Queue Compiler Development

Arquimedes Canedo,† Ben Abderazek † and Masahiro Sowa†

Queue processors are a viable alternative for high performance embedded computing and parallel processing. We present the design and implementation of a compiler for a queue-based processor. Instructions of a queue processor implicitly reference their operands making the programs free of false dependencies. Compiling for a queue machine differs from traditional compilation methods for register machines. The queue compiler is responsible of scheduling the program in level-order manner to expose natural parallelism and calculating instructions relative offset values to access their operands. This paper describes the phases and data structures used in the queue compiler to compile C programs into assembly code for the QueueCore, an embedded queue processor. Experimental results demonstrate that our compiler produces good code in terms of parallelism and code size when compared to code produced by a traditional compiler for a RISC processor.

1. Introduction

Although silicon technology will continue delivering massive levels of integration, current computer architectures have serious limitations to utilize all those transistors into an equivalent performance\(^1\). Novel computer architectures are being designed to fully exploit the future silicon technology\(^2,3\). Queue computing\(^4\)∼\(^10\) is a promising computation scheme for the present challenges of microprocessor design. A queue-based computer uses a FIFO queue as the storage location to perform operations. Instructions write data into the rear of the queue, and read data from the head. In our previous work we have investigated and proposed a 32-bit queue-based processor, the QueueCore\(^5\), featuring a simple hardware design, parallel processing capabilities, reduced instruction set architecture, and high code density.

Generating code for the queue computation model introduces serious challenges for the compiler\(^11\),\(^12\). Architectural enhancements in a queue-based instruction set allow the design of realizable code generation algorithms\(^13\),\(^14\). The QueueCore processor implements a producer order queue computation model instruction set where operands can be read from a distinct location of the head of the queue specified as an offset reference in the instruction. This architectural modification enables the processor and the compiler to find and execute common subexpressions.

Our previous attempts of developing a queue compiler were focused on using a conventional retargettable compiler for register machines and mapping the register code into queue code\(^15\),\(^16\). This approach led to complex mapping algorithms and inefficient queue code. This paper presents the development of the first queue compiler infrastructure designed for the queue computation model. The goal of the design of such infrastructure is to provide an automated code generation tool for a universal queue computation model. Algorithms and data structures were developed to maintain queue code generation independent from the source language and the target architecture. The uniqueness of the queue compiler is given by the level-order scheduling of programs and instruction’s offset calculation. The resulting compiler is a self-hosted compiler targeting the QueueCore embedded processor capable of translating any application written in C language to assembly code.

The remaining of this paper is as follows. In Section 2 we give the related work to queue computing and compilation for queue machines. In Section 3 we describe the queue computation model, we introduce the problems of generating code for such paradigm, and we present our producer order queue computation model that copes with all these problems. Section 4 describes the development of the queue compiler. In Section 5 we present the evaluation of the queue compiler. Finally, Section 6 gives conclusion.

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


2. Related Work

A handful of queue computer designs have been proposed. In 1985, Bruno\(^9\) established that a level-order scheduling of an expression’s parse tree generates the sequence of instructions for correct evaluation. A level-order scheduling of directed acyclic graphs (DAG) still delivers the correct sequence of instructions but requires additional hardware support. This hardware solution is called an Indexed Queue Machine. The basic idea is to specify, for each instruction, the location with respect of the head of the queue (an index) where the result of the instruction will be used. An instruction may include several indexes if it has multiple parent nodes. All these ideas were an abstract machine until the hardware mechanisms of a superscalar queue machine was proposed by Okamoto\(^7\). These superscalar queue machine realizes the abstract design into a hardware implementation capable of executing instructions in parallel.

The operand queue has been used as a supporting hardware for two register based processors for the efficient execution of loops. The WM architecture\(^9\) is a register machine that reserves one of its registers to access the queue and demonstrates high streaming processing capabilities. The authors of WM Architecture claim to develop compiler support but details are not elaborated. In\(^17,18\) the use of Register Queues (RQs) is demonstrated to effectively reduce the register pressure on software pipelined loops. The compiler techniques described in this processor do not present a significant contribution for queue compilation since the RQs are treated as special registers.

