Performance Comparison of ILP Machines with Cycle Time Evaluation

Tetsuya Hara, Hideki Ando, Chikako Nakanishi, Masao Nakaya

System LSI Laboratory
Mitsubishi Electric Corporation
4-1 Mizuhara, Itami, Hyogo, 664 Japan

Abstract

Many studies have investigated performance improvement through exploiting instruction-level parallelism (ILP) with a particular architecture. Unfortunately, these studies indicate performance improvement using the number of cycles that are required to execute a program, but do not quantitatively estimate the penalty imposed on the cycle time from the architecture. Since the performance of a microprocessor must be measured by its execution time, a cycle time evaluation is required as well as a cycle count speedup evaluation. Currently, superscalar machines are widely accepted as the machines which achieve the highest performance. On the other hand, because of hardware simplicity and instruction scheduling sophistication, there is a perception that the next generation of microprocessors will be implemented with a VLIW architecture. A simple VLIW machine, however, has a serious weakness regarding speculative execution. Thus, it is a question whether a simple VLIW machine really outperforms a superscalar machine. We recently proposed a mechanism called *predicating* that supports speculative execution for the VLIW machine, and showed a significant cycle count speedup over a scalar machine. Although the mechanism is simple, it is unknown how much it imposes a penalty on the cycle time, and how much the performance is improved as a result. This paper evaluates both the cycle count speedup and the cycle time for three ILP machines: a superscalar machine, a simple VLIW machine, and the VLIW machine with predicating. The evaluation results show that the simple VLIW machine slightly outperforms the superscalar machine, while the VLIW machine with predicating achieves a significant speedup of 1.41x over the superscalar machine.

1 Introduction

Current high-end microprocessors exhibit good performance through superscalar techniques [9][10][12]. A superscalar machine dynamically schedules instructions from an instruction window on a predicted control path to exploit instruction-level parallelism (ILP). *Speculative execution* is essential for instruction scheduling so that the scheduler can exploit ILP beyond basic block boundaries. Although the mechanism for dynamic scheduling with speculative execution reduces the number of cycles for program execution, the complicated hardware required for the mechanism imposes a cycle time penalty.

Very long instruction word (VLIW) machines, on the other hand, only require simple hardware since the complexity of instruction scheduling is transferred to the compiler. Therefore, there is only a small cycle time penalty. A compiler is fundamentally better at instruction scheduling than a dynamic scheduler because of its large instruction window and sophisticated algorithms. Thus, VLIW machines have the potential to outperform superscalar machines. Yet, a simple VLIW has a serious weakness in speculative execution. That is, the compiler has only a limited ability to handle the side effects of speculative execution. This limitation greatly reduces the amount of ILP available for the compiler to exploit. Thus, it is a question whether a simple VLIW machine really outperforms a superscalar machine.

Recent VLIW studies proposed hardware mechanisms to remove the restrictions imposed on compiler's scheduling (e.g. *guarding* [3] and *boosting* [11]). We recently proposed a mechanism called *predicating* [2] which provides the compiler with unconstrained speculative code motions. Although that paper reported great ILP improvement through the mechanism and the simplicity of the mechanism, it is unknown how much the hardware mechanism imposes a cycle time penalty, and how much the performance is improved as a result.

This paper answers these questions. That is, we estimate the performance improvement of a superscalar machine, a simple VLIW machine, and our VLIW machine with predicating over a scalar machine by evaluating both the cycle count and the cycle time. We have built an instruction scheduler and simulators for the cycle count evaluation, and have designed critical hardware for the cycle time evaluation. Section 2 describes the three ILP architectures we evaluated. Section 3 discusses the complexity of each architecture. Section 4 describes evaluated machine models. Section 5 shows evaluation results of cycle counts, cycle time, and resultant performance. Finally, Section 6 concludes this paper.

2 ILP Machine Architectures

We evaluated three ILP machines: a simple VLIW machine, a VLIW machine with predicating, and a superscalar machine. The simple VLIW machine we evaluated exists in an extreme side in ILP machines in terms of simplicity. This machine may run with the fastest clock rate. Although the compiler schedules instructions, the amount of exploitable ILP is severely limited because of the limited hardware support for speculative execution. Thus, of the

ISCA'96 5/96 PA. USA

ILP machines, it may consume the largest number of cycles to execute a program. In contrast, the superscalar machine exists in the other extreme side from its complexity. This machine may run with the slowest clock rate. Yet, the machine is strong in exploiting ILP since the hardware supports instruction scheduling and speculative execution sufficiently. While these two machines have weakness in either the clock rate or the ability to exploit ILP, we expect that a VLIW machine with predicating would be strong in both aspects. In this section, the three ILP machine architectures are described.

2.1 Simple VLIW Machine

Our simple VLIW machine simply duplicates hardware resources such as the function unit, the register file port, and the D-cache port for multiple instruction execution. Although the machine may run with the fastest clock rate, the amount of exploitable ILP is limited since it has the smallest amount of hardware support for speculative execution. When the compiler speculatively moves an instruction past branches, the compiler must ensure that the speculative instruction does not overwrite a register whose value may later be referred by other instructions; otherwise, the program semantics will be changed. For this purpose, the compiler uses register renaming. That is, the compiler assigns an empty register as the destination of the speculatively moved instruction, and later copies the value of the newly assigned register to the original destination register. This renaming technique, however, cannot be applied to preserve a value in a memory location.

The compiler must be further conservative in unsafe code motions. A code motion is said to be *unsafe* if the moved instruction may cause an exception. If we allow the compiler to move unsafe instructions speculatively, immediate handling the exception may incorrectly terminate the program or decrease performance because it is unknown whether the handling is necessary or not. Thus, only those instructions that will not cause an exception are allowed to move speculatively.

