A HEURISTIC FOR GLOBAL CODE MOTION

Partha Tirumalai and Meng Lee
Hewlett-Packard Laboratories
Mailstop 1U-13, 1501 Page Mill Road
Palo Alto, California 94304, U.S.A.

Abstract

Programs that are not loop intensive and which have small basic blocks present a challenge to architectures that rely on instruction level parallelism to deliver high performance. This paper presents a heuristic for moving operations across basic block boundaries based on profile information gathered from previous executions of a program. Architectural support is required for executing these operations speculatively. Performance improvements on a heapsort routine are discussed for a Very Long Instruction Word (VLIW) machine model, using two different sets of operation latencies. Experiments were done both at the assembly code level and on the intermediate representation produced by the compiler. The results show a speedup of about two over the original code after the global code motion heuristic is applied.

Keywords: Instruction level parallelism, RISC, VLIW, superscalar, execution profile, basic block, intermediate representation, compiler optimization, speculative execution, code motion.

1 Introduction

Many recently introduced high performance processors have the ability to exploit parallelism at the machine instruction level. For instance, a pipelined RISC processor can keep more than one machine instruction in execution at the same time. Superscalar and Very Long Instruction Word (VLIW) processors have the ability to fetch and execute more than one instruction in the same cycle. To effectively utilize such processors, a compiler should extract sufficient instruction level parallelism from the program to be executed.

Techniques that extract parallelism from inner loops are well known. Instructions from successive iterations of a loop can often be executed simultaneously. Software pipelining and loop unrolling are two techniques that have been effectively used to exploit the instruction level parallelism present in loops [1, 2, 3]. Programs that spend a large percentage of their execution time in inner loops can thus be accelerated significantly on processors that possess instruction level parallelism. Many scientific numerical applications fall in this class. Unfortunately, the same level of success has not yet been achieved with programs that spend much of their execution time outside parallelizable inner loops. Trace scheduling is a technique that has been used to speed up such programs [4, 5, 6]. Here, profiles of previous runs are used to predict frequently executed paths (traces) through the program. These traces are then scheduled to derive high performance. Trace scheduling has been used commercially in the Multiflow Trace computer [7]. Another technique, called superblock scheduling, also uses profiles gathered from previous executions to construct superblocks (blocks that have one entry but more than one exit point) from which instruction level parallelism is extracted [8]. Superblock scheduling is a restricted form of trace scheduling which simplifies its implementation in a compiler.

In trace and superblock scheduling the program is first segmented into traces or superblocks, and the segments are then scheduled. This paper proposes a simple heuristic for global code motion in which the program is treated as a unit. An experiment on a heapsort program is described in detail. The program was analyzed and hand-optimized at the assembly code level using a heuristic for moving operations across basic block boundaries. Performance improvements on a VLIW machine model similar to the Cydra 5 [11] were measured for two sets of latencies: the first being somewhat conservative, and the second quite aggressive. The experiment was then repeated with the code motion performed on the intermediate representation used by the Cydra 5 compiler, and allowing the back-end of the compiler to schedule the altered intermediate representation. All three experiments were executed on a simulator and on the Cydra 5 computer [11] and the results checked for correctness.
2 Description of the program

The source code for a heapsort routine obtained from [10] is shown in Fig. 1. This FORTRAN routine sorts the elements in the array RA. N represents the number of elements in the array. In the experiment, the array RA was set up with four hundred randomly generated numbers before the sort routine was called. Note the unstructured nature of this routine: there are no DO loops, but there are several branches and cycles in the control flow graph. In addition, this routine is small enough to understand with little effort and it also performs a meaningful function. We were specifically interested in experimenting with such code because of the challenge they present to machines that rely on instruction level parallelism.

It is not necessary to understand how this program works to read the rest of this paper. The program was compiled at the highest optimization level using the FORTRAN 77 production compiler on the Cydra 5. The compiler produced a control flow graph with fourteen basic blocks for the program, numbered, say, BB1 through BB14. The program was executed by the simulator and the basic block visit and transition counts collected. The control flow graph, annotated with these execution statistics, is shown in Fig. 2.