In our previous work\(^4,19\), we have designed the QueueCore, a parallel queue processor capable of executing any data flow graph. The QueueCore breaks the rule of dequeueing by allowing operands to be read from a location different than the head of the queue. This location is specified as an offset in the instruction. The fundamental difference with the indexed queue machine is that our design specifies an offset reference in the instruction for reading operands instead of specifying an index to write operands. In the QueueCore’s instruction set, the writing location at the rear of the queue remains fixed for all instructions.

Compiling for queue machines still is an undecided art. Only few efforts have been made to develop the code generation algorithms for the queue computation model. A linear time algorithm to recognize the covering of a DAG in one queue has been demonstrated in\(^12\) together with the proof of NP-completeness for recognizing a 2-queue DAG. In\(^8\), Schmit. et. al propose a heuristic algorithm to cover any DAG in one queue by adding special instructions to the data flow graph. From their experimental results a large amount of additional instructions is reported, making this technique insufficient for achieving small code size. Despite the large amount of extra instructions, the resulting size of their tested programs is smaller than RISC design. We tried to develop a queue compiler based on a retargetable compiler for register machines\(^15\). A large amount of registers was defined in the compiler to model the operand queue and to avoid spill code. Nevertheless, mapping register code into the queue computation model turns into low code quality with excess of instructions making this approach inappropriate for both, a native compiler for our queue machines, and the generation of compact code. In this article we present part of the initiative to deliver a compiler technology designed specifically for the queue computation model.

3. Queue Computing Overview

A queue-based computer uses a first-in first-out queue to evaluate expressions. Comparably, stack computers use a last-in first-out stack, and register computers use random access registers to compute expressions. Queue computation model (QCM) is the set of rules and operations which allow any program to be executed in a queue. Data is written into the queue, enqueued, at the rear (or tail). And data is read, dequeued, at the head of the queue. Two pointers are maintained by the hardware to keep track of the location of the head and tail of the queue. The \(\text{QH}\) and \(\text{QT}\) pointers, respectively. Since the reading and writing locations are fixed, the instruction format of a queue computer consists only of the operation. Operands are implicitly accessed by the operations. This characteristic makes the queue programs free of false dependencies\(^4,19\) and simplifies the hardware as no register renaming mechanism\(^20\) is needed. Moreover, the zero operand instruction format contributes to high density in the resulting code.

A level-order traversal of the parse tree of any expression gives the instruction sequence to cor-
rectly compute the result$^6)$. Figure 1(a) shows
the parse tree for expression $x = \frac{a^2 + b^2}{a}$.
The traversal visits all the nodes in every level
from left to right starting from the deepest level
towards the root. The resulting sequence of
instructions, queue program, is shown in Fig-
ure 1(b). After the first four instructions in
level $L_4$ are executed, the contents of the queue
are $\{a, a, b, b\}$ with $QH$ pointing to the first
element (leftmost), and $QT$ at an empty loca-
tion after last loaded element (rightmost). The
“mul” instruction is a binary operation that
reads its operands from $QH$ computes the mul-
tiplication, and writes the result at $QT$. After
executing all instructions of level $L_3$ the con-
tents of the queue changed to $\{T_1, T_2\}$, where
$T_1 = a \times a$, $T_2 = b \times b$. The rest of the program
is executed in similar way until the final result
“$x$” is computed and stored to memory by the
last instruction leaving an empty queue.

One solution is converting the input DAG
into a level-planar DAG. Nevertheless, recog-
nizing this kind of DAGs is a NP-Complete
problem$^{11),12)}$. In$^8)$, a heuristic to level-
planarize DAGs by inserting special operations
that manipulate data in the queue by rotating
and swapping elements has been proposed.
This study has shown that this approach has
the overhead of a large number of extra instruc-
tions. Another solution is to allow the instruc-
tions of the queue processor to explicitly spec-
ify the location from where to read operands.
In our previous work$^{4),5),19)}$, we have investi-
gated and developed a parallel queue processor
(QueueCore) that allows instructions to read
data not only from $QH$ but from anywhere in
the queue. Yet, all instructions implicitly write
the result into $QT$. As the instructions allow
operands to be read from anywhere in the queue
but results are always written (produced) into
$QT$, we call this scheme producer-order queue
computation model.