To reduce the restrictions on speculative code motions, our simple VLIW machine assists the compiler with a pipeline squashing mechanism. That is, the hardware squashes a speculative instruction in the pipeline if the control is not transferred to the moving path of that instruction. To use this mechanism, the compiler must ensure that speculative instructions that write to a memory location and/or those that might cause an exception can be squashed in the pipeline before actual write occurs. Since this is a simple extension of a squashing branch, we believe that the mechanism never affects the cycle time.

2.2 VLIW Machine with Predicating

Predicating is a mechanism that provides the compiler with unconstrained speculative code motions. While the speculative code motions are limited in the simple VLIW machine, any type of speculative code motion is allowed with predicating. That is, predicating allows the compiler to speculatively move instructions that may overwrite a location with a live value and/or cause exceptions. In predicating, an instruction is predicated with its control-dependent branch conditions. The side effects caused by a speculative execution are buffered with its predicate. This buffered predicate efficiently commits or squashes the side effects and appropriately handles exceptions caused by the speculative execution. We will now briefly describe the predicating architecture. A comprehensive description is found in [2].

An instruction is predicated with the following format:

```
predicate ? operation
```

where the predicate is a Boolean expression of the branch conditions that the operation depends on. Semantically, the result of the execution specified by the operation part is valid if and only if the commit condition specified by the predicate part is true. In the following example, the result of the add operation is valid if branch conditions `c1` and `c2` are both true.

```
c1&c2 ? r1 = r2 + r3
```

The predicating machine has two states: a *sequential state* and a *speculative state*. The sequential state consists of the results of instructions whose control dependencies are all resolved; the speculative state consists of the results of instructions which have at least one unresolved control dependence. The result in the speculative state has its predicate for a later commit to the sequential state.

At the issue point, the predicate of the instruction is evaluated. If the predicate evaluates to true, the instruction is executed and writes the result into the sequential state. This is non-speculative execution. If the predicate evaluates to false, the instruction is simply squashed because the execution is no longer necessary. In the last case, at least one branch condition of the predicate is not determined, that is, the value of the predicate is not *specified*. In this case, the instruction is executed, but writes the result into the speculative state. Unlike non-speculative writes, these speculative writes label the result with the predicate for a later commit. The predicate of the result in the speculative state is evaluated every cycle by referring to the branch conditions. During cycles in which the predicate is unspecified, the result is held continuously. If the predicate evaluates to true, the result is committed; if the predicate evaluates to false, the result is squashed.

Figure 1 illustrates the organization of the predicating architecture with an *N*-instruction issue. The architecture differentiates itself from our simple VLIW machine by including a control path, a predicated register file, and a predicated store buffer. The control path evaluates the predicate of instructions executed in the datapath by referring to the branch conditions stored in the condition code register (CCR). It squashes instructions if their predicate evaluates to false. To reduce the complexity of predicate evaluation, we limit the expression of the predicate to ANDed operation with the negation of branch conditions. We then encode the predicate in a vector where each entry is associated with a branch condition. Each entry has a Boolean value which is necessary to let the predicate be true. Thus, merely a match operation between the encoded predicate and the content in CCR evaluates the predicate.

The register file consists of sequential registers and shadow registers. The sequential register contains the sequential state; the shadow register contains the speculative state. If the predicate evaluates to true in the control path, the result is written into the sequential register. If the predicate evaluates to unspecified, the result is written into the shadow register. We provide only a single shadow register for each sequential register. This may cause a storage conflict among writes of the results with different predicates, but this conflict has little impact on performance because it rarely occurs[1]. When an instruction fetches a register value, the instruc-

1. Our evaluation shows that a single shadow register model degrades performance by just 0-1% below an infinite shadow register model.

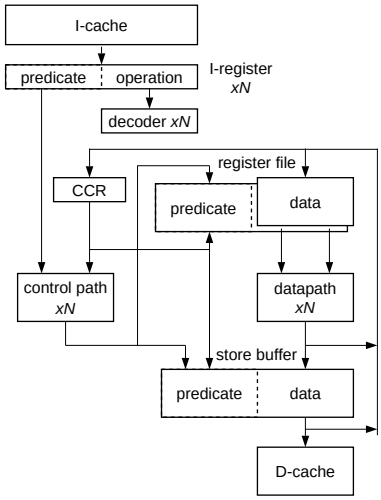


Figure 1: Organization of the predicating architecture (*N*-instruction issue)

tion explicitly specifies which register should be referred to. For a speculative state write, the predicate of the instruction is also written into the predicate storage associated with the destination register for a later commit.

Figure 2 shows the configuration of the predicted register file. Each entry has two storages for sequential and speculative values. Flag W indicates which storage stores a speculative value. Flag V indicates that the speculative value is valid. Flag E indicates that there exists an outstanding speculative exception. Each entry of the register file has dedicated hardware which evaluates the predicate stored in its entry. The commit action is done by updating flags V and W according to the result of the evaluation. If the predicate evaluates to true, flag W is flipped and the flag V is reset to commit the speculative value. If the predicate evaluates to false, flag V is reset to squash the speculative value.