3 A heuristic for global code motion

When an operation is executed before it is known that its execution is required, that operation is said to be executed speculatively. Executing operations speculatively can reduce the time required for a computation. However, to prevent the program from reporting an error due to a speculative operation that would not have been executed in the original program, speculative operations must not
cause exceptions or side-effects. Speculative execution requires hardware support. To derive the highest performance, an architecture must permit the speculative execution of nearly all operations. For example, the Trace [7] and Cydra 5 [11] compilers simply turn off exceptions on most operations by setting a mode bit in the processor, which has the disadvantage of reducing the error reporting ability of the program. The error reporting behaviour of a program can be improved by providing non-trapping versions of those operations that can cause exceptions [12]. These non-trapping operations can be used during speculative execution. This study uses the general percolation model described in [12] which allows the speculative execution of virtually all operations other than branches and memory writes.

If an operation is moved across basic blocks, it must be speculatively executed. A simple heuristic for moving operations from one basic block to another is as follows. This is illustrated in Fig. 3.

1. Compile the program and obtain the static control flow graph.
2. Execute the program, collect the basic block visit and transition counts, and annotate the graph obtained in step 1. See Fig. 2 for an example.
3. Select a basic block BB₁ that is visited a large number of times.
4. Find a critical path in BB₁. Suppose, for example, that this path is A-B-C where A, B and C are machine operations.
5. Move the operation at the top of the critical path (A) to all predecessor basic blocks of BB₁. The moved operation (A) must now be executed speculatively; therefore, the non-trapping version corresponding to A should be used. (A must not be a memory write or some other operation that cannot be executed speculatively.) Let A* denote this speculative operation.
6. Let the result of operation A* be written to a new global register.
7. Introduce a copy from the new global register (target of A*) to the previous target of A in BB₁.
8. Change the sources of operations in BB₁ that use the result of A to the new global register.
9. Reschedule BB₁ as well as all its predecessors.
10. Estimate the execution time based on the new lengths of BB₁ and its predecessors.
11. Repeat steps 3-10. Stop when the improvement obtained becomes small.

The hope is that the length of BB₁ will decrease but the lengths of its predecessors will not change, or will not change by a significant amount. There would then be an overall improvement in execution time. In practice, moving an operation to a previous basic block bumps other operations to basic blocks farther up. However, in most cases, it is possible to obtain improvement in execution time by repeating the same process for the predecessor blocks.

Selecting an appropriate basic block is important. If the visit counts of the predecessor basic blocks are much larger than the visit count of the basic block under consideration, and if the length of the predecessors is increased by the move, this transformation could slow the program down. At each stage it is possible to estimate the performance gain by calculating the new lengths of the basic blocks after the proposed move. As the heuristic is repeatedly applied, the most frequently executed basic blocks become shorter as operations are moved outside and executed speculatively. After a number of steps, the performance improvements become small or may even decrease. The heuristic stops when it cannot find a performance increasing move.

![Figure 3. Illustration of a simple heuristic for global code motion.](image-url)
If a basic block has many critical paths each starting with a different operation, then an improvement cannot be seen until all these operations are moved up. Therefore, it is best to start by selecting basic blocks with only a few critical paths, preferably just one. In most cases, the compiler used in this study did an excellent job of scheduling individual basic blocks; virtually all the basic blocks were scheduled optimally (the scheduled length was equal to the critical path length or limited by machine resources). Therefore, performance could not be increased by simply rescheduling the instructions in existing basic blocks.

The introduction of new temporaries increases register pressure. However, it appeared in this study that the register pressure did not increase greatly for most small routines, and in the cases that it did increase significantly, the reduction in critical path lengths more than compensated for any lengthening of schedules due to the increased register pressure. There is also a concern that with the speculative execution of operations that are moved, and the introduction of new copy instructions, machine resources might be strained. However, the machine was highly under-utilized to start with and, therefore, there were plenty of empty operation slots available for placing the newly introduced operations.

4 Results

Performance improvements were studied for three cases. The first two cases involved optimizing the assembly code by hand: one for somewhat conservative latency assumptions, and the other for aggressive latency assumptions. It is interesting to compare the improvements that occur for long and short operation latencies. The third case involved optimizing the intermediate code produced by the compiler using the heuristic proposed in section 3, and feeding the optimized intermediate code to the back-end for scheduling and register allocation. This is a less tedious process than generating correct assembly code after inter-basic block code motion. However, because the Cydra 5 compiler uses many ad-hoc rules in generating code, scheduling and register allocation, the final result was not always exactly what was expected (or desired) but it was close. For each case, three performance metrics were calculated: execution time, total number of operations issued, and the average number of operations issued per cycle.

