2003－ARC－153 (10)
2003／5／8

Fast, Effective Instruction Generation Algorithm For Queue-Java Compiler (QJAVAC)

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

In this paper, we propose an effective Queue syntax tree algorithm (QST) that is used for an optimized QJVM instruction generation in a Queue Java compiler (QJAVAC).With the QJAVAC, that embedding QST, we have successfully compiled the Java source code to the QJVM byte code. The generated QJVM byte code validates that we can achieve a considerable higher ILP from QJVM than that from stack based.We describe the QST algorithm implementation and evaluation. From our evaluation results, we have found that the instruction generation time with the proposed QST algorithm is about 17% less than instructions generation time with conventional abstract tree (AST) algorithm over a range of nodes within an expression.

1. Introduction

Java is steadily increasing in popularity in the embedded and network chips arena, fueled primarily by the convenience and elegance of its "write once and run anyway", and also has advantage of small code size, 1.8 bytes per instruction on average as compared to other CISC or RISC machines, no source or destination register identifiers need to be assigned for the instructions, making the instructions small size[1].

The Java technology achieves its platform independence by compiling Java source code into machine independent "byte codes" which are executed on the JVM. These byte codes are typically executed by an Interpreter, or first, translated into native code and then executed by a Just-In-Time (JIT) compiler, or executed directly by specialized Java processors[2].

However, execution of typical Java applications, which are 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[3][4]. 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 byte code 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 30x slower than optimized C Code, whereas JIT compilers can provide up to 20x speedup with respect to interpreted code[5][6]. However, the memory requirement of JIT compilers is extremely expensive for pervasive applications[7][8].

To this end, and 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[9][10][12] that compiles Java source code into Queue-Java byte code(QJBT). A new Java execution mode named queue-Java based execution model (QJVM) in JVM has been, then, developed. Thus, we place the emphasis of high "in-code" parallelism at the expense of portability.

The QJVM uses a first-in-first–out (FIFO) data structure as the underlying control mechanism for the manipulation of operands and results. Thus, operands are retrieved from the front of an operand Queue (OPQ) pointed by a Queue head pointer (QH) and results are returned to the Queue tail pointed by a Queue tail pointer (QT).

In this paper, we will describe a Java System for QJVM and its instruction generation algorithm; which is a part of a whole research project being designed at Sowa Laboratory[12]. The compiler is named queue Java compiler (QJAVAC). The syntax trees in QJAVAC are a new type of syntax tree-queue syntax tree

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

(QST) and many of its sub-category QSTs that to presents different java language grammar items, all of them are optimized for QJVM instructions generating and it has been embed in QJAVAC.

The focus in this paper will be put on instruction generation algorithm with QST and its sub-QST. 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 to fast and effective to product queue instruction in QJAVAC. Base on it, we had successfully realized the QJAVAC with QST and suitable algorithm.

The rest of this paper is organized as follow: In section two, we present an overview of the QJAVA system and the QJAVAC compiler. In section three, we describe the proposed instruction generation algorithm of QST. Section four gives the evaluation results of this algorithm. In the last section, we give the conclusion and our future work.

2. QJAVA System Overview

2.1 QJAVA Architecture Overview

As was mentioned earlier, the queue-based execution model (QEM) performs most operations on a FIFO data structure[11]. However, the stack-based execution model (SEM) performs most operations on a first-in-last-out (FILO). As indicated in Figure 1, 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[13] released by Sun Microsystems, to protect Java

