Performance of Dynamically Scheduling VLIW Instructions

Sunghyun Jee  
Chonan College in Foreign Studies  
Department of Computer Information  
Chonan, Chungnam, South Korea  
jsh@ccfs.ac.kr

Kannappan Palaniappan  
University of Missouri-Columbia  
Department of Computer Science  
Missouri, Columbia, U.S.A.  
palani@cecs.missouri.edu

Abstract

This paper evaluates performance of the Dynamically Instruction Scheduled VLIW (DISVLIW) processor architecture. The DISVLIW processor architecture is designed for dynamically scheduling VLIW instructions using dependency information. Features such as explicit parallelism, balanced scheduling effort, and dynamic scheduling of VLIW instructions can be used to provide a sound structure for supercomputing. We simulate the DISVLIW processor architecture and show that the DISVLIW processor performs significantly better than the VLIW processor across various numerical benchmark applications.

1. Introduction

Recent high performance processors have depended on Instruction Level Parallelism (ILP) to achieve high execution speed [1~8]. Very Long Instruction Word (VLIW) is one particular style of processor design that tries to achieve high levels of ILP by executing long instructions composed of multiple instructions [2,3,4,6,7]. In the VLIW processors, the compiler has complete responsibility for creating a package of operations that can be simultaneously issued. The VLIW processors do not dynamically make any decisions about multiple instruction issue and scheduling, and thus they are simple and fast. However, the VLIW processor has performance bottlenecks due to static instruction scheduling and the unoptimized large object code containing a number of NOPs (No OPerations) and LNOPs (Long NOPs), where the LNOP means a long instruction that has only NOPs.

Superscalar VLIW (SVLIW) is the improving style of VLIW processor design that tries to execute object code constructed by removing all LNOPs from VLIW code [6,7]. The SVLIW processor also has a performance limitation similar to the VLIW processor due to static scheduling.

Another technique to overcome the VLIW performance limitation problem is to allow the compiler to exploit high ILP using Explicit Parallel Instruction Computing (EPIC) [5]. The basic EPIC principle is that the compiler should be able to indicate the inherent parallelism of programs explicitly in the instruction sequence. The IA-64 architecture allows the compiler to exploit high ILP using EPIC techniques [5]. IA-64 processor architecture implementing this concept is the processor architecture where the compiler is responsible for efficiently exploiting the available ILP and keeps the executions busy. Instead of the merits, the IA-64 processor has performance limitations due to static instruction scheduling and the difficulty of complicated compiler design. In order to overcome current performance bottlenecks in modern architectures, a processor architecture that satisfies the following criteria is required: (1) balanced scheduling effort between compile time and run time, (2) dynamic instruction scheduling, and (3) reducing the size of object code.

This paper evaluates the ILP processor architecture called Dynamically Instruction Scheduled VLIW (DISVLIW) that achieves these goals. Figure 1 shows a simple diagram of the DISVLIW processor architecture. The DISVLIW processor is aimed specially at dynamic scheduling VLIW instructions with dependency information. In order to schedule instructions dynamically, the DISVLIW processor uses functional unit and dynamic scheduler pairs. Each dynamic scheduler decides to issue the next instruction to the associated functional unit, or to stall the functional unit due to possible resource collisions or data dependencies among instructions per every cycle.

Such designing of the DISVLIW processor results in three positive characteristics. First, the DISVLIW compiler can maximize the use of current VLIW compiler techniques to generate object code for DISVLIW. Second, the DISVLIW processor can get higher cache effects, high cache hit ratio or reduced instruction fetch time, due to the reduced object code sizes.

Figure 1. DISVLIW processor architecture
Third, the DISVLIW processor can dynamically schedule each instruction using dependency information in the instruction. Such features can reduce the total number of execution cycles of the DISVLIW processor better than those of other ILP processors that statically schedule long instructions.

2. DISVLIW compiler design

DISVLIW code generation can be subdivided into two phases. The first phase, VLIW code generation, contains two schedulers: the software pipeliner for targeted cyclic regions and the global code scheduler for all remaining regions [2,3,4,6,7]. After the first phase, the result is VLIW code composed by a sequence of long instructions so that each long instruction can be executed per clock cycle without violating data dependences or resource constraints. We can maximize the use of the VLIW compiler techniques for the first phase. In the second phase, dependency information insertion, the VLIW code compactor compacts the VLIW code by removing nearly all LNOPs and NOPs from the generated VLIW code, and the dependency inserter then inserts dependency information into each instruction in the compacted VLIW code. It is necessary to express explicit dependencies of long instruction so that each instruction can be executed per clock cycle without violating data dependences or resource constraints. We can maximize the use of the VLIW compiler techniques for the first phase. In the second phase, dependency information insertion, the VLIW code compactor compacts the VLIW code by removing nearly all LNOPs and NOPs from the generated VLIW code, and the dependency inserter then inserts dependency information into each instruction in the compacted VLIW code. It is necessary to express explicit parallelism within the DISVLIW object code for synchronization. The result of both phases represents the final DISVLIW code composed of long instructions. Each long instruction has multiple instructions that may depend on each other due to data dependencies or resource collisions [6,7].