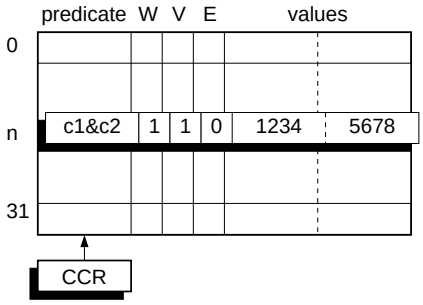


Figure 2: Predicated register file

2.3 Superscalar Machine

The superscalar machine we evaluated is a dynamically-scheduled, out-of-order execution machine with support for speculative execution. Register renaming is used to avoid output and anti-dependences. Reservation stations [12] are provided for each function unit to check operand availability and issue instructions in parallel. A reorder buffer [10] is used to maintain the correct machine state. With these mechanisms in conjunction with dynamic branch prediction, the superscalar machine fetches a single instruction stream and schedules instructions in the stream so that pipelines never stall.

3 Complexity

The sources of complexity in ILP machines are parallel execution, instruction scheduling, and speculative execution. The complexity caused by these sources can impose a cycle time penalty. In this section, the complexity of each ILP machine is discussed.

3.1 Parallel Execution

Register File. Parallel execution requires multiple ports for the register file to provide operands for parallel execution. The multiple ports simply make the area for the register file large. Since driving the word lines and bit lines that traverse the register file is critical for register file access, the area itself can impose a cycle time penalty. Furthermore, each data storage cell must drive multiple bit lines for parallel operand supply. This also makes the register read slow. Since all ILP machines we have evaluated require multiple ports for registers to supply operands for multiple decoding instructions, this complexity is commonly imposed on all ILP machines.

Operand Bypassing. Parallel execution also adds complexity for the operand bypassing. This complexity grows rapidly as the degree of parallelism increases. There are two types of complexity regarding the operand bypassing. The first complexity is regarded as the *bypass control* which determines which result in the pipeline should be forwarded or if no forwarding is required. This complexity increases in ILP machines since the number of results in the pipelines is increased. The bypass control for a scalar machine with a typical five-stage pipeline is required to compare only two destination register numbers of the currently executing instructions with each source register number of the currently decoding instruction, while the bypass control for a four-issue VLIW machine with the same pipeline organization is required to compare eight destination register numbers of the currently executing instructions with each source register number of the currently decoding instructions. Therefore, a scalar machine only requires a total of four comparisons, while a four-issue VLIW machine requires a total of 64 comparisons. Furthermore, finding the latest result in the pipeline adds extra complexity due to prioritized selection. For the superscalar machine we evaluated, the results are forwarded to all the reservation station entries in addition to the decoding instructions. Tag compare circuits are required for all the results in each reservation station entry. Consequently, the complexity grows as the number of reservation station entries increases.

The second complexity in the operand bypassing concerns the *result drive*. This complexity is increased in ILP machines since the number of destinations to which the result must be forwarded is increased. The number of forwarding destinations in a scalar machine is only two, while it is eight in a four-issue VLIW machine. Thus, the signal delay time for the result drive is increased. For a superscalar machine, this problem is quite serious since the results must be forwarded to all of reservation station entries in addition to the decoding instructions. For example, the four-instruction decode superscalar machine we actually evaluated has a total of 48 destinations (40 operands in reservation stations and 8 operands of decoding instructions). The fan-out of the result driver is 24 times larger than that in the scalar machine, or 6 times larger than that in the four-issue VLIW machine.

In general, the result latency of ALU operations limits the cycle time. Since the signal delay caused by the result drive is included in

the result latency, this complexity directly affects the cycle time. This is in contrast to some other types of complexity. The penalty caused by some complexities can be less sensitive to the cycle time by using pipeline stage allocation. That is, we can keep the cycle time fast by allocating more pipeline stages for more complex work. As a simple example, half of a single pipeline stage is allocated for the register read in the MIPS R3000 architecture [5], while we can allocate all of a single pipeline stage for the slow register read in the VLIW machine. The modification to deeper pipelines causes other types of penalty. In the previous example, a bigger branch penalty is imposed. Yet, this penalty migration is justified by the following considerations: first, an aggressive branch prediction mechanism can reduce the branch penalty. Second, a cycle time penalty is imposed every cycle, while a branch misprediction penalty is only imposed when branch prediction misses. On the other hand, allocating multiple pipeline stages for the ALU operations is not allowed because the data dependences on frequently executed ALU instructions unacceptably increase the number of cycles to execute a program. Therefore, the result latency of ALU operations is the lower limit of the cycle time.

Branch Execution. Most scalar machines use a delayed branch for branching. This is as effective as dynamic branch prediction in reducing the branch penalty for a machine with a single branch-delay slot since the compiler can fill it with an effective instruction [7]. The effectiveness is, however, reduced for a VLIW machine with multiple branch-delay slots; it is obviously hard to find instructions to fill the multiple branch-delay slots. Thus, dynamic branch prediction with a branch target buffer [6] is used for the VLIW machines we evaluated. For superscalar machines, using dynamic branch prediction for speculative execution is well accepted. Dynamic branch prediction may impose a cycle time penalty because the next PC of the current cycle must be determined after the prediction hit check; if a branch is found to be mispredicted, then the correct address must be sent as early as possible to keep the branch misprediction penalty small. This extra work increases the branch execution time, and thus it may affect the cycle time.

We allow parallel execution of multiple branch instructions in the VLIW machines we evaluated. This is necessary to reduce cycle counts because branch instructions are frequently executed in non-numerical applications. In the simple VLIW machine, the branch hardware gives priority to the branch instruction whose instruction address is small if multiple branches are taken. The compiler ensures correctness when it schedules multiple branch instructions. This function increases the complexity of the next PC selection since the next PC must be selected from the target addresses of multiple branch instructions with priority. For the VLIW machine with predicating, this prioritized selection is not necessary because the predicate of the branch instruction has precise control dependence. That is, the predicating architecture guarantees that only a single branch instruction is committed in a single cycle even if multiple branch instructions are executed.