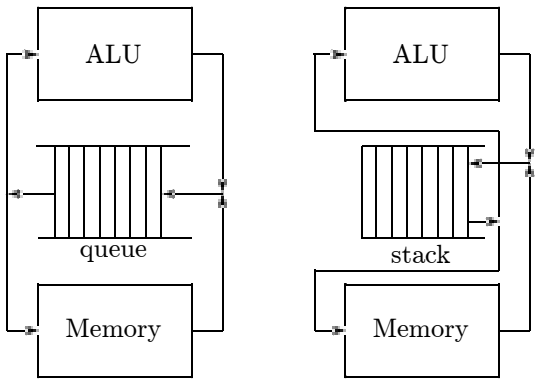

Fig. 1 Data Flow of Queue and Stack Models

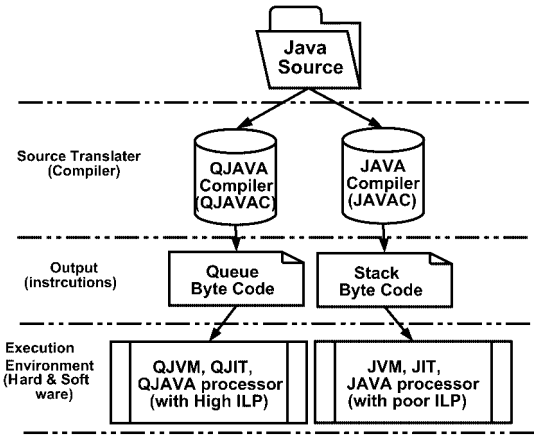

Fig. 2 QJAVA and Java Execution Environment

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 use QEM. However the conventional Java Virtual Machine uses SEM.

We will demonstrate later the merit of the QJAVA System. To ease understanding we give in Figure 2 a schematic representation of QJAVA and Java execution environment.

2.2 QJAVAC Implementation Overview

The QJAVAC is a complier, which has a similar implementation like other multiple phases Java Compilers. The QJAVAC overview is shown in Figure.3

In order to target the novel QJAVA architecture, a "re-Parsing" stage is added into the front end of JAVAC. So the QJAVAC front end consists of the following phases:

(1) Scanning: a scanner maps input charac-

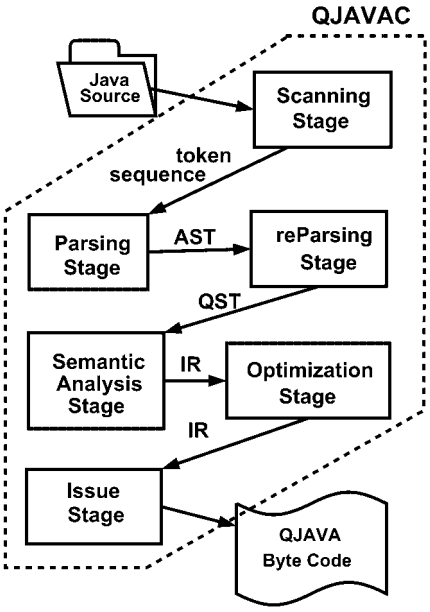

Fig. 3 QJAVAC Compiler Phases

ters into tokens.

(2) Parsing: a parser recognizes sequences of tokens according to some grammar and generates Abstract Syntax Trees (ASTs).

(3) Re-parsing: a translator that translates Abstract Syntax Trees (ASTs) into the Queue Syntax Trees (QSTs) according to some grammar.

(4) Semantic analysis: this phase performs type checking and translates QSTs into universal Intermediate Representations (IRs).

(5) Optimization: to optimize IRs.

The back end consists of the following phases:

(1) Instruction selection: to map IRs into assembly code

(2) Code optimization: to optimize the assembly code using control- and data flow analysis, etc.

(3) Code emission: to generate machine code from assembly code.

3. AST to QST Translation

In this section, we will explain what is QST, why we propose the QST and how we can fast and effective generate the Queue Byte Code ?

The QST is a new type of syntax tree that is optimum for QJAVAC compiler. To compare the QST with the abstract syntax tree (AST), we will use a simple example shown in Figure

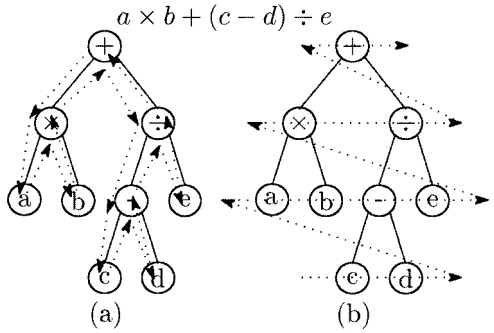

Fig. 4 Post and Level Traversal Order

4. In Figure 4(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 4(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".

