2003－HPC－94 (2)
2003／6／13

An Ambiguous, Context-Free Grammar for Deterministic Parsing In Queue-Java Compiler

li.qiang.wang ,† ben.a.abderazek ,† soichi shigeta ,†
tsutomu yoshinaga † and masahiro sowa †

In this paper, we propose an effective parsing method to generate Queue Abstract Syntax Tree (QAST) that is used for an optimized QJVM instruction generation in a Queue Java compiler (QJAVAC).With the QJAVAC, that using this parsing approach and embedding QAST, we have successfully compiled the Java source code to the QJVM byte code.We describe the QAST parsing algorithm implementation and evaluation.

1. Introduction

The definition of the Java programming language includes both the definition of the programming language itself and the definition of the virtual machine which can run compiled Java applications represented in the form of bytecodes. An application in the Bytecode form can run on any computer with an implementation of the Java virtual machine. A Bytecode for of an application is distributed in class files which in addition to the bytecodes contain information required for linking and verification.

On most computers, people want an efficient execution of Java Bytecodes. But, execution of typical Java Bytecode, which is stack-based, with interpretation, with a JIT compiler, or directly with a special java processor is invariably constrained by the limitations of the stack architecture for accessing operands. Thus, the conventional stack based applications cannot take advantage of a pipelined arithmetic logical unit (ALU), since the result of one operation must be returned to the top of the stack before it becomes the operand of the next operation. Hence, the lack of instruction level parallelism (ILP) in Java Bytecode streams is the main cornerstone of Java applications.

Several software or/and hardware techniques were proposed to cope with the lack of instruction level parallelism within a Java bytes code streams. When interpreted, the Java bytes code streams have been seen to be 30 times slower than optimized C Code, whereas JIT compilers can provide up to 20 times speedup with respect to interpreted code. However, the memory requirement of JIT compilers is extremely expensive for pervasive applications.

In order to easily and efficiently exploit ILP without the need of intelligent static scheduling of byte code streams and sophisticated hardware support, we proposed a Queue-Java system that compiles Java source code into Queue-Java Bytecode(QJBT). A new Java execution mode named queue-Java based execution model (QJVM) in JVM has been developed.

Because the QJBT sequence is very different from Java Bytecode sequence, the previously-known variants of parsing approaches can not fit the requirement of QJBT. In this paper, we will describe a new Type of syntax tree-queue abstract syntax tree (QAST) and its generation paring approach from Java language. With our approach we can keep ambiguous, context-free grammar of Java. The paring method is implemented for a Queue Java compiler (QJAVAC).

The focus in this paper will be on instruction paring methods of QAST and its sub-QAST in case of grammar items in Java language. The design of QJAVAC was driven by a detailed examination of the use of customizable protocols and algorithms. We delineate the issues that must be considered when targeting fast and effective product queue java instruction in QJAVAC. Base on it, we had successfully realized the QJAVAC with QAST and suitable QAST paring methods.

The rest of this paper is organized as follow: An overview of the QJAVAC compiler follows in section 2. In section 3, we describe the the paring of QAST. Section 4 gives the evaluation results of this approach. In the last section, we give the conclusion and our future work.

† Graduate School of Information Systems, The University of Electro-Communications

2. QJAVAC Compiler Overview

2.1 QJAVAC Implementation Overview

As was mentioned earlier, the queue-based execution model (QEM) performs most operations on a FIFO data structure. However, the stack-based execution model (SEM) performs most operations on a first-in-last-out (FILO). The QEM is analogous to the usual SEM. The QEM has operations in its instructions set which implicitly reference to an operand Queue (OPQ), just as SEM has operations, which implicitly reference an operand stack. Each instruction removes the required number of operands from the head of the OPQ, performs some computations, and stores the results of the computations at the tail of the OPQ, which occupies continuous storage locations (described later). That is, the execution order of instructions coincides with order of the data in the OPQ.

The QJAVA is a high instruction level parallelism (ILP) execution environment based on QEM and Java Platform. It consists of basically of a language and virtual machine components.

In our system, the syntax and semantics of Queue-Java Language is consistent with the conventional Java Language Specification released by Sun Microsystems, to protect Java Easy-Using and Object Oriented features. The difference is found in the Virtual Machine environment itself.