3.2 Instruction Scheduling

The complexity of instruction scheduling is transferred to the compiler in VLIW machines. Thus, no penalty is imposed on the hardware. In the superscalar machine, instruction scheduling is done by the hardware. The dynamic instruction scheduling is quite complex. The tasks include a data dependence check, a tag allocation, and an instruction issue from a reservation station. These tasks must be done after an instruction fetch but before execution, i.e., in the decode stage of a typical five-stage pipeline. Our preliminary evaluation shows that these tasks (with the reorder buffer allocation and reorder buffer read described later) should be decomposed to at least two stages for a fast clock rate; otherwise, the complexity unacceptably lengthens the cycle time. We allocate the data dependence check and the tag allocation to the first decode stage, and allocate the instruction issue to the second decode stage. The instruction issue is still complex since it must be sequentialized after an operand availability check. That is, the operand tags in a reservation station entry are first compared with the result tags, and then a set of ready instructions whose operands are all available is determined. The instruction issue logic then determines which instruction should be issued to the function unit from the instructions ready. These two tasks must be sequentialized and cannot be separated to keep the result latency a single cycle. This may affect the cycle time.

We ignore the complexity of other tasks in instruction scheduling by allocating a full single cycle to them. Two-stage allocation to instruction scheduling imposes a three-cycle branch misprediction penalty. Since at most a single cycle is enough for the instruction decode stage in the VLIW machines (discussed later), a bigger branch misprediction penalty is imposed on the superscalar machine.

3.3 Speculative Execution

The simple VLIW machine only relies on pipeline squashing for speculative execution. This mechanism is simple enough never to affect the cycle time. The predicated register file to support speculative execution in the VLIW with predicating adds complexity. As described in Section 2.2, each register file entry has two data storage locations. One of the registers is read according to flag W and the *shadow register specifier*; flag W indicates which location is currently a shadow register, and the shadow register specifier indicates which register of the shadow or sequential register the instruction requires to read. A *location selector* selects one of two locations with these two signals. Figure 3 illustrates the hardware for this function. Since the location selector is inserted after the address decoder, it makes the register read from the address decode slow. Another critical path concerns the path which includes the predicate evaluation. Since flag W may be modified depending on the predicate evaluation result, the signal path including the predicate evaluation, flag modification, location selection, and then data read are also critical for the register read.

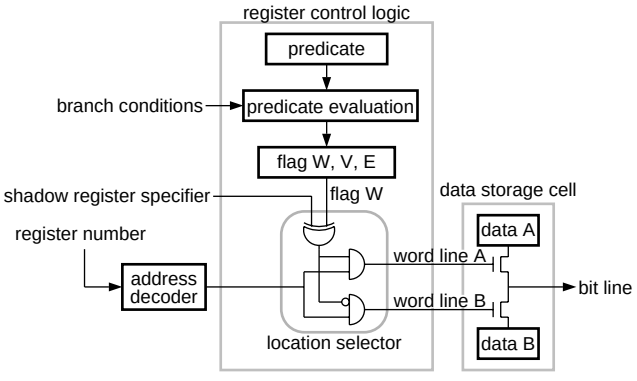


Figure 3: Control logic in the predicated register file

For speculative execution in the superscalar machine, a reorder buffer entry allocation, a reorder buffer read, and instruction squashing add complexity. We ignore the complexity of the reorder buffer entry allocation and the reorder buffer read by allocating these tasks along with the data dependence check and the tag allocation to a single cycle as described above. Instruction squashing is obviously not complex.

4 Evaluated Machine Models

Our base scalar machine model in the evaluation is the MIPS R3000 [5]. All ILP machine models duplicate the hardware resource of the MIPS R3000 such as the instruction decoders, the function unit, the register file port, and the D-cache port. We evaluated four-degree ILP machines. The *degree of ILP machines* is defined as the number of instructions which are decoded per cycle. In other words, it is the number of instructions that are issued in parallel for the VLIW machines, or the maximum number of instructions that are written into the reservation stations in parallel for the superscalar machine[2]. Thus, the speedup of an ILP machine over a scalar machine is limited by the degree of that machine. Four-degree ILP machines are the most interesting in this research from the view point of complexity and ILP trade-offs. Our cycle count evaluation for a superscalar machine with fully duplicated resources shows that the extra benefit from an ILP machine with a degree over four is severely limited; a four-degree superscalar machine improves performance in cycle counts by 32% over a two-degree superscalar machine, while an eight-degree superscalar machine improves performance by only 3% over a four-degree superscalar machine. A similar trend is observed in VLIW machines. Therefore, we evaluated four-degree ILP machines. In this section, we describe machine models we evaluated.

4.1 Simple VLIW Machine Model

Our simple VLIW machine model has four ALUs, four branch units, two load units, and one store unit. Four instructions are issued in parallel. The register file has eight read ports and four write ports to sustain the peak access rate, while the D-cache has only two ports to keep the D-cache cost reasonable. The instruction set is nearly identical to that of the MIPS R3000. The latency of instructions is the same as that of the MIPS R3000; the latency of load and branch instructions is two cycles, and the latency of other integer instructions is a single cycle. The MIPS architecture uses the compare-and-branch for branching, while the simple VLIW machine uses the branch-on-condition. A 2K-entry branch target buffer [6] whose each entry has the two-bit history and the branch target address is used to predict branches.