When evaluating an expression, a directed
acyclic graph (DAG) is often preferred over a
parse tree as it reduces the number of nodes
by eliminating common subexpressions$^{21),22)}$.
Given a DAG, a queue program is also ob-
tained by a level-order traversal$^5)$. However,
a pure queue-based computer does not suffice
to correctly evaluate DAGs. To illustrate the
problem, consider the DAG of expression “$x = \frac{a^2 + b^2}{a}$” in Figure 2(a). Compared to its
tree representation in Figure 1 the operand “$a$”
has three parent nodes instead of one, operand
$b$ has two parents, and the number of nodes
has been reduced by three. If the program
in Figure 2(b) is executed, the problem arises
when executing the first “mul” operation. At
that point, the queue contains two elements
$\{a, b\}$, enqueued by the first two load instruc-
tions. The multiplication then dequeues its first
operand $a$ correctly, but fails to obtain its sec-
ond operand getting $b$ instead of $a$. Therefore,
value \( N \neq 1 \). And for 2-offset instructions, both offset references are read from a location different than \( QH \). Unary operations follow the same idea but are restricted to only one operand.

Figure 3 shows the QueueCore pseudoprogram to evaluate the expression \( x = (a^2 + b^2)/a^3 \). First two instructions load operands \( \{a,b\} \) into the queue. The third instruction is a 1-offset instruction that dequeues \( a \) two times, the first from \( QH \) and the second from \( QH-1 \) leaving the queue status as \( \{b,T_1\} \), where \( T_1 = a \times a \). Similarly, the fourth instruction computes \( b \times b \) leaving the queue as \( \{T_1,T_2\} \). The addition instruction is a 0-offset instruction that computes the temporary \( T_3 = (a^2 + b^2) \). Before the execution of the division instruction, the queue contains a single element \( \{T_3\} \). The division requires its first operand to be dequeued from \( QH \) and its second operand, \( a \), to be dequeued from four queue words away from \( QH \). The negative offset reference indicates that the operand to be dequeued has been already consumed by a previous operation. This example illustrates how the QueueCore instruction set allows the correct evaluation of any DAG.

![Fig. 3 Producer order instructions](image)

### Table 1 Producer order instruction classification

<table>
<thead>
<tr>
<th>Type</th>
<th>Binary</th>
<th>Unary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-offset</td>
<td>mul 0, 1</td>
<td>not 0</td>
</tr>
<tr>
<td>1-offset</td>
<td>add -3, 0</td>
<td>neg -3</td>
</tr>
<tr>
<td>2-offset</td>
<td>div -3,-1</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### 4. Queue Compiler

Compiling for the queue computation model differs from the conventional techniques used in compilers for register machines since queue instructions require an offset reference value rather than a location name (e.g. register number). In\(^{13,14}\), we have investigated and developed the code generation algorithm specifically for the queue computation model. In this section, we propose the implementation of the queue compiler infrastructure. We describe the compiler internals and data structures used to successfully build a queue compiler.

The queue compiler infrastructure is divided into five phases, including the front-end. The block diagram of our queue compiler is given in Figure 4. The front-end parses C files into abstract syntax trees (AST) which are a high level intermediate representation (HIR). Having language independent ASTs as the HIR facilitates the addition of multiple front-ends parsing different languages. The queue compiler infrastructure consists of the remaining four phases. Instruction selection phase lowers the high level constructs in ASTs into a generic low level queue sequence of instructions. The resulting representation is a tree-like structure called QTrees. The second phase is the offset calculation where offset references for every instruction in the program are computed. Offset calculation phase depends on a special data structure called leveled DAGs where nodes are assigned to hierarchical levels according data dependences among operations. Therefore, QTrees are first transformed to LDAGs. The third phase takes LDAGs and schedules the program in level-order manner to comply with the queue computing principle. As a result, a linear low level intermediate representation is generated. The fourth and last phase generates assembly code for the given target architecture. Throughout the compilation process, a common symbol table is maintained and shared by all phases of the queue compiler.

