QJAVAC: Java Compiler Design for High Parallelism Queue Java Byte Code

Li.Qiang.Wang, Ben A.Abderazek, Soichi Shigeta, Tsutomu Yoshinaga and Masahiro Sowa

Abstract: In this paper, we will describe the implementation and the evaluation of a Queue-Java compiler (QJAVAC), which is a part of whole research project at our laboratory, for high-level parallelism Queue-Java bytecodes without too much need for parallelism scheduling. We will also describe a new type of syntax tree-Queue Abstract Syntax Tree(QAST) that is used for an optimized Queue Java Virtual Machine (QJVM) instruction generation. With the QJAVAC compiler, we have successfully compiled the Java source code to the QJVM byte code, the achieved average degree of parallelism is about 2.11 times greater than that of a general Java byte code.

1. Introduction

Java is steadily increasing in popularity in the embedded and network chips arena. The Java technology achieves its platform independence by compiling Java source code into machine independent “bytecodes” that are executed on the JVM. These bytecodes are typically executed by an Interpreter, or a Just-In-Time (JIT) compiler, or executed directly by specialized Java processors. In addition, it was widely recognized that each current execution mode for java has its own strength and weakness. While software emulation is easy to implement, it incurs a high overhead during execution of certain Java workloads. A Java processor eliminates the software layer. However, a direct execution of bytecodes on stack-based processors is invariably constrained by the limitations of the stack architecture for accessing operands. This inherent the dependence severely limits the instruction level parallelism.

In order to improve the instruction level parallelism in java byte code, we proposed a new parallel Java execution model named queue-based Java execution model (QJVM). The above new computing model is based on the Queue computing scheme and uses a first in first out (FIFO) queue data structure as the underlying control mechanism for the manipulation of operands and results.

As a result, we will show that the QJVM implemented in a pipelined arithmetic/logic unit (ALU) is more efficient than the stack based JVM. That is, the QJVM has the potential to take advantage of a pipelined ALU when the operand queue is not empty. On the other hand, the Stack based JVM cannot exploit pipelined ALU since the result of one operation must be returned to the top of the stack before they can become the operands of the next operation.

In this paper, we will describe the implementation and the evaluation of a Queue Java compiler (QJAVAC), which is a part of whole research project at Sowa Laboratory, for high-level parallelism Queue-Java bytecodes without too much need for parallelism scheduling of byte code. We will describe a new type of syntax tree-Queue Abstract Syntax Tree(QAST) that is used for an optimized QJVM instruction generation.

The rest of this paper is organized as follow: In section 2, we review the related work and the background. In section 3, we describe the Queue-based Java compiler (QJAVAC) implementation. We also show in section 2 the algorithm for generating the new proposed QAST and how instructions are generated in QJAVAC by using QAST. In section 4, we describe the benchmarks as well as our evaluation results. Finally, we give our concluding remarks and the future work.

2. QJAVA System Description

2.1 QJAVA Architecture Overview

The queue-based execution model (QEM) performs most operations on a first-in-first-out (FIFO) queue data structure. While the stack-based execution model (SEM) performs the 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, just as SEM, operations, which implicitly reference an operand stack.

Each instruction takes out the required number of operands from the head of the operand queue, performs some computations, and stores the results of the computations at the tail of the operand queue, which occupies continuous storage locations (described later). The order the execution order of instructions coincides with the position of the data in the operand queue.

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

In our system, the syntax and semantics of QJAVA 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 ILP of QJAVA comes from the Queue Java Virtual Machine(QJVM) which use
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 1 a schematic representation of QJAVA and Java execution environment.

2.2 QJAVAC Implementation Overview

The QJAVAC is a compiler, which has a similar implementation like other multiple phases Java Compilers. The QJAVAC overview is shown in Figure 2.

In order to target the novel QJAVA architecture, a "reParsing" 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 characters into tokens.
2. Parsing: a parser recognizes sequences of tokens according to some grammar and generates ASTs.
3. Re-parsing: a translator that translates ASTs into the QASTs.
4. Semantic analysis: this phase performs type checking and translates QASTs 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.

2.3 AST to QAST Translation

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 the 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.