We evaluated two simple VLIW machine models with different pipelines. The *VLIW-BP1* model has a five-stage pipeline which is identical to the MIPS R3000 pipeline. We call this pipeline the *MIPS five-stage pipeline*. Figure 4 shows the pipeline. As described in Section 3, complexity is added to the register read, the operand bypassing, and the branch execution. In the VLIW-BP1 model, the complexity may affect the cycle time since each task must be done in a half cycle. The *VLIW-BP2* model, on the other hand, has a *full five-stage pipeline* where a full single cycle is allocated to each task[3]. Figure 5 shows the full five-stage pipeline. Although the complexity may not affect the cycle time in the VLIW-BP2 pipeline, the execution cycle counts increase due to branch misprediction penalty; a single cycle for the MIPS five-stage pipeline, but two cycles for the full five-stage pipeline. We examined trade-offs between the clock rate and branch misprediction penalty.

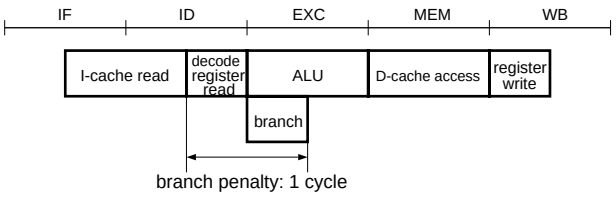


Figure 4: MIPS five-stage pipeline

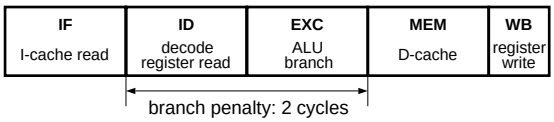


Figure 5: Full five-stage pipeline

Our instruction scheduler for the simple VLIW models picks up a *trace*, a frequently executed path, from a program using a branch profile and moves instructions within the trace. Our scheduler speculatively moves instructions using register renaming and pipeline squashing. After register renaming, copy propagation [1] optimization is applied to eliminate the data dependences upon the replaced copy instruction. Furthermore, we eliminate the copy instruction if the copied variable is no longer used [1]. Although our scheduler does not unroll any loop, it moves instructions from the loop head to the loop tail to increase ILP.

4.2 VLIW Machine Model with Predicating

The VLIW machine model with predicating, which we call *SPEV* (SPeculative Execution VLIW), is equivalent to the simple VLIW machine model except for the predicating mechanism. The SPEV allows speculative code motions past four branches. In addition to the complexity due to the VLIW machine architecture, predicating adds complexity to the register read. As in the simple VLIW machine, we evaluated two SPEV models with different pipelines. The *SPEV-BP1* model has a MIPS five-stage pipeline, and the *SPEV-BP2* model has a full five-stage pipeline.

Our instruction scheduler for the SPEV moves instructions along multiple control paths. For this purpose, the scheduler groups some basic blocks, referred to as a *region* [1], using a branch profile before code motions. A region is a control flow graph (CFG) which includes a header block, and the header block dominates all other blocks in the region. Our region selection algorithm grows a region (candidate) from a seed block (usually a loop head) which is an initial region in the direction of the edges in the CFG. The region is grown if the growth to the next block of the current region is considered beneficial. We use a heuristics which is a function of static branch predication to drive this region growth. After region selec-

2. The superscalar machine has a higher peak issue rate than the degree with reservation stations.

3. The register write sequence begins with sending a write register number to the register file in the MEM stage. A write word line is set up within the MEM stage. Data is then written at the very beginning of the WB stage. This write sequence does not affect the register read time since the read sequence begins at the beginning of a cycle, overlapping the data write with a read register number decode.

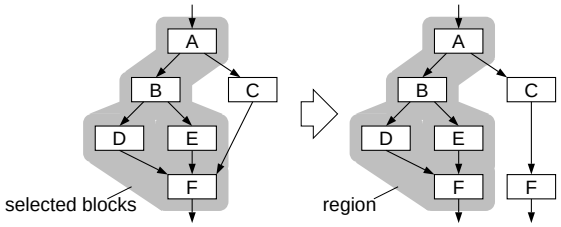


Figure 6: Region selection and formation

tion, the scheduler duplicates the blocks (if necessary) so that the header is able to dominate them[8]. Figure 6 illustrates the region selection and the region formation.

The scheduler is allowed to perform any type of speculative code motions with hardware support in the SPEV. Besides speculative code motions, the scheduler performs some optimizations such as node splitting [8] and loop-invariant instruction migration [1]. Node splitting duplicates a join block that has multiple predecessors to eliminate data dependences along different control paths. This optimization avoids that an instruction in the join block is scheduled late due to a long data dependence path along a less frequently executed path to the join block. An instruction is said to be *loop-invariant* in a loop if the result of that instruction does not change as long as control stays within the loop. Since our scheduler selects a part of a loop as a region, an instruction that is variant in the original loop may change to be invariant in the region. The scheduler moves a loop-invariant instruction outside of the loop if the code motion does not change the program semantics. Since there is no constraint on speculative code motions in the SPEV, the scheduler can move loop-invariant instructions regardless of control dependences. In contrast, loop-invariant instructions cannot be moved in the simple VLIW models due to control dependences in most cases.

4.3 Superscalar Machine Model

The superscalar machine model has the same function units[4] and the branch target buffer as those in the VLIW machines. The hardware organization is shown in Figure 7. Although the degree of the superscalar machine is four, the machine can issue up to nine instructions per cycle using reservation stations. Since the peak rate of the reservation station write from the instruction decoder is four per cycle, only four result buses are provided. This may cause conflict among the result writes from the eight function units, but it has little effect on performance in cycle counts; only 1% improvement without the conflict. A 16-entry reorder buffer is used to buffer results. This is enough for the instruction issue to never stall while waiting for a reorder buffer allocation; only 0.5% cycle count improvement with a 32-entry reorder buffer.

