in section 4 and bytecodes processing and code generation in section 5. Section 6 reports tests conducted on JaNi and in section 7 there are conclusions concerning the presented work, as well as future work issues.

2. Overview of Java language and Nios II instruction set

The next subsections show how Java compilation and the bytecodes generation works; furthermore, the Nios II instruction set properties.

2.1. Java bytecodes

We can say a code compilation in Java is performed in two stages. In the first one, the source

978-1-4244-4671-1/09/$25.00 ©2009 IEEE

code is literally compiled into an intermediate representation in form of bytecodes. In the second, a Java Virtual Machine (JVM), according to the target architecture and operating system, interprets this intermediate code [13].

Bytecodes are instructions, 1-byte large, that may or may not have parameters, which are also specified byte after byte. The bytecodes are prearranged in stack form, where operations are performed only over the top operands [12].

2.2. Nios II instructions

Altera™ Nios II processor is a soft-core processor, which differs from common processors especially because it is designed in software. This permits flexibility for hardware designers while software designers are not affected by device implementation details [2].

Nios II instructions are all 4-byte words and are organized by their operand types into three groups: I-Type, for immediate values, R-Type, for register-stored values, and J-Type for an address value (sub-routine calls) [1].

3. JaNi conception

The JaNi project aims at building a compiler able to generate code in the format of ordered instructions for the Altera™ Nios II soft-core processor [4] [5]. Besides this, we were also concerned about code optimization and code parallelization and, in a further step, providing means to implement them.

Figure 1 depicts the structure of our compiler. The elements that compose the JaNi layers are described as follows:

- .class file: a file that contains the intermediate representation, or bytecodes, from a class specification in Java.
- Bytecodes Interpreter (BI): bytecodes are processed in this layer, as detailed in section 5. The BI processes the bytecodes and generates the compiler intermediate code.
- Flow Analyzer (FA): responsible for developing both control and data flow analysis.
- Dependence Analyzer (DA): performs both control and data dependence analysis.
- Code generator (CG): creates the words in format to be executed according to Nios II processor and, before this, optimizations at intermediate code level.
- Executable: binary file with Nios II instruction words organized according to the compiled source code.

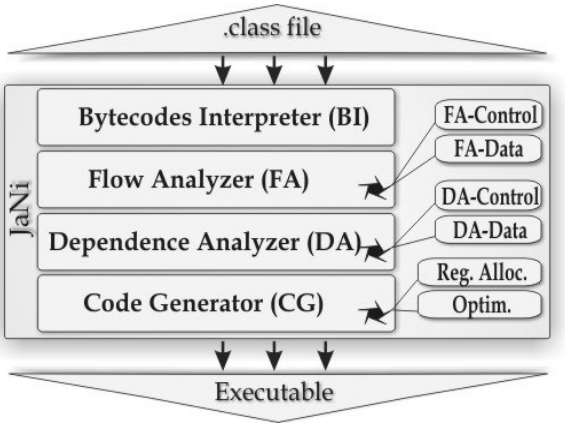

Figure 1. JaNi conceptual scheme.

The intermediate code used from BI to CG follows the Nios II instruction set pattern, differing only on how they are internally stored. The only translations required by CG are related to instruction storage and register allocation, as the final code is in 32-bit words format having the addresses of Nios II registers.

In Section 5 we will analyze each JaNi's layer role in a source code compilation.

4. Control graphs

The use of control graphs in a compiler is important as they allow studies of program instructions execution paths, enabling optimizations on implementation and, indeed, vectorization or parallelization [6].

The next subsections show how we implemented both control flow and control dependence graphs.

4.1. Control Flow Graph (CFG)

A CFG is responsible for storing information about program control flow, which means that this kind of structure keeps track of possible execution paths. CFG is used as the basic structure in a compiling process, especially because it supplies important information to the following compilation phases.

Implementation of a CFG may vary about the level of information a node must contain. A node can represent either an intermediate instruction, or a basic

block[1]. Edges are directed and represent a control flow between two nodes in the CFG, which means that the program control can go from the former to the latter.