In order to cope with this problem and reduce the instruction generation time, we propose another traversal algorithm. We call it Queue Syntax Tree (QST) in QJAVAC. The QST for a simple expression is shown in Figure 5. 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.

From Figure 5, we can also notice that the

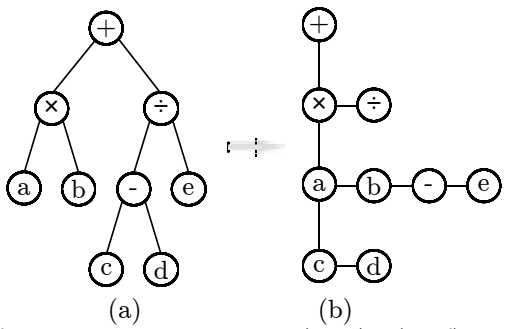

Fig. 5 QST for the expression $(a \times c) + (c - d) \div e$

sum of nodes in the expression are kept, the connections are created between brother and brother (same-level) nodes. In addition the necessary connections among some parents' and childs nodes are still preserved and the connection which are not exploited are removed. In other words, only necessary information is left in the above translated graph (right hand side of Figure 5)

Thus, using the same algorithm we can, then, translate any AST to QST. The translation flow chart is shown in Figure 6. However, there are three rules we have added:

- The connection between parent and its right-child is removed.
- The connection 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 stay as it is.
- A new connection is inserted between two children nodes since they are in the same level.

With the afro-described algorithm the instruction sequence can be easily generated for the QEM model.

4. Instructions Generation

Using the proposed QST, instruction sequence generations can be easily derived as described in flow chart given in Figure 7 acting as a recursive fucnction. There are basically five steps that should be followed to get a correct and efficient instructions generation. The above steps are summarized as follow:

(1) Enter a node; check whether it is a parent node.
(2) If it is, flow to his child node
(3) If it is not, emit the instruction(s) belonging to this node
(4) Check it for whether it has right side-brother node.
(5) If it has, flow to its right side-brother node. Then, back (loop) to step (1).

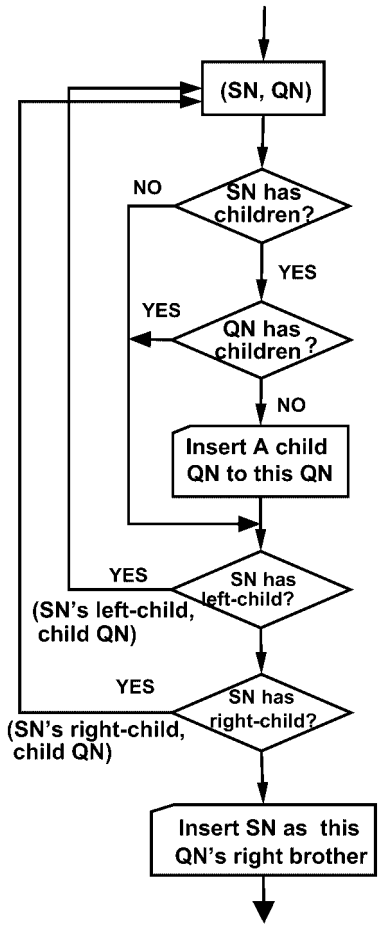

Fig. 6 Translation from AST to QST Flow Chart

5. Evaluations Results and Discussions

5.1 Methodology

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).