The number of reservation station entries strongly affects both cycle counts and cycle time. Having more entries reduces the cycle counts with a large window, but lengthens the cycle time due to the increase in complexity of the operand bypassing and instruction issue. We evaluated two superscalar machine models with different numbers of reservation station entries. The *SS-RS2* model has two

4.We assume that the ALU in the VLIW machine includes a shifter. Thus, the VLIW machine can issue four shift operations per cycle, while the superscalar machine can issue only a single shift operation. This restriction, however, has little effect on performance since shift operations are infrequent.

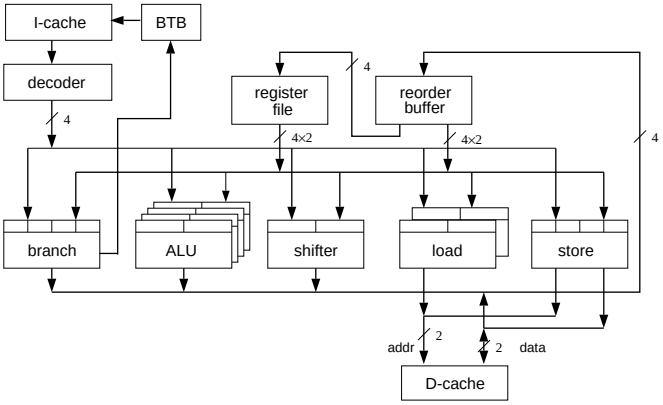


Figure 7: Hardware organization of the superscalar machine

Table 1: Number of reservation station entries for the superscalar machine models

Function Unit (#of FUs)	SS-RS2	SS-RS1
ALU (4)	2×4	1×4
shifter (1)	2×1	1×1
load (2)	2×2	1×2
store (1)	4×1	2×1
branch (1)	4×1	2×1
total	22	11

reservation station entries for each ALU, the shifter, and each load unit; and four reservation station entries for the store unit and the branch unit. Since four entries are allocated to the reservation station of the branch unit, the SS-RS2 can speculatively execute instructions beyond four branches. This is an ability equivalent to that of the SPEV in terms of the number of branches that instructions are allowed to move beyond. The *SS-RS1* model has exactly half the number of entries of the SS-RS2 for each reservation station. Thus, the SS-RS1 has a smaller size of window for instruction issue and less speculative execution ability compared with the SS-RS2. The SS-RS1, however, operates with a faster clock rate than the SS-RS2. Increasing the number of reservation station entries from those of the SS-RS2 is not interesting for this research because the cycle count performance is improved by only 1% with doubled the number of reservation station entries of the SS-RS2. Table 1 summarizes the number of reservation station entries for each superscalar machine model.

As described in Section 3, our superscalar pipeline has two cycles for the dynamic instruction scheduling. Figure 8 illustrates the superscalar pipeline. The ID1 decodes instructions, reads the register file and the reorder buffer, checks data dependences, allocates reorder buffer entries, and allocates tags. The ID2 stage writes instructions to the reservation station, and issues instructions.

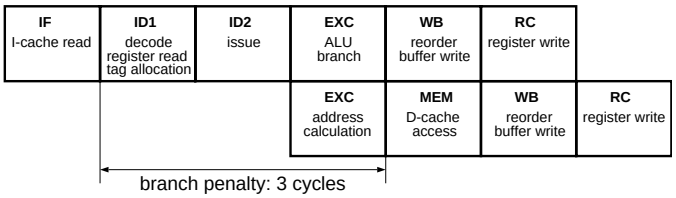


Figure 8: Superscalar pipeline

5 Results

This section first presents the results of evaluation of performance improvement in cycle counts, and the results of cycle time estimation are then shown. Overall performance is compared by calculating the execution time using the cycle count and the cycle time.

5.1 Cycle Count Evaluation

5.1.1 Evaluation Method

The benchmark programs are listed in Table 2. The base machine is the MIPS R3000. We used *pixie* to obtain the R3000 cycles. Pixie is a utility that collects dynamic statistics for programs that run on MIPS machines. Cycle counts for the each ILP machine, on the other hand, are derived from a trace-driven simulator. The speedup over the MIPS R3000 is defined as the total number of R3000 cycles divided by the total number of ILP machine cycles.

Our instruction scheduler schedules code for the simple VLIW machine models and the SPEV from an assembly code that is optimized for the MIPS R3000 by the MIPS compiler. The cycle counts for those VLIW models are obtained from our trace-driven simulator. For the simulation, we annotate the MIPS assembly code to generate an instruction trace in the MIPS program. Our simulator captures the trace information, finds where the control is traversing the VLIW code, and simulates the dynamic behavior of the hardware. For the simulator to correctly identify the control transfer in the VLIW code, the instruction scheduler provides the simulator with information regarding the control flow graph transformations that the scheduler has performed.

The cycle counts for the superscalar machine model are obtained from the trace-driven simulator built by Mike Johnson [4]. The simulator captures an instruction trace which is generated by execution of an annotated program by pixie, and simulates the functionality of the superscalar machine. Simulators for the VLIW machines and the superscalar machine models both assume a perfect memory system. A real memory system will impose more penalty on the VLIW machines than on the superscalar machine since the code amount in the VLIW machines increases. We believe, however, that the penalty can be reduced using a sophisticated memory system since the code expansion by our scheduler is not significant; the code expansion is less than 80%.