Our approach to build the CFG is presented by Figure 2. Nodes representing basic blocks are found traversing the instructions, looking for branches or targets from branches. The code sequences that form the basic blocks are inserted afterwards.

1.	Find all basic blocks in the intermediate code, considering branch instructions and instructions that are the target of branches.
2.	For each basic block:
2.1.	Inspect the last instruction to track where the control can flow after its execution.
2.2.	Create an edge from the basic block analyzed to the basic(s) block(s) that could pass a program control.

Figure 2. Algorithm for CFG building.

4.2. Control Dependence Graph (CDG)

The CFG construction enables CDG construction since relations of control dependence are strongly associated to control flow [3]. Control dependence graphs are useful for control dependence analysis, which allows stronger program optimizations and instruction scheduling.

We use standard control dependence definition [6], which states that an instruction Ψ is control-dependent of another instruction Φ ($\Psi \neq \Phi$) if and only if we can reach the end node T of the control flow graph from Ψ passing through Φ. In other words, this means that we cannot reach the end node of the control flow graph from Ψ without executing instruction Φ.

To use the previous definition, it is necessary to have only one end node in the CFG. This is achieved by adding a "sink" instruction that will be the target for any other instruction that causes program termination [3].

Considering the fact that when a basic block is reached during a program execution all of its instructions are executed, we shaped our CDG to have basic blocks as nodes. As a consequence, this will produce a small graph structure.

In Figure 3 we present our DFS-based algorithm to generate the CDG.

[1] A sequence of instructions in which once the first instruction of the set is executed, then every instruction in the sequence will also be.

1.	Take CFG and copy all nodes to CDG.
2.	For each two nodes at CFG:
2.1.	If they are control-dependent:
2.1.1.	Create an edge connecting these two nodes at CDG.

Figure 3. Algorithm for CDG building.

5. The compilation process

In this section we explain how JaNi actually works. Figure 4 summarizes the compilation process, and the following subsections provide details of each step.

5.1. Bytecodes processing

The first task to be done is to translate the bytecodes being compiled into JaNi intermediate representation (step 1 in Figure 4). The Bytecodes Interpreter is responsible for that.

The BI traverses all bytecodes simulating their execution but, instead of producing the results, it generates instructions during bytecodes interpretation. Thus, we first transform the stack top operands [13] into variables and then choose the most suitable instructions that apply.

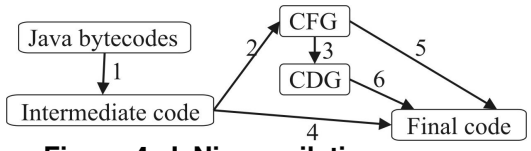

Figure 4. JaNi compilation process.

As an example, consider an addition operation. In bytecodes representation, the two operands must be at the operand stack top to be popped out, and then the result of this operation is pushed onto the operand stack. These two values could have been obtained by a load operation or they could be constants. In this situation, the BI identifies the origin of the operand and assembles the correlated intermediate instruction taking into account this information.

JaNi currently supports only a relevant subset of Java bytecodes. The supported bytecodes are the ones capable of implementing the following high-level instructions (commands) in Java:

- Selection: if, if-else;
- Loop: while, for, do-while, break, continue;
- Operators: +, -, *, /, %, ++, --, <<, >>, ||, &&, |, &, ^, ~, <, >, <=, >=, ==, !=;
- Data types: boolean, byte, short, int, char;

5.2. Graphs generation

With the intermediate code, it is possible to create the control graphs through steps 2 and 3 in Figure 4. Both graphs are represented at basic block level, and the basic blocks finding stage is common to all. The CFG is generated by FA module while the DA generates CDG, as explained in section 4.

5.3. Target code generation

As mentioned earlier, JaNi intermediate code meets the requirements of Nios II instruction set. Therefore, the CG can build a binary file through a one-to-one association with intermediate instructions relationship with intermediate instructions (step 4 in Figure 4), without extra effort.