Figure 2 shows the example of the DISVLIW code structure (N=4), where N equals the number of instructions within one long instruction. Each instruction format consists of an instruction Iij, pre-dependency Dpre, and post-dependency Dpost. Iij refers to the jth (j=1,...,N) instruction within the ith (i=1,...,M) long instruction. Dpre provides information about functional units executing prior instructions that have dependencies with Iij. Dpost provides information about functional units that will execute subsequent instructions that depend on Iij. Dpre and Dpost are individually composed of a bit vector that has (N-1) bits.

In order to store the information as bit vector, the compiler allocates one bit for every other functional unit. If Iij depends on a prior instruction Iik (k<j if l=i;k=1,...,n if l<i) being executed by functional unit Fk, the bit designating Fk in the Dpre is set to 1. Otherwise, it is set to zero.

3. DISVLIW processor design

The block diagram of the DISVLIW processor is shown in Figure 1. Each IQ stores an instruction (separated in long instruction) in its own tail, and individually provides an instruction and dependency information stored in its own head to decode unit and the associated DS. IQs are placed in front of each FU. It seems like instructions within IQ issue in order, but instructions among IQs slip with respect to each other. Each DC saves Dpost of executed instructions on the associated FU. Using the DC values, each DS dynamically decides whether to assign the next instruction to the associated FU, or to stall the FU due to resource collisions or data dependencies. Dynamic scheduling allows instructions in different IQs (i.e. different FUs) are synchronized by having DC at each FU. If there are N FUs, then each FU has a DC composed of N-1 counters, 1 counter for every other FU. The processor also utilizes the BTB structure for branch prediction [7].

![Figure 2. Compiler phases and long instruction format](image)

![Figure 3. Dynamic scheduler units](image)

Each DS in Figure 3 compares Dpre of the next instruction to counter values in the associated DC per a cycle. If ith bit in Dpre, Dprei is set to 1, the DS checks Ci in the corresponding location in the DC. If Ci is 0, it means that the execution of prior dependent instruction hasn’t finished. That is, di returns zero. Otherwise, the execution of prior dependent instruction has finished. That is, di returns 1. In order to assign the next instruction to the associated FU, the DS should confirm that the execution of all prior dependent instructions is finished (all of di return 1).

\[
\text{check signal of } D^i = \bar{d}_i \land d_0 \land \ldots \land d_{i-1} \land d_{i+1} \land \ldots \land d_{N-1} = (D^0 \lor D^1 \lor \ldots \lor D^N)
\]

The equation above represents the logic for DSi necessary to check the dependency between instructions for dynamic scheduling: If check signal is 1, the DS assigns the next instruction to FUi. Otherwise, DSi have waited until the
data dependencies or resource collisions between instructions are solved. The operator \(\otimes\) means binary vector comparison. \(D_1^{i-1} \otimes C_1\) evaluates to true (That is, 1) if both of \(D_1^{i-1}\) and \(C_1\) are 0 or \(C_1\) is greater than \(D_1^{i-1}\) (when \(D_1^{i-1}\) is set to 1). The operator \(\cdot\) means logic and. After assigns the next instruction, the DS simultaneously decrements the counter values in corresponding location in its DC using the set bits in given Dpost. It is necessary to clear the Dpost of the prior instructions from the DC before next execution.

4. Dynamic scheduling strategies

Figure 4 shows dynamic execution examples of the DISVLIW code. We assume that the processor has three untyped functional units. The DISVLIW compiler generates the DISVLIW code from the assembly code in Figure 4(a). From Figure 4(b), we know that instruction sub.d within the 1st long instruction depends on previous instruction lwc1 executed by FU1 since the first bit in Dpre is set to 1. Figure 4(c) shows the changes of DC values according to the execution of DISVLIW code. FU0 first executes instruction lwc1 since Dpre of lwc1 is 00, and simultaneously increments the first counters (indicating FU0) in the DC1 and DC2 because Dpost of lwc1 is 11. Then, FU1 and FU2 individually check Dpre of instruction sub.d, add.d and the counter values in its DC1 and DC2. If both of them are greater than 0, FU1 and FU2 simultaneously begin the execution of instructions sub.d and add.d. Then, FU1 and FU2 simultaneously decrements the first counter value in its DC1 and DC2 using set bits in Dpre. It is necessary to clear Dpost of lwc1 from each DC before the execution of FU1 and FU2. Figure 4(d) demonstrates the execution steps of instructions in the DISVLIW code.