The high performance of QJAVA comes from its Queue Java Virtual Machine (QJVM) which uses QEM. However the conventional Java Virtual Machine uses SEM.

The QJAVAC is a complier, which has a similar implementation like other multiple phases Java Compilers. The QJAVAC overview is shown in Figure.1. In order to parsing and representing the grammars items of Java Language for the novel QJAVA architecture, a "reparsing" stage is added into the front end of JAVAC. The "reparsing" stage processes the grammar representations from input and reparse it to grammar representations for QJAVA.

2.2 Intermediate Representation of Java In QJAVAC

The intermediate representation (IR) is a representation collection for every Java class defined in the input. Every class node contains references to nodes for all interfaces, fields, methods and statement and expression defined

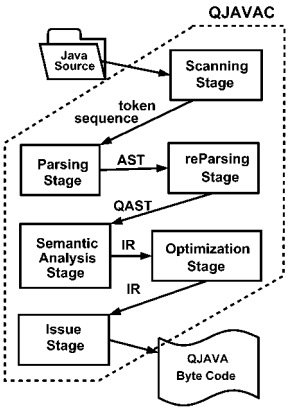

Fig. 1 QJAVAC Compiler Phases

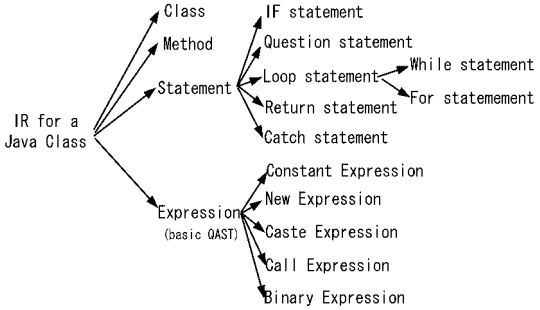

Fig. 2 IR and QAST from Java class

for that class as well as some other information (access flags, a reference to the super class etc.). QAST is contained in the intermediate representation. An overview of IR is presented in this section.

Figure.2 shows a class hierarchy for a subset of the classes used in IR. QJAVAC used for the experiments presented in this paper is implemented in C++ and was successfully integrated with IR which includes QAST, We will use the following, very simple example to illustrate how IR represents Java class.

```
class simple{
//Class IR
    int i=0;
    //Constant Expression IR
    public simple() {
    //Method IR
        int a = 0;
        // Binary Expression IR
        super()
        //Call Expression IR
    }
}
```

A complete IR representation of class simple takes care of many details that are required to

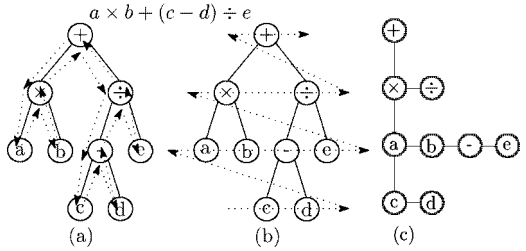

Fig. 3 Post, Level Traversal Order and QAST

maintain the meaning of the program. The image gets even more complicated by the fact that some information is replicated so that it can be accessed quickly.

The IR representation is exactly the same as the original source. And also, IR corresponds exactly to the code generated by the compiler. In our example, the code for the constructor contains one class IR, one method IR and three expressions IRs.

All implicit class, method and expressions are represented explicitly in IR. A class is represented with an object of a class IR, class IR which contains lists of objects representing fields, methods and interfaces. A method is represented with an object of the method IR. A statement may contain other statements or expressions. Expression IR is a QAST with nodes for every expression in a Java Class All nodes in the syntax tree are objects of some class derived from the class Expression IR.

2.3 The QAST In IR

The QAST is a new type of syntax tree that is optimum for QJAVAC compiler. To compare the QAST with the abstract syntax tree (AST), we will use a simple example shown in Figure 3. In Figure 3(a), the instruction generation for SEM is obtained by traversing the tree in post order travel. From above traversal, it is very clear to notice that ASTs provides enough information for the SEM's compiler by connection between two nodes in AST. With the "connection" lines, the compiler can easily jump to the children and back to parent to produce instructions in post order manner. Note that from some viewpoints, we can call ASTs as a stack syntax tree.