Steps 5 and 6 are included in order to provide intermediate code optimizations and a register allocation policy. This task becomes feasible by the use of the control graphs (CFG and CDG).

The binary file contains a list of Nios II instructions in 4-byte words with little-endian representation.

6. Case studies

In this Section we present and discuss the compilation of two different examples by JaNi.

In the first test, the Java source code in Figure 5a is compiled, and the associated bytecodes can be viewed in Figure 5b. For the sake of brevity, parameters for bytecodes that have parameters (the ones in bold in Figure 5b) are not shown.

It is important to note that it was necessary to perform a transformation from a code specified in operands stack form to another one in instructions sequence. To do this we create movement instructions, where operands from stack become values for auxiliary variables at the intermediate code.

The generated intermediate code has several *add* and *addi* instructions. This happens because in Nios II instruction set the movement instructions are not implemented. They are pseudo-instructions [1], and they can be performed by addition operations where one of the parameters has a zero value. This parameter is represented by register zero, which is a read-only register that always contains value zero.

Figure 6 shows the CFG and the CDG for the compiled code. In both sides of the figure, the shaded node represents the "sink" basic block of each graph.

By looking at Figure 6a, it is possible to notice that we have two loops: B1, B2, B1 and B4, B5, B4. Formally, we can find loops in a CFG by identifying back-edges inside the graph, such as edges B2 – B1 and B5 – B4.

In addition, just as with the information provided by CFG, we can infer that blocks B1 and B4 are loop control headers, because they decide whether the program control enters the loop path or the other path.

In Figure 6b we have the CDG for the CFG in Figure 6a. Its edges are directed from node X to node Y if X is control-dependent of Y. As an example, let us take into account the node referring to basic block B2. By analyzing the CFG we know that it is impossible for B2 to reach B6 without passing through B1, thus B2 is control-dependent of B1. However, B1 can reach B6 by passing through nodes B3 and B4, making it control-dependent of B3 and B4. As B2 is control-dependent of B1, it will also be control-dependent of B3 and B4. Moreover, node B4 is not control-dependent of any other node, since the program control can flow directly from B4 to B6, and no instruction is control-dependent of B6, because B6 is the end node of CFG.

```
int fib = 1, fat = 1;
int n = 12;

int fAux1 = 1, fAux2 = 1;
for ( int i = 2; i <= n; i++ ){
    fib = fAux1 + fAux2;
    fAux1 = fAux2;
    fAux2 = fib;
}

int i = n + 1;
while ( --i >= 2 )
    fat *= i;
```

(a)

```
ICONST_1        ILOAD_3         ICONST_1
ISTORE_1        ILOAD_3         IADD
ICONST_1        IF_ICMPGT       ISTORE
ISTORE_2        ILOAD           IINC
BIPUSH          ILOAD           ILOAD
ISTORE_3        IADD            ICONST_2
ICONST_1        ISTORE_1        IF_ICMPLT
ISTORE          ILOAD           ILOAD_2
ICONST_1        ISTORE          ILOAD
ISTORE          ILOAD_1         IMUL
ICONST_2        ISTORE          ISTORE_2
ISTORE          IINC            GOTO
ILOAD           GOTO            RETURN
```

(b)

Figure 5. Source code (a) in Java and (b) its bytecodes.

Hence, for the first test, we could achieve the construction of intermediate graphs that permit obtaining useful information from the code (in

bytecodes form) that is being compiled. It also showed that the CFG mapped the program compiled.

Our second test aims at building a more complex graph structure. The Java source code is given in Figure 7, and its respective graphs in Figure 8. The end nodes of each graph are also shaded.

As previously said, a loop can be identified at CFG of Figure 8a by looking at the unique back-edge. The presence of forward-edges allows us to infer the existence of selection instructions. Differently from CFG in the first test, this CFG enables a number of distinct paths to get to the end node for some nodes.