Many operations have to be moved to preceding basic blocks to achieve the final results shown below. It would be difficult (and possibly not so interesting) to describe all of them here. Rather than get into the gory details of the Cydra 5 assembly language, only the final results of the optimization process are described in sections 4.1 and 4.2.

A couple of detailed examples are given in section 4.3 because it is easiest to explain the moves using the compiler's intermediate representation. It is worth noting here that the Cydra 5 compiler is quite aggressive in scheduling basic blocks, very often achieving a schedule whose length that is nearly equal to the critical path length, or which saturates some machine resource. It also allows operations issued in one block to complete in a successor block though it does not move operations from one block to another. The code produced by the compiler at the highest optimization level is used as the basis for the performance improvements presented in this section.

4.1 Using conservative latencies

Tables I(a) and I(b) show the latencies and performance improvements for a deeply pipelined machine similar to the Cydra 5 [11]. This machine is capable of issuing up to seven operations per cycle (2 memory operations, 2 address calculation operations, 1 integer or floating point add operation, 1 integer of floating point multiply operation and a branch operation). The latencies of the operations are close to those of the Cydra 5 except for the memory read operation. The Cydra 5 does not have a data cache and its memory read latency is, therefore, much larger. Without the global code motion heuristic, less than half an operation is issued per cycle; after the heuristic is applied the performance nearly doubles. As might be expected, the long latencies make it difficult to extract a significant amount of instruction level parallelism from the program.

4.2 Using aggressive latencies

Table II(a) shows the same machine as in table I(a) but with shorter operation latencies. These latencies are representative of values seen in today's microprocessors. Prior to applying any global code motion, the compiler was able to issue just over one operation per cycle for this program. After the code motion heuristic was applied, this number jumped to three indicating a fair amount of instruction level parallelism. The net result was a speedup of 2.25. Note that this number is for a piece of code without DO loops and with many short basic blocks. This type of code challenges most ILP compilers.

Also compared was the effect of issuing multiple instructions (multiop) to that of issuing only one instruction (uniop) per cycle. This shows the potential benefit of a VLIW or superscalar processor over a processor that issues only one instruction per cycle. Without global code motion, the use of multiop had a relatively small benefit over uniop. However, after global code motion and speculative execution were added, the improvement due to multiop execution was significant.
Fig. 4 shows the execution times for the three comparisons made: uniop (without global code motion), multiop without global code motion, and multiop with global code motion. Fig. 5 shows the histogram of operations issued per cycle for the same three cases. After global code motion is done there are almost no cycles in which no operation is issued. Speculative execution might cause more operations to be issued than is necessary. This can artificially inflate the operations issued per cycle. To be fair, the useful operations issued per cycle was also computed. Table III shows these numbers.

Table I(a). Machine model with long latencies.

<table>
<thead>
<tr>
<th>Unit</th>
<th>#</th>
<th>Lat.</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>1</td>
<td>4</td>
<td>Int and flt point adds, compares</td>
</tr>
<tr>
<td>Multiply</td>
<td>1</td>
<td>4</td>
<td>Int and flt point mults</td>
</tr>
<tr>
<td>Memory</td>
<td>2</td>
<td>4(rd) 1(wr)</td>
<td>Memory reads and writes</td>
</tr>
<tr>
<td>Address</td>
<td>2</td>
<td>2</td>
<td>Addr calculation adds and mults</td>
</tr>
<tr>
<td>Control</td>
<td>1</td>
<td>3</td>
<td>Branches</td>
</tr>
</tbody>
</table>

Table I(b). Performance improvement with the machine in table I(a).

<table>
<thead>
<tr>
<th>Metric</th>
<th>Orig. Program</th>
<th>After Optimizn</th>
<th>% gained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execn. cycles</td>
<td>139894</td>
<td>69478</td>
<td>101.4</td>
</tr>
<tr>
<td># ops issued</td>
<td>69248</td>
<td>78489</td>
<td>-13.3</td>
</tr>
<tr>
<td>Ops per cycle</td>
<td>0.49</td>
<td>1.13</td>
<td>130.6</td>
</tr>
</tbody>
</table>

Table II(a). Machine model with short latencies.