One of the goals in our design was to keep the compiler implementation independent from the target architecture. Although the presented compiler generates code for a specific target architecture, the QueueCore processor, all the algorithms and data structures are machine independent and can be applied for the queue computation model in general.

#### 4.1 Instruction Selection

The front-end of the compiler is based on GCC 4.0.2 and it parses C files into abstract syntax trees (AST) called GIMPLE\(^{23}\). GIMPLE representation is a three-address code suitable for code generation for register machines but not for queue machines. As the level-order scheduling traverses the full DAG of an expression, we facilitate the job of the scheduler by reconstructing GIMPLE trees into trees of arbitrary depth and width called QTrees. During the expansion of GIMPLE into QTrees we also translate the high-level constructs such as ag-
aggregate types into their low level representation in generic queue instructions. We have defined a set of generic queue instructions divided into seven classes as shown in Table 2. The instruction selection phase is in charge of expressing any high level statement into generic queue instructions equivalent. Figure 5 shows a fragment of a C program and its GIMPLE representation. Notice that in the C-like GIMPLE representation in Figure 5(b), the statement inside the conditional is split into three-address statements with the help of compiler generated temporaries (i.e. D1098). To illustrate the high level nature of GIMPLE representation we show in Figure 5(c) the same program but using GIMPLE language. As shown in Figure 6(a), QTrees are unrestricted in the number of operands and operations. For the given example, the QTree representation is shown in Figure 6(b). Now the program is expressed through low level generic queue instructions.

### 4.2 Offset Calculation

After having fully expanded expressions to QTree representation, we build the core data structure, the leveled DAGs (LDAG) using a recursive algorithm. Formally, a LDAG $\vec{G} = (V, \vec{E})$ is the mapping of the nodes to integers such that if there is an edge from $u$ to $v$, then $lev(v) = lev(u) + 1$ for all edges in the graph. A node $v$ is a level-$j$ node if $lev(v) = j$. The characteristic that makes LDAGs the most suitable data structure for finding the offset references is the hierarchical organization of the nodes in the DAG according their true data de-

---

**Table 2** Generic queue instructions

<table>
<thead>
<tr>
<th>Class</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, div, mul, neg, rsh, leh, ior, xor, and, not</td>
</tr>
<tr>
<td>&amp; Logic</td>
<td>leh, ior, xor, and, not</td>
</tr>
<tr>
<td>Memory</td>
<td>ld, st, lea, sld, stt</td>
</tr>
<tr>
<td>Comparison</td>
<td>ceq, cne, clt, cle, cgt, cge</td>
</tr>
<tr>
<td>Ctl. Flow</td>
<td>bt, bf, jal, ret</td>
</tr>
<tr>
<td>Conversion</td>
<td>conv</td>
</tr>
<tr>
<td>Special</td>
<td>copy, dup, rot</td>
</tr>
<tr>
<td>Queue</td>
<td>moveq, movet</td>
</tr>
</tbody>
</table>

---

![Fig. 4 Queue compiler block diagram](image)

Fig. 4 Queue compiler block diagram

![Fig. 5](image)

Fig. 5 High-level intermediate representation. (a) C fragment, (b) C-like GIMPLE representation, (c) GIMPLE representation

![Fig. 6](image)

Fig. 6 QTrees. (a) C-like QTree representation, (b) QTree representation using low level generic queue instructions
dependencies into levels. Notice that all instructions in the same level are independent from each other. Therefore, the queue compiler exposes maximum natural instruction level parallelism to the queue processor. The compiler builds a list of slots that bind together all nodes of every level. Figure 7 shows a LDAG for expression “\[a[i] = (&a + (i \times \text{sizeof}(a))) \times (x + y)\]”. The squared nodes in the left of the figure are the slots that indicate the levels of the DAG.

The squared nodes in the left of the figure are the slots that indicate the levels of the DAG.