5.2 Instructions Generation Speed

We have compared the instructions generation speed of our proposed QST algorithm with abstract syntax tree instruction generation algorithm over a number of nodes. The above experiment is shown in Figure 8. From this experiment we conclude that the instructions gen-

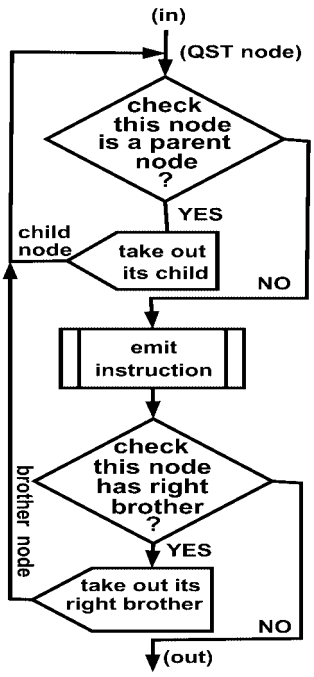

Fig. 7 Instruction Generation Algorithm

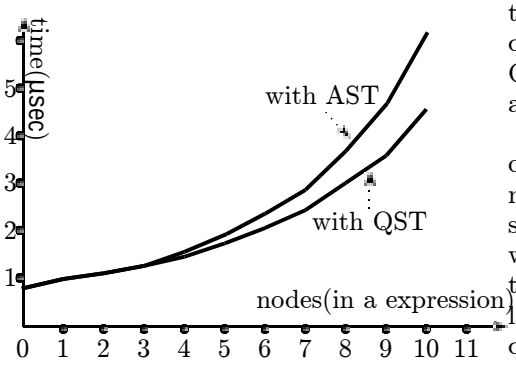

Fig. 8 Instruction Generation Speed Comparisons between AST and QST Algorithms

eration time is considerably reduced with the QST algorithm (i.e., for 9 nodes, the time is reduced from 4.43μsec to 3.68μsec). In other words, for 9-nodes expression, the generation time is about 16% less.

5.3 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 translation stage to translate an abstract syntax tree

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

to the queue syntax tree is added. That is in order to get a correct instruction sequence we must "re-parse" the AST to QST. 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 QST 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 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 QST attributes and status information.

We have to summarize that the translation 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 translation stage, mentioned earlier, to translate an abstract syntax tree to the queue 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.)

6. Conclusions and Future Work

In this paper, we proposed a fast and effective QST algorithm that is used for an optimized instruction generation in a QJAVAC compiler.

We first presented an overview of QJAVA system and QJAVAC compiler. Then we presented the QST algorithm and its evaluation over a ranges of nodes. The evaluation of the QST algorithm is given in term of instruction gener-

ation speed. We also compared its performance with the conventional AST algorithm

From our evaluation results, we conclude that the instruction generation time is considerably reduced with the QST instruction generation algorithm. In other words, for 9-nodes expression, the generation time is about 17% less. We also conclude that, the compiling speed of our QJAVAC compiler was found to be slower than the well-known JAVAC compiler. This speed degradation comes from the fact that a translation stage to translate an abstract syntax tree to the queue syntax tree 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 QST 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) Overview of Java^{TM} platform productct family, http://www.javasoft.com/profucts/OVjdkProduct.html
3) Radhakrishnan, R.: Java Runtime Systems Characterization and Architectural Implications, IEEE Trans. on Computers, Vol. 50, No.2, pp. 131–146 (2001).
4) Sun Microsystem: picoJava microprocessor core architecture, http://www.sun.com/microelectronics/picoJava/
5) 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).
6) 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).
7) 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).
8) Adl-Tabatabai, A., Cierniak, M., Lueh, L., Parikh, V. M., and Stichnoth, J. M.: Fast, effective code generation in a just-in-time Java compiler: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 280–290 (1998).
9) Sowa, M.: Fundamental of Queue machine, Technical Reports SLL97302, Sowa Lab., Univ. of Electro-Communications (2000).
10) Sowa, M., Abderazek, B. A., Shigeta, A., 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).
11) Abderazek, B. A., Nikolova, K., and Sowa, M.: FARM-Queue Mode: On a Practical Queue Execution Model, it Proc. of the Intl. Conf. on Circuits and Systems, Computers and Communications, pp. 939–944 (2001).
12) Sowa Laboratory: http://www.sowa.is.uec.ac.jp
13) Sun Microsystem: The Java^{TM} Language Specification, Second Edition, http://java.sun.com/docs/books/jls/index.html