However, this traversal will not work when dealing with queue execution model. Therefore, to get a correct instruction generation sequences for the QJAVAC, the traversal of the above tree is changed as illustrated in Figure 3(b). We call this traversal a level order traversal. So, in order to find the deepest and shallowest nodes, the compiler must traverse the entire tree at first, remember the position of the nodes, and then load the tree again to emit instructions. However, we have to note that, since there is not relation between the same-level of nodes, finding the same nodes in instructions generation stage is very difficult and its algorithm becomes very complex, huge, and "time hungry".

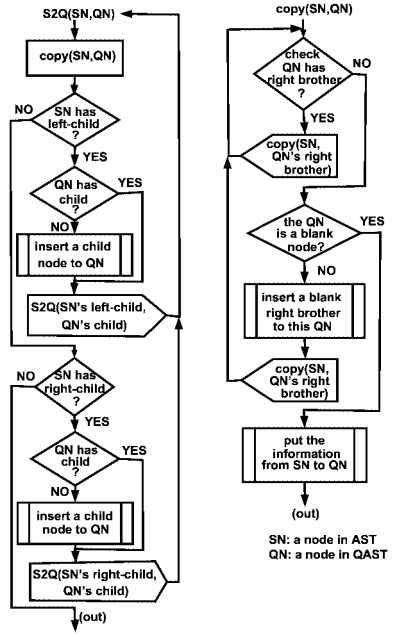

Fig. 4 Work Flow Chart of parsing Binary Expression QAST

In order to cope with this problem and reduce the instruction generation time, we propose another traversal algorithm. We call it Queue Abstract Syntax Tree (QAST). The QAST for a simple expression is shown in Figure 3(c). The right hand part of the above figure is the translated graph. In the above graph, we have to note that the connect path (Line) only appears between two nodes. In addition, the connect information does not only appear between parent and child nodes; it also appears between "brother" and "brother" nodes.

3. Grammar Items Parsing from AST to QAST

3.1 Grammar Item of Parsing Binary Expression QAST

The Binary Expression grammar items have a common character-when they act as a node in AST, they must be a parent-node and have two

children and those nodes only emit one Java Instruction. Parsing the binary AST to QAST is as follows:

(1) The Relation between parent and right-child is removed.

(2) The Relation between parent and left-child may be removed or preserved: if it is not in most-left side of stack syntax tree, it will be cut off, if it is just there, it will not be cut.

(3) A new Relation is inserted between two children nodes-because they are in same-level.

With this algorithm, a Binary AST is parsed to a Binary QAST. Then, by using the Instruction Emitting Algorithm-we had talked about, we can very easily get a QEM instruction sequence.

3.2 Grammar Item of Parsing Comparison QAST

The comparison expression QAST grammar items (<, <=, ==, >, >=) are parsed by following next rules:

- Value produced by the < is true if the value of the left-hand operand is less than the value of the right-hand operand, and otherwise is false.
- Value produced by the <= operator is true if the value of the left-hand operand is less than or equal to the value of the right-hand operand, and otherwise is false.
- Value produced by the == operator is true if the value of the left-hand operand is equal to the value of the right-hand operand, and otherwise is false.
- Value produced by the > operator is true if the value of the left-hand operand is greater than the value of the right-hand operand, and otherwise is false.
- Value produced by the >= operator is true if the value of the left-hand operand is greater than or equal to the value of the right-hand operand, and otherwise is false.

Here, the false means 0; the true means 1.

The comparison QAST sometimes is very similar to a binary item; its parents have two children. For integer numeric comparison, there are several instructions for different comparison case in QJVM, so it is very easy. But for long and double numeric comparison, there is one comparison instruction, so one time comparison operation has those steps:

(1) Get all of three comparison -status of two numbers (less -1, equal- 0, greater 1)

(2) According to the operator type, load 1 or 0 to queue

(3) Compare the result we had got and the 1(or 0) that we load in Setp2, according the operator type, load the correct value to queue.

With this, the Long type numeric comparison becomes a common integer type numeric comparison.

3.3 Grammar Item of Parsing Question Statement

Question statement expression uses the Boolean value of one expression to decide which of two other expressions should be evaluated.