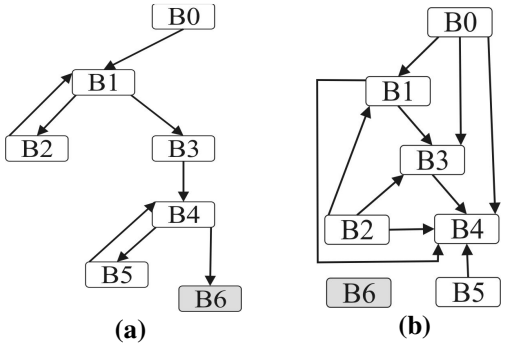

Figure 6. (a) CFG and (b) CDG for the compiled source on test 1.

The Java code presented in Figure 7 has a repetition structure, which includes two simple selection structures, and still has a compound selection instruction after the loop. Variables of different types are used and diverse operators are introduced as well.

```
boolean TRUE = true, FALSE = false;
short s = -14, r = 21;
char letter;
int n = 4, n = 2008;

int i = n;
do{
   if ( s < r && TRUE && i < m/256 ){
      i += 2;
      continue;
   }
   i *= 2;
   if ( !FALSE || FALSE ) n = n << 1;
}while( i < n2 );

n = n & 1;
if ( n == 0 ) letter = 'P';
else letter = 'I';
```

Figure 7. Java code for test 2.

With CFG, we see that all nodes executed before node B9[2] are control-dependent of B9, because all paths to reach B12 pass through B9. In this case, it is true to say that B9 is a digraph articulation point. At a lower level, if the program control was previously on some instruction of nodes B0..B8, then it is necessary that all instructions in basic block B9 be performed in order to reach the end node.

Like the first test, the last one provided results showing how JaNi performs the classic intermediate graphs generation.

For both tests, the binary file was generated successfully, as did our register allocation policy, based on Chaittin-Briggs algorithm [9].

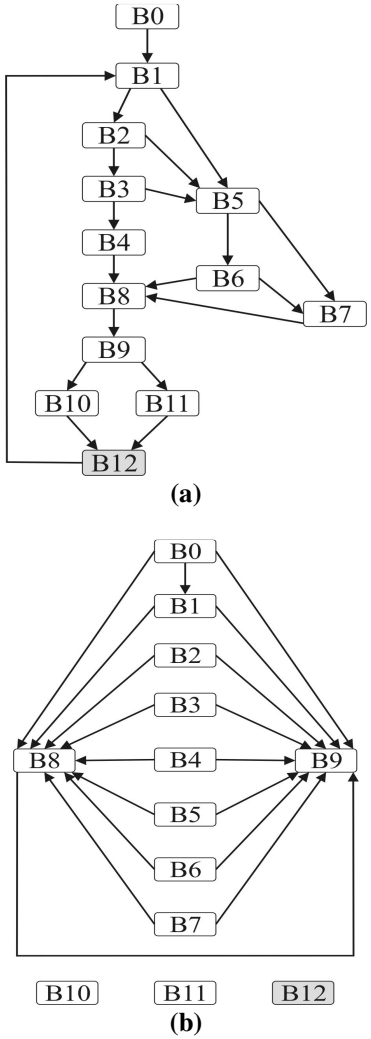

Figure 8. (a) CFG and (b) CDG for test 2.

The binary files generated by JaNi were tested on a Nios II simulator [15] and produced the expected results. All registers had the correct values for the variables that were mapped to them, during the whole simulation.

[2] The statements related to node B9 in this paragraph are also valid for node B8, in the second test.

7. Conclusions and future work

In this work we introduced a new compiler called JaNi. JaNi aims to be a full compiler for the Nios II soft-core processor, which can be configured into a *reconfigware*. We have shown how JaNi handles Java bytecodes processing and controls graphs generation.

Tests conducted on JaNi showed that the compiler successfully compiles bytecodes generated from source code comprised by statements present on the defined subset of Java language, being able to separate basic blocks and correctly create both CFG and CDG. It was also possible to guarantee that our strategy for bytecodes translation works for the tested Java subset.