However, this traversal will not work when dealing with QEM. Therefore, to get a correct instruction generation sequences for the QJAVAC, the traversal of 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 produce 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 difficult and its algorithm becomes very complex, huge, and “time hungry”.

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

From Figure 3(c), we can also notice that the sum of nodes in the expression are kept, the connection created between brother and brother (same-level) nodes.
In addition the necessary connections among some parents’ and child 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.

Thus, using the same algorithm we can, then, translate any AST to QAST. The translation flow chart is shown in Figure 4. 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 AST, 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.

2.4 Instructions Generation

Using the proposed QAST, instruction sequences can be easily derived as described in flow chart given in Figure 5 acting as a recursive function. There are basically five steps that should be followed to get correct and efficient instructions generation. These 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).

The instructions generation algorithm with QAST is very simply and effective.

3. Evaluations Results and Discussions

3.1 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).

3.2 Instructions Generation Speed

We have compared the instructions generation speed of our proposed QAST algorithm with AST instruction generation algorithm over a number of nodes. The above experiment is shown in Figure 6. From this experiment we conclude that the instructions generation time is considerably reduced with the QAST 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.

3.3 Parallelism Achievements

With the QJAVAC compiler, we have successfully compiled the Java source code to the QJVM byte code. We also simulated the parallelism performance of QJAVA for some algorithms. As indicated in Table 1, we used four benchmark programs (FFTR, Quick sort, RSA and LZW). For each benchmark we measured the Maximum parallelism (Max. Parallelism) and the Average Parallelism (Ave Parallelism). For example, for FFT benchmark the Max parallelism in QJAVA...
times greater than in JAVA. The average Max Parallelism for all benchmark we compiled in QJAVA is 2.72 time greater than in JAVA and the average Parallelism in QJAVA is about 2 times greater than that in JAVA.

Table 1. comparing result of qjavac and javac for some algorithm

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Max parallelism</th>
<th>Ave parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JAVA</td>
<td>QJAVA</td>
</tr>
<tr>
<td>FFT</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>RSA</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>LZW</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

3.4 Compilation Speed and Space

Unfortunately the compiling speed of the QJAVAC compiler was found to be slower than the well-known JAVA compiler. This speed degradation comes from the fact that a translation stage to translate an AST to the QAST 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 can be 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 is shown in Table 2. 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.

Table 2. QJAVAC Compiler Size

<table>
<thead>
<tr>
<th>Compiler components</th>
<th>Function source size (KB)</th>
<th>Head source size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical scanner</td>
<td>709</td>
<td>339</td>
</tr>
<tr>
<td>AST parsing</td>
<td>219</td>
<td>294</td>
</tr>
<tr>
<td>QAST reparsing</td>
<td>291</td>
<td>30</td>
</tr>
<tr>
<td>Code generator</td>
<td>789</td>
<td>30</td>
</tr>
<tr>
<td>Others</td>
<td>822</td>
<td>123</td>
</tr>
<tr>
<td>Total</td>
<td>2897</td>
<td>789</td>
</tr>
</tbody>
</table>

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 AST to the QAST.

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.)

4. Conclusions and Future Work

In this paper, we proposed a Queue Java compiler (QJAVAC) for high parallelism Queue Java byte codes. The compiler is targeted for Queue Java System. We described the implementation and the evaluation of the above QJAVAC compiler. The evaluation is performed in terms of degree of parallelism, compiling time and space over a range of benchmark programs. For fast and effective instruction generation, we also proposed a Queue abstract syntax tree (QAST) algorithm.

With the QJAVAC compiler, we have successfully compiled some Java source code programs to Queue byte codes. The generated QJVM byte code validates that we can get a considerable high instruction level parallelism from Queue based instruction sequences compared with the stack based instruction sequences in some java applications.

From our evaluation results, we found that the AST generation time is considerably reduced in the QAST instruction generation algorithm. In other words, for 9-nodes expression, the generation time is about 16% less. We also conclude that, the compiling speed of the QJAVAC compiler was found to be slower than the well-known JAVA compiler. The speed degradation comes from the fact that a translation stage to translate 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