<table>
<thead>
<tr>
<th>Unit</th>
<th>#</th>
<th>Lat.</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>Int and flt point adds, compares</td>
</tr>
<tr>
<td>Multiply</td>
<td>1</td>
<td>2</td>
<td>Int and flt point mults</td>
</tr>
<tr>
<td>Memory</td>
<td>2</td>
<td>2(rd) 1(wr)</td>
<td>Memory reads and writes</td>
</tr>
<tr>
<td>Address</td>
<td>2</td>
<td>1</td>
<td>Addr calculation adds and mults</td>
</tr>
<tr>
<td>Control</td>
<td>1</td>
<td>2</td>
<td>Branches</td>
</tr>
</tbody>
</table>

Table II(b). Performance improvement with the machine in table II(a).

<table>
<thead>
<tr>
<th>Metric</th>
<th>Orig. Program</th>
<th>After Optimizn</th>
<th>% gained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execn. cycles</td>
<td>63831</td>
<td>28356</td>
<td>125.1</td>
</tr>
<tr>
<td># ops issued</td>
<td>69223</td>
<td>85347</td>
<td>-23.3</td>
</tr>
<tr>
<td>Ops per cycle</td>
<td>1.08</td>
<td>3.01</td>
<td>178.7</td>
</tr>
</tbody>
</table>

Figure 4. Execution times with uniop and multiop with and without code motion.

Figure 5. Histogram of number of operations issued per cycle.

4.3 Experiment done using the compiler's intermediate representation

The code motion heuristic was applied to the intermediate representation (IR) in the compiler. The control flow graph for the program is shown in Fig. 2. A couple of moves are discussed in detail leaving the progress of such transformations to the reader's imagination.

Consider basic blocks BB7, BB8 and BB9. The IR for these three blocks as obtained from the compiler is shown in Fig. 6(a). The IR dumped by the compiler has a lot of information about the basic blocks eg. the internal
number, the type, any labels, nesting level, internal flags etc. Much of this information can be disregarded for this paper; attention need be paid only to the operations listed inside each block. The tn numbers are temporary names used prior to register allocation. Notice from the control flow graph (Fig. 2) that BB9 has a unique immediate predecessor which is BB8. Similarly, the unique immediate predecessor of BB8 is BB7. BB9 contains two READ operations which feed into an flt operation. This creates two READ-flt-branch critical paths in BB9. We shall therefore move the two READ operations in BB9 to BB8. The ampy and Aadd in BB8 generate the address for use by the two READ operations now in BB8. To maximize the effect of having moved the READs up, we shall move the ampy and Aadd operations from BB8 to BB7. The modified IR is shown in Fig. 6(b).

Once again consider the same three basic blocks. The critical path in BB9 is now flt-branch. In an effort to reduce this critical path, the flt operation may be moved to BB8. The critical path in BB8 then becomes READ-flt. Therefore, the two READs in BB8 may now be moved to BB7. Fig. 6(c) shows the IR after these moves have been performed. Note that all that now remains in BB9 is a conditional branch.

**Figure 6(a).** IR for basic blocks BB7, BB8 and BB9 as obtained from the compiler.

**Figure 6(b).** IR for basic blocks BB7, BB8 and BB9 after some operations are moved up.

**Figure 6(c).** IR for basic blocks BB7, BB8 and BB9 after more operations are moved up.
After the first few moves, the improvement resulting from each subsequent move becomes smaller, while the move itself becomes more difficult. In the above cases, for example, there was only one unique predecessor; if there are multiple predecessors, an operation has to be placed in each of them. The moved operation will bump operations farther up in something like a chain reaction. After several moves on the IR a speedup of 1.5 over the original program was achieved.

5 Conclusion

A simple heuristic for global code motion, that is, the movement of operations across basic block boundaries, has been proposed, and its performance on a short heapsort routine has been studied. This routine has branches and cycles in the control graph, and is typical of code that presents a challenge to compilers attempting to extract instruction level parallelism. Using the heuristic for moving operations across basic blocks, the routine was hand optimized at the assembly code level and scheduled. Performance metrics were gathered by running the result on a simulator. Two sets of latencies were used: one somewhat conservative, and the second more aggressive. The experiment was then repeated at the intermediate code level. The results show significant speedups for all three cases - a speedup of 2 for both long and short latencies in the assembly level experiments, and a speedup of 1.5 for the experiment done at the intermediate level.

References