ExpressionIR?ExpressionIR : ExpressionIR

The question statement has three operand expressions: the "?" appears between the first and second expressions, and ":" appears between the second and third expressions. The first expression must be the type of Boolean.

At run time, the first operand expression of the conditional expression is evaluated first; its Boolean value is then used to choose either the second or the third operand expression:

- If the value of the first operand is true, then the second operand expression is chosen.
- If the value of the first operand is false, then the third operand expression is chosen

For example, expression f(a)?f(b) : f(c) + f(d) + f(e), if f(a) is greater than 0 then the value of expression is f(b) + f(b) + f(e),else the value is f(c) + f(d) + f(e), its AST is shown in figure 5(a), In SEM, because the result of the left part of the question-mark is pushed to the stack, and the value of select-controller is also saved on the top of stack, the branch-direction can be decided at once.

The parsing of the Question Statement AST to QAST normally will cause failure of evaluating the expression, because the results of the left part of "judgment" result is lined in the front of it, the select-controller can get value to decide which direction should jump to. So we choose a speculation model to deal this case: three expressions are evaluated in parallel, at last, when the three results come the head of queue together, we can select one correct result, if the first value in the queue-head is true, it means the first expression's result is true, and we should select the second expression's result as the correct result, the selection step as follow:

(1) Duplicate one time ,copy the correct re-

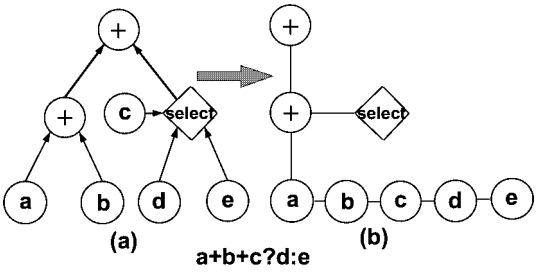

Fig. 5 Parsing Question statement from AST to QAST

sult to the tail of the queue

(2) Pop two times, to discard the correct(it had been saved) and incorrect result if the first value in the queue-head is false, it means the first expression's result is false, and we should select the third expression's result as the correct result, the selection step as follow:

  (a) Pop one time, to discard the incorrect(the result of second expression)

  (b) Duplicate one time ,copy the correct result(the result of third expression) to the tail of the queue

  (c) Pop one time again, to discard the correct(it had been saved)

The Algorithm of this conversion is that:

(1) Get a stack syntax tree's node and a queue syntax tree's node

(2) Check whether the stack syntax's node is question mark operator

(3) If it is,

  (a) Get the child-level node of current queue syntax's node

  (b) Append the first expression into the most right end of the child node of queue node

  (c) Append the second expression into the most right end of the child node of queue node

  (d) Append the third expression into the most right end of the child node of queue node

  (e) Append a select node to current queue syntax's node as its right-brother.

Although the parallelism is very high in speculation model, it can cause no useful computing, for example, because we only use one of results of f(d) and f(e), so the part of calculation must be no useful computing. But, after all we can execute the question mark correctly.

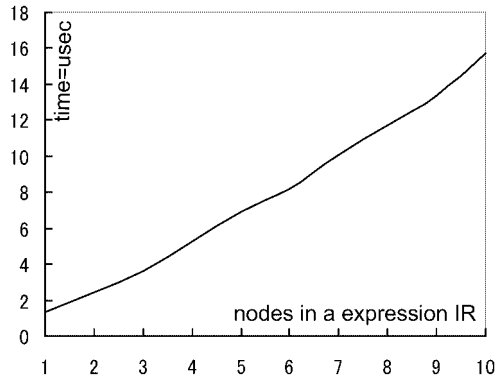

Fig. 6 Parsing QAST Experiment for a Expression IR

4. Evaluations Results and Discussions

We have developed the QJAVAC compiler with ANSI/ISO C++ language. The above compiler was successfully ported to Windows (with Visual C++6.0), Red Hat Linux 7.1J and SunOS 5.6 (with GNU C++ Compiler 2.7).

4.1 QAST Parsing Speed

We have experimented the QAST parsing speed from AST over a number of nodes. The above experiment is shown in Figure 6. From this experiment we conclude that: with the increasing of the nodes in a AST expression, we need more time to pasring the QAST, and this increasing is not a proportional one. For example, for a AST expression with 10 nodes, we need 15.7μsec to parsing it from AST to QAST.