There are still some features that must be added to JaNi. First of all, data graphs generation must be included: DFG (Data Flow Graph) and DDG (Data Dependence Graph) [6]. The addition of program data graphs enables data analysis, which is responsible for a large variety of optimizations, including copy propagation for reducing the size of the intermediate code. Moreover, data analysis also allows vectorization or parallelization to be implemented.

Thus, JaNi makes possible for a program written in Java to be run over an Altera™ Nios II soft-core processor, contributing to reconfigurable computing. JaNi can be found at <http://www.dcce.ibilce.unesp.br/spd/english/index.html>.

8. Acknowledgments

The authors would like to thank Brazilian research funding agencies CNPq (process n. 134450/2008-6), FAPESP and Fundunesp.

9. References

[1] Altera *Nios II Custom Instruction - User Guide*, http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf. 2005.

[2] Altera *Literature: Nios II Processor*, http://www.altera.com/literature/lit-nio2.jsp. 2007.

[3] Amtoft, T., Banerjee, A., Hatcliff, J. and Ranganath, V. P. A New Foundation for Control Dependence and Slicing for Modern Program Structures, *ACM Trans. on Programming Languages and Systems*, 29, 5 (2007), 1-43.

[4] Cardoso, J. M. P. and Neto, H. C. Towards an Automatic Path from Java™ Bytecodes to Hardware Through High-Level Synthesis, *In Proc. of 5TH IEEE Intl. Conf. On Electronics, Circuits And Systems (ICECS-98)* (Lisboa, Portugal, 1998).

[5] Cardoso, J. M. P. and Neto, H. C. Macro-based hardware compilation of Java™ bytecodes into a dynamic reconfigurable computing system, *In Proc. of IEEE Symp. on FPGAs for Custom Computing Machines* (United States, 1999).

[6] Chapman, B. and Zima, H. *Supercompilers for Parallel and Vector Computers*, ACM Press, 1990.

[7] Cheung, P. Y. K., Constantinides, G. A., Luk W., Mencer, O., Todman, T. J. and Wilton, S. J. E. Reconfigurable Computing: architectures and design methods, *IEE Proc. - Computers And Digital Techniques*, 152, 2 (2005), 193-207.

[8] Compton, K. and Hauck, S. Reconfigurable Computing: A Survey of Systems and Software, *ACM Computing Surveys*, 34, 2 (2002), 171-210.

[9] Cooper, K. D., Dasgupta, A. and Eckhardt, J. Revisiting Graph Coloring Register Allocation: A Study of Chaitin-Briggs and Callahan-Koblenz Algorithms, *In Proc. of Workshop on Languages and Compilers for Parallel Computing* (2005).

[10] El-Ghazawi, T. and Saha, P. Extending Embedded Computing Scheduling Algorithms for Reconfigurable Computing Systems, *In Proc. of 3rd Southern Conf. on Programmable Logic* (Argentina, 2007).

[11] El-Araby, E., El-Ghazawi, T. and Nosum, P. Productivity of High-level Languages on Reconfigurable Computers: A HPC Perspective, *In Proc. of Intl, Conf. on Field-Programmable Technology* (Japan, 2007).

[12] Henessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, Morgan Kauffman Publishers, 2002.

[13] Lindholm, T. and Yellin, F. *The Java™ Virtual Machine Specification - Second Edition*, http://java.sun.com/docs/books/jvms/. 1999.

[14] Mange, D., Mudry, P., Tempesti, G. and Vannel F. CONFETTI: A reconfigurable hardware platform for prototyping cellular architectures, *In Proc. of IEEE Intl. Parallel and Distributed Processing Symp.* (United States, 2007).

[15] Lima, W. S., Lobato, R. S., Silva, A. C. F., Ulson, R. S. Simulação de execução de instruções do processador de núcleo virtual Nios II, *In Proc. of Conf. Latinoamericana de Informática* (Santa Fé, Argentina, September 8-12, 2008).