The offset calculation phase calculates, for every binary and unary instruction in the program, the offset reference values to access their operands. Two steps are required to obtain the offset reference value for any operation \(u\). First, the \(\text{QH}\) position relative to \(u\) must be determined\(^{13},14\). And second, the distance \(\delta\) from \(\text{QH}\) to the operand must be calculated. For any operand \(m\) of \(u\), the offset reference value is given by the following equation:

\[
\text{offset}(m) = \delta(\text{QH}, \text{pos}(u), m)
\]  

(1)

The distance between two nodes \((u, v)\) in a LDAG is given by the number of nodes found in the level-order traversal from \(u\) to \(v\). For example, the \(\text{QH}\) relative position with respect of \(\text{sst}\) node in level \(L_0\) of Figure 7 is in the only node of level \(L_1\), shown by the dotted line. For its first operand, the \(\text{QH}\) points exactly to the same location in the LDAG, making an offset reference value of zero. For its second operand, the first node in level \(L_3\), the distance between \(\text{QH}\) and the second operand is five as five nodes are visited in a reversed level-order traversal (+, 1\(\text{ds}\), y, x, +). The output of this phase is LADGs with offset reference values computed for every operand of every instruction.

4.3 Instruction Scheduling

The instruction scheduling algorithm of our compiler\(^{14}\) is a variation of basic block scheduling\(^{22}\) where the only difference is that instructions are generated from a level-order topological order of the LDAGs. The input of the algorithm is an LDAG annotated with offset reference values. For every level in the LDAG, from the deepest level to the root level, all nodes are traversed from left to right and an equivalent low level intermediate representation instruction is selected for every visited node. The output of the algorithm is QTL, a linear low level intermediate representation of the program implemented as a double linked list. To match the nature of the queue computation model as much as possible, we include in the QTL nodes a single operand. The only operand is exclusively used by memory, and branch instructions to specify the memory location or the target of the jump. All binary and unary instructions resemble the queue computation model having zero operands. We consider offset references attributes of the instructions. Additionally, we insert annotations in QTL to facilitate machine dependent transformations.

Figure 8 shows the QTL debugging representation of the sample C program in Section 4.1. The instructions have one to one equivalence to the generic queue instructions defined for the queue compiler. Annotations that mark the beginning of levels are shown and every level has a unique identifier. All instructions show the data type attribute as [iws]. The offset reference values are indicated for binary operations. Notice that memory operations (\(\text{PUSH}_\text{Q}, \text{LOAD}_\text{Q}\)) have an explicit operand that represents the memory location of local variables or numeric constants.

4.4 Assembly Generation

The last phase translates the QTL program list into assembly code for the QueueCore processor. Figure 9 shows the assembler output for the C program fragment in Section 4.1. All instructions in the assembly language consist of the opcode followed by the data type on which the instruction should be executed. Depending on the instruction type, there maybe be from one to three extra operands. For example, the instruction “\(j \text{ iws}, L2\)” has only one extra operand that indicates the target label to where the jump instruction should pass control. Binary instructions such as “\(\text{add}_\text{Q}, \text{qt}, \text{qh}, \text{qh+1}\)” have three extra operands. The \(\text{qh}\) in-
Fig. 8 QTL representation indicates that the first source operand should be taken from QH (the zero offset is omitted), and the second source operand from QH + 1. Although the QueueCore implicitly writes to the queue always to the QT, we include qt operand in the assembly for readability. The assembler is in charge of removing unnecessary fields from the code when generating object code, therefore, the qt field is removed from the instructions in the object code. Memory operations such as "ld iws, qt, $fp+12" have two extra operands, the destination qt and the memory location where to access the operand. The $fp represents the frame pointer, a special purpose register to access local variables.

```
ld  iws, qt, ($fp)+
ldi  iws,  1
eq  iws,  0,  0,  0,  0
bl   iws,  li,  bec

li:   ld  iws, qt, ($fp)+
      li   iws,  0
      lea  iws, qt, ($fp)+
      mov  iws, qt, $a0+1
      add  iws, qt, $a0+1
      ld   iws, ($fp)+
      add  iws, qt, $a0+1
      mov  iws, qt, $a0+1
      add  iws, qt, $a0+1
      ble  iws, li, 0
```