5.1.2 Evaluation Results

Figure 9 shows the evaluation results of speedup in cycle counts. The cycle count speedup of the SS-RS2 over the scalar machine is 2.09x as a geometric mean. The SS-RS1 degrades performance to 1.87x, or 10.5% below the SS-RS2. Figure 10 shows the relation between the cycle count speedup and branch prediction accuracy. As shown in the Figure 10, the higher accuracy of branch prediction yields good performance. This is not only because of the small branch misprediction penalty, but because of the large window. If branches are frequently mispredicted, the window is frequently truncated. Thus the effective window size is reduced. The large effective window is beneficial for a superscalar machine having sufficient locations to register instructions of that window. On the other hand, a superscalar having a limited number of locations physically limits the window size. Thus, the benefit from high branch prediction accuracy is limited. Figure 10 indicates this trend. The SS-RS1 shows only a 6% performance degradation for li with a low branch prediction accuracy, while it shows a 15% performance degradation for grep with a high branch prediction accuracy.

Table 2: Benchmark program

Program	Lines	R3000 cycles	Remarks
compress	1,557	21.4M	Data compression
eqntott	3,441	27.5M	Boolean equation minimization
espresso	13,511	11.4M	Optimization of PLA structure
grep	430	15.7M	String search
li	7,429	15.0M	Lisp interpreter
nroff	7,276	19.9M	Formatting document

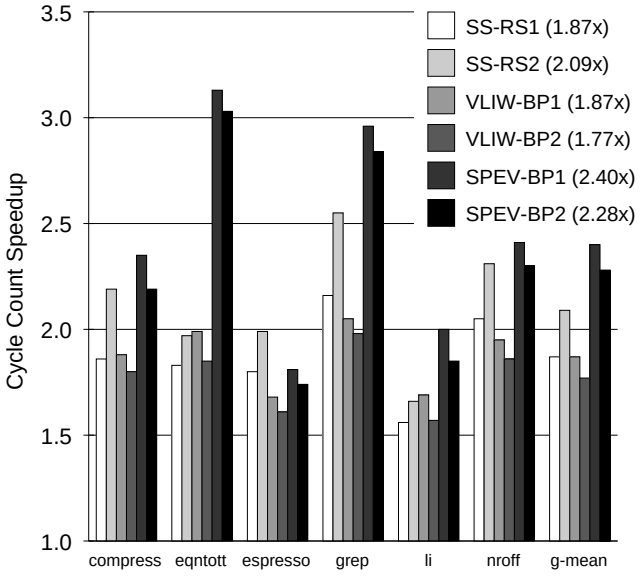


Figure 9: Cycle count speedup of ILP machines

The cycle count speedup over the scalar machine is 1.87x in the VLIW-BP1 and 1.77x in the VLIW-BP2. Thus, the VLIW-BP1

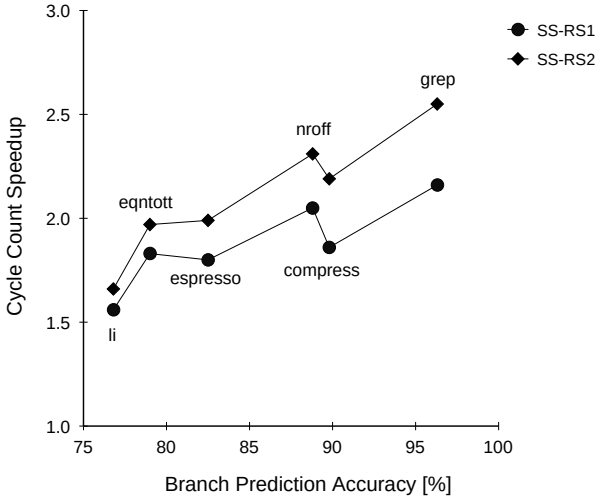


Figure 10: Cycle count speedup vs. branch prediction accuracy

shows a performance degradation 10.5% below the SS-RS2. The VLIW-BP2 further degrades performance by 5.3% below the VLIW-BP1 due to a larger branch misprediction penalty. Our trace scheduling for these VLIW models is superior to dynamic instruction scheduling since it has a large window and has prior knowledge of critical paths across multiple basic blocks in the trace. This scheduler's ability is, however, greatly limited by the restricted speculative execution; the scheduler only relies on software register renaming and hardware pipeline squashing for speculative code motions.

The cycle count speedup over the scalar machine is 2.40x in the SPEV-BP1 and 2.28x in the SPEV-BP2. Thus, the SPEV-BP1 shows a 14.8% performance improvement over the SS-RS2, or a 28.3% improvement over the VLIW-BP1. The SPEV-BP2 degrades performance by 5.0% below the SPEV-BP1 with a larger branch misprediction penalty. With sufficient hardware support for speculative execution, the scheduler is allowed to move instructions speculatively from any path across multiple basic block boundaries in the region. By removing scheduling limitations, the sophistication of compiler's instruction scheduling greatly improves performance.

5.2 Cycle Time Evaluation

Precise cycle time estimation is derived from the entire physical chip design which includes logic, circuit, and layout design. This method is, however, unacceptable for architecture comparison since it requires excessive time and work. Instead, we evaluate just the critical paths that could limit the cycle time. Fortunately, we have a hardware library which includes commonly used components such as memory and adders. These library components have precise information regarding their physical specifications (delay time, input capacitance, area, etc.). Since we had precise information about these components, we had only to design the other logics. We used a 16K-byte cache with 6.0 ns access time, and a 32-bit adder with 3.9 ns latency from the library.