4.2 Compilation Speed and Space

Unfortunately the compiling speed of the QJAVAC compiler was found to be slower than the well-known JAVAC compiler. This speed degradation comes from the fact that a repasing stage to parsing an abstract syntax tree to the queue abstract syntax tree is added. That is in order to get a correct instruction sequence we must "re-parse" the AST to QAST. This extra-phase consumed nearly 30% of the total compilation time. However, the above overhead maybe reduced if there is a parsing algorithm that can directly generate the QAST from token sequences that produces from scanning stage.

The QJAVAC space is the compiler source size, compiler binary size and the memory requirement. The QJAVAC compiler source size

Table 1 QJAVAC Compiler Size

Compiler components	Function source size (KB)	Head source size (KB)
Lexical scanner	709	339
AST parsing	219	294
QST reparsing	291	30
Code generator	789	30
Others	822	123
Total	2897	789

is shown in Table1. It is classified into translation category and instruction generation category. They are about 1140KB (1080KB source size and 60KB head define source size). The compiler binary size is 2401KB compiled by Visual C++6.0 under Windows2000. From the test benchmarks, the running peak memory requirement is also found to be bigger because the QJAVAC must save AST and QAST attributes and status information.

We have to summarize that the reparsing QAST costs, which mainly includes the peak run time memory requirement and the compiler binary size, are all found to be worse when compared with JAVAC compiler. This also comes from the additional reparsing stage, mentioned earlier, to translate an abstract syntax tree to the queue abstract syntax tree.

Finally, we note that the AST generation algorithm can be easily realized because there are many mature full-grown system that can help to construct it (YACC, BISON, etc. are some examples.), but QAST is not so.

5. Conclusions and Future Work

In this paper, we proposed a parsing approach for generating QAST that is used for an optimized instruction generation in a QJAVAC compiler.

We first presented an overview of QJAVAC compiler. Then we presented the parsing QAST algorithm and its evaluation over a ranges of nodes. The evaluation of the QAST algorithm is also given.

With the QJAVAC compiler, we have successfully compiled some Java source code programs to QJVM byte codes. The compiling speed of the QJAVAC compiler was found to be slower than the well-known JAVAC compiler. This speed degradation comes from the fact that a reparsing stage to parsing AST to the QAST is added. The above extra-phase consumed nearly 30% of the total compilation time. However, the above overhead can be reduced if there is a parsing algorithm that can directly generate the QAST from token sequences that produces from scanning stage. This will be our future work.

References

1) Tremblay, M. and O'Connor, M.: picoJava^{TM}: A Hardware Implementation of the Java Virtual Machine, Proc. of IEEE Symp. on High-Performance Chips (1996).
2) Radhakrishnan R.: Java Runtime Systems Characterization and Architectural Implications, IEEE Trans. on Computers, Vol. 50, No.2, pp. 131–146 (2001).
3) Radhakrishnan R, Talla D, and John L.K.: Allowing for ILP in an embedded Java processor, Proc. of the ACM 27th annual Intl. Symp. on Computer Architecture, pp. 294–305 (2000).
4) Radhakrishnan R, Vijaykrishan N, John L.K, and Sivasubramanium A.: Archiitectural Issues In Java Run Systems, Proc. of 6th Intl. Symp. on High-Performance Computer Atchitecture, pp. 387–398 (2000).
5) Krall A, and Gra R.: CACAO-a 64 bit JavaVM Just-In-time Compiler: Concurrency Practice and Experience, Vol. 9, No. 11, pp. 1017–1030 (1997).
6) Sowa M, Abderazek B. A, Shigeta S, Nikolova K, and Yoshinaga T,: Proposal and Design of a Parallel Queue Processor Architecture (PQP), Proc. of 14th IASTED Intl. Conf. on Parallel and Distributed Computing and System, pp. 554–560 (2002).
7) Abderazek B.A, Nikolova K, and Sowa M.: FARM-Queue Mode: On a Practical Queue Execution Model, Proc. of the Intl. Conf. on Circuits and Systems, Computers and Communications, pp. 939–944 (2001).
8) Sun Microsystem: The Java^{TM} Language Specification, Second Edition, http://java.sun.com/docs/books/jls/index.html