Fig. 9 QueueCore assembly output

5. Results

We have successfully developed a queue compiler infrastructure. All algorithms, data structures, and phases of the compiler were designed specifically for the queue computation model. Our queue compiler transforms C programs into assembly language for the QueueCore processor. The primary concern of this paper is to evaluate the quality of our compiler by means of measuring the characteristics of the output code. We first demonstrate the improvement of the proposed queue compiler over our previous attempts of compilers for queue machines. Second, we measure the quality of the generated code in terms of code size and parallelism using the output of an optimizing compiler for a conventional 8-way issue RISC processor as the baseline. The target architecture of our compiler is the QueueCore processor, a 32-bit processor capable of parallel execution.

Excluding the front-end, the queue compiler is about ten thousand lines of C code. Table 3 shows the number of lines of code for each phase. Offset calculation and instruction scheduling phase account for about 50% of the code of the compiler. These two phases are the most critical parts of the compiler since they are in charge of calculating the offset references for the instructions in the program, and scheduling the code in level-order manner. The fifth phase in the table, others, are the complementary supporting functions and macro definitions for all phases of the compiler. The symbol table is included in this category.

Table 3 Cost of the phases of the queue compiler in lines of C code.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Lines of C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction selection</td>
<td>2000</td>
</tr>
<tr>
<td>Offset calculation</td>
<td>2500</td>
</tr>
<tr>
<td>Instruction scheduling</td>
<td>2500</td>
</tr>
<tr>
<td>Assembly generation</td>
<td>1000</td>
</tr>
<tr>
<td>Others</td>
<td>2000</td>
</tr>
</tbody>
</table>

To analyze the complexity of our compiler, we measured the real time taken by the queue compiler to compile SPEC CINT95 benchmarks. The system utilized for this measurement is a dual 3.2 GHz Xeon processor computer running Linux 2.6.20 kernel. Table 4 shows the results. The second column (LOC) shows the lines of C code corresponding to each benchmark including comments. The third column (Queue) shows the compile time, in seconds, taken by our compiler to generate assembly code for the QueueCore processor. The last column (Native) shows the time taken by a native compiler, GCC 4.0.2, to compile the benchmarks into x86 assembly code. For a fair evaluation and not to affect compile time, the optimizations in the native compiler were turned off. These results show that the complexity of the queue compiler is comparable to the complexity of a compiler for a register machine. For most of benchmarks, the queue compiler takes less time to generate
5.1 Queue Compiler Infrastructure Effectiveness

Our previous work delivered a compiler for queue machines based on a retargetable compiler infrastructure for register machines\(^\text{15}\). This compiler defined a large number of general purpose registers in the machine description file of the code generator to model the queue register file and avoid the insertion of register spill code. Then, the program compiled for a register machine (after scheduling and register allocation) was re-scheduled following the level-order traversal. This approach led to complex algorithms and poor code quality in both, code size, and parallelism. Many instructions were needed to map the register code into its queue program equivalent. The compiler infrastructure we present here is a completely different compiler from our previous attempt. We estimate the efficiency of our queue compiler infrastructure by comparing the number of instructions generated for a set of six test programs against two versions of our previous compiler: PQP-GCC\(^{16}\), and QRP-GCC\(^{15}\).

The graph in Figure 10 shows the instruction count generated by two versions of our previous compiler and the queue compiler infrastructure for the test programs. From these experimental results we can conclude that the queue code generation algorithms implemented in the queue compiler infrastructure efficiently generate programs without unnecessary instructions. The quality of the code generated by the queue compiler is higher than the code produced by the PQP-GCC and QRP-GCC. In average, the queue compiler generates about 45% shorter programs. In contrast to the QRP-GCC and PQP-GCC, our queue compiler infrastructure is capable of compiling complete applications rather than simple and small toy programs.