\[\begin{align*}
lwc1 & \quad $16,5,1,0 \\
sub.d & \quad $5,15,6,5,10 \\
add.d & \quad $7,16,6,5,4 \\
add.d & \quad $8,15,6,5,6 \\
mul.d & \quad $10,17,5,11 \\
swc1 & \quad $8,8,2,522
\end{align*}\]

Figure 4. Example of dynamic scheduling

5. Experiment

Using a simulator testbed, we evaluate the performances of the VLIW, the SVLIW and the DISVLIW processor architectures. The simulator starts with the MIPS assembler, generating MIPS assembly code by compiling C-language benchmarks. Next, the macro expander inputs the MIPS assembly code while simultaneously expanding macros. The Macro expander then passes the assembly code to each parallelizer. Three parallelizers, each of which is associated with a unique processor, are designed with the ability to exploit ILP across basic blocks using compile techniques such as register renaming, branch prediction, function inlining and loop unrolling [6,7]. From now on, VLIWC, VLIWS, and VLIWDIS individually mean VLIW, SVLIW, and DISVLIW code, respectively.

For these experiments, processor speedups are calculated by dividing the total number of execution cycles of the VLIW processor by the total number of cycles of the DISVLIW or the SVLIW processor. In the Table 1, the fixed parameters and the variable parameters are also shown. Except when stated otherwise, the default values were used in the simulations.

\begin{table}[h]
\centering
\caption{Input parameters}
\begin{tabular}{|c|c|}
\hline
Fixed Parameters & \\
\hline
Processor pipeline & Four-stage(F,D,EX, WB) \\
Decoded instruction size & 4 bytes \\
Integer instruction latency & 1 cycle \\
Floating point instruction latency & 1-32 cycle \\
Data cache size & Perfect(no miss penalty) \\
cache mapping method & Direct mapped \\
cache replacement policy & LRU(Least Recently Used) \\
\hline
Variable Parameters & \\
\hline
A number of integer unit / floating-point unit & 2/2 \\
next long instruction miss penalty & 4 \\
\hline
\end{tabular}
\end{table}

We chose numerical benchmarks that have a high proportion of floating-point instructions because the DISVLIW processor is more effective given dynamic instruction scheduling and reduced object code size. Even though VLIWDIS contains many bits of dependency information, the result indicates that VLIWDIS averages 45% smaller than VLIWC and is almost the same size as VLIWS [7].

Figure 5 illustrates the impact of cache size on speedups of the DISVLIW processor with respect to both the SVLIW and VLIW processors. We varied the instruction cache size from 16k bytes to 32k bytes to compare performance according to changes in cache size. The speedups of the DISVLIW and the SVLIW processors were measured relative to the VLIW processor regardless of cache size. In this experiment, we also reduced the number of loop iterations in each benchmark to reduce simulation duration. These results indicate that larger cache sizes result in smaller speedup differences between the VLIW and DISVLIW processors. At smaller cache sizes, the VLIW’s performance is slower due to higher cache miss rates. Unlike the VLIW, the DISVLIW’s performance is not as sensitive to cache size.
due to its smaller object code. But as cache size increases, performance difference decreases and the VLIW’s performance approaches that of the DISVLIW. Yet, even assuming perfect cache, the DISVLIW is still faster than the VLIW’s.

Figure 6. Speedup according to scheduling strategies

Figure 6 shows the speedup of the DISVLIW processor over the VLIW (or the SVLIW) processor using different scheduling strategies. In order to evaluate scheduling performance only, we ignore cache effects such as cache miss rates. We assume that an instruction cache size is perfect (no miss penalty). Figure 6 illustrates that even though we assume a cache with a zero miss rate, the DISVLIW’s performance is still 9%-15% higher than that of the VLIW processor regardless of benchmark application. We have the DISVLIW’s scheduling strategies to thank for this speedup. This scheduling decreases the waiting time to process a set of long instructions when compared to the VLIW and SVLIW processors. By contrast, the VLIW and the SVLIW processor can’t execute pending long instructions until the execution of all instructions in the previous long instruction finishes. In Figure 6, the SVLIW processor shows same performance when compared to the VLIW processor.

6. Conclusion

The DISVLIW processor is a hybrid architecture that has inherited features as ILP exploitation at compile-time of the VLIW processor and dynamic scheduling at run-time of the superscalar processor. The experimental evaluations have shown that the DISVLIW processor achieves a high speedup over the VLIW and the SVLIW processors for a wide range of cache sizes and across various numerical benchmarks. These performance gains of the DISVLIW processor result from dynamic instruction scheduling and size reduction of object code. The DISVLIW processor architecture opens several new avenues of research. Optimization of dependency information within object code and DISVLIW compilers are just a few examples that will be investigated in future work.

7. References