5.2 Compiler Output Analysis

We selected eleven applications from MiBench\(^{26}\) and SPECint95\(^{25}\) benchmark suites to measure the code size and instruction level parallelism features of our compiler. Code size is measured from the text segment of the object code. Given the small instructions of our QueueCore processor, two byte instruction set, our programs present high code density. We compare the code size generated by an optimizing compiler for a conventional register machine against the output of our compiler to judge the efficiency of the queue compiler. For the conventional register machine, we use GCC 4.0.2 compiler with maximum optimizations enabled (-O3) targeting the MIPS I\(^{27}\) instruction set. The queue compiler was configured without optimizations. Figure 11 shows the normalized code size for the benchmark programs using the register machine as baseline. In average, queue programs 30% denser programs than fully optimized register machine code. We believe that our results can be improved further by the addition of code optimizations to the queue compiler infrastructure. Our ongoing work includes the addition of classical optimizations\(^{22}\) to the queue compiler.

Using the same compiler configuration, we
obtained the compile-time exposed parallelism. Although run-time parallelism would give a more realistic estimation of the real system, it is affected by some hardware-dependent factors such as cache, and branch prediction performance that are difficult to calculate by the compiler. In the queue assembly the parallelism is obtained by measuring the number of instructions executed in every independent level of the data flow graph. Figure 12 shows the compile-time exposed parallelism by the traditional optimizing compiler for a universal RISC machine and the queue compiler. On average, the queue compiler without ILP optimizations is capable of extracting about 7% more parallelism than fully optimized register code. The inclusion of sophisticated ILP optimizations in the queue compiler, we presume, would benefit the extraction of ILP in queue programs as these transformations are machine independent and can be applied to queue machines.

6. Conclusion

In this paper we presented the development of the first compiler infrastructure tailored specifically for the queue computation model. The queue compiler effectively solves the problems of code generation for the queue computing model by constructing a proper representation of the data flow graph. The chosen data structures allow us to calculate the two conditions needed for the calculation of offset references: $Q_H$ relative position to any instruction, and distance between two nodes. The queue compiler exposes maximum parallelism by scheduling the program in level-order manner. We demonstrated that this approach delivers higher code quality compared to our previous hybrid queue-register compiler attempt. We estimated the overall quality of our compiler by compiling a set of multimedia and integer applications for the QueueCore processor and measuring the code size and extracted ILP against the output of an optimizing compiler for a conventional register architecture. The queue programs show, in average, 30% more density and 7% more parallelism than fully optimized register code. These results highlight the potential of queue-based architectures for high-performance and embedded computing.

References


(Received January 1, 2007)
(Received January 11, 2007)

**Arquimedes Canedo** received his B.Eng degree from the National Polytechnic Institute (IPN) of Mexico and his M.Eng degree from the Graduate School of Information Systems of the University of Electro-Communications, Tokyo, where he is currently pursuing his doctor degree under the supervision of Prof. Sowa. His research interests include optimizing compilers, algorithms and data structures, computer languages, non-conventional computer architectures, and queue computing.
Ben Abderazek received his M.E. degree from Huazhong University of Science and Technology, and his Ph.D degree from the Graduate School of Information Systems, National University of Electro Communications, Tokyo, in 1997 and 2002 respectively. Currently he is an assistant professor at the Graduate School of Information Systems, Univ. of Electro-Communications. His research interests include On-chip interconnection networks, embedded systems, mobile computing, parallel and queue computing. He published more than fifty publications in international journals and conferences. He is a member of IEEE and a member of the design group in the International Technology Roadmap for Semiconductors (IRST).

Masahiro Sowa received the Ph.D degree in electrical and electronics engineering of Nagoya University, Aichi, Japan, in 1974. From 1974 to 1987, he was an assistant and associate professor at Gunma University, Gunma, Japan. From 1987 to 1993 he was a professor at Nagoya Institute of Technology, Nagoya, Japan. Since 1993 he has been a professor at the University of ElectroCommunications, Tokyo, Japan. His interests include parallel computer systems, parallel processors, distributed processing, computer networks, embedded computing, and queue computing. He is a member of ACM, IEEE, IPSJ, JSSST and IEICE.