Before the detailed investigation, we first chose the critical paths that have the potential to limit the cycle time. These paths were manually examined with a coarse technology-independent delay estimation method. This method represents the delay time of a logic gate with a number which is derived by the real delay time of the gate divided by that of an inverter with a fan-out of four. For example, the delay time of a NAND gate with two inputs is 1.2 in this method. Similarly, large components such as register files, ALUs, and memory also have technology-independent delay time. Through this preliminary coarse estimation, we choose critical paths that need further precise estimation.

For precise estimation, we first did the logic design for each path. We then added load capacitance to each logic gate output, which was estimated from fan-out and wire length. After that, we measured the delay time using the *PathMill* from the EPIC Inc. The PathMill is a transistor-level static timing analyzer. It reports the delay time of all paths from any input pin to any output pin. Since it models a transistor with a table which is generated in advance by SPICE, the reported time is as precise as one measured by SPICE. We repeatedly performed transistor sizing and logic optimization until the measured time was minimized.

We carefully estimated the wire length since its impact on delay time is significant. The wire length was estimated using the estimation of the area of a hardware unit. We estimated the area of hardware units using the average transistor density data obtained from other designs in our company. We first counted the number of transistors that the hardware unit contained, and then calculated its area by dividing the number of transistors by the transistor density.

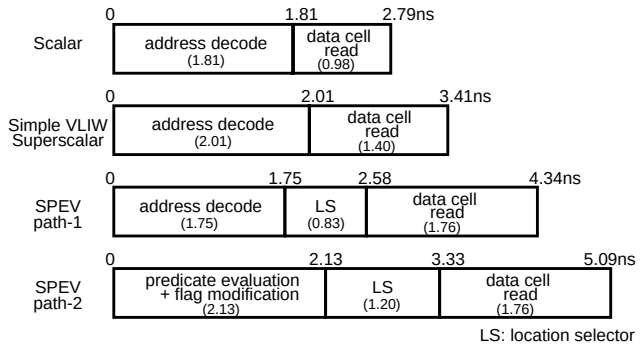


Figure 11: Delay time of register file read

We do not have a multi-ported register file in our hardware library. The delay time of register files strongly depends on the circuit and layout design. Thus, we designed an actual circuit and layout for the data storage cell of a multi-ported register file, and used it to estimate the delay time of the multi-ported register file precisely. All our results assume a 0.5-µm, single-poly, three-layer metal CMOS process.

5.2.1 Evaluation of Complexity

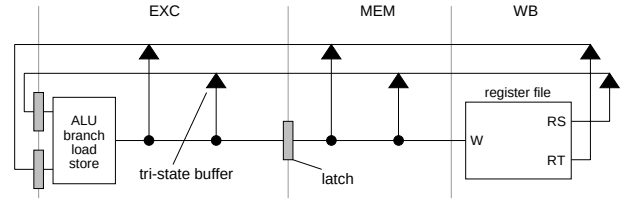
Before estimating the cycle time for each ILP machine, we evaluated the penalty caused by the complexity.

Register File Read. The path from address decode to data out is obviously the critical path in the register file. The predicated register file, as described in Section 3.1, has another critical path; the one that includes predicate evaluation. Figure 11 shows the results of evaluation of the register file read time. First, the penalty caused by the increase in the number of ports is observed by comparing the delay time in the simple VLIW machine with that in the scalar machine. The penalty is 0.62 ns, or 22.2% of the scalar register read time. Second, the penalty caused by two data storage locations in the predicated register file is observed by comparing the SPEV path-1 delay time with the delay time in the simple VLIW machine; the SPEV path-1 bar represents the delay time from address decode to data out in the predicated register file. The penalty is 0.93 ns, or 27.3% of the simple VLIW register read time. A single AND gate delay in the location selector and slow data cell read make the register read slow[5]. Finally, the further penalty added by predication of register values is observed by comparing the SPEV path-2 delay time with the SPEV path-1 delay time; the SPEV path-2 bar represents the delay time from predicate evaluation to data out. The total amount of the penalty on the predicated register file is 1.68 ns, or 49.3% of the simple VLIW register read time[6].

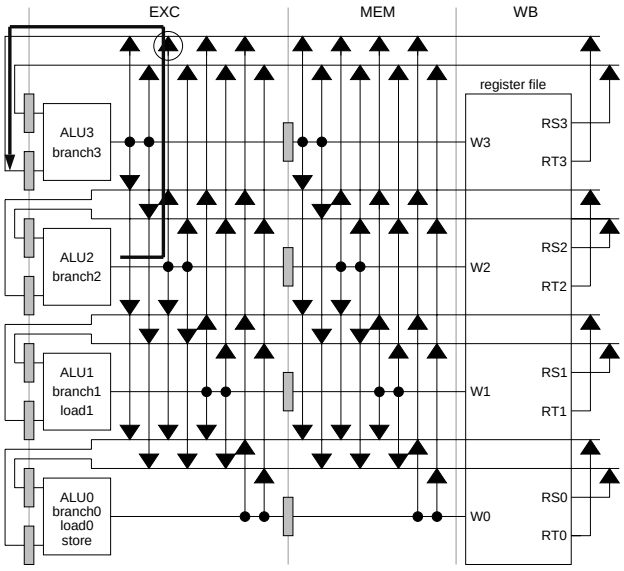
Operand bypassing and instruction issue. Figure 12 shows the operand bypass network for each ILP machine. The circuit for op-

5. The address decode in the simple VLIW is slower than in the SPEV path-1 because it is required to drive a word line; the word line drive is included in the location selection time in the SPEV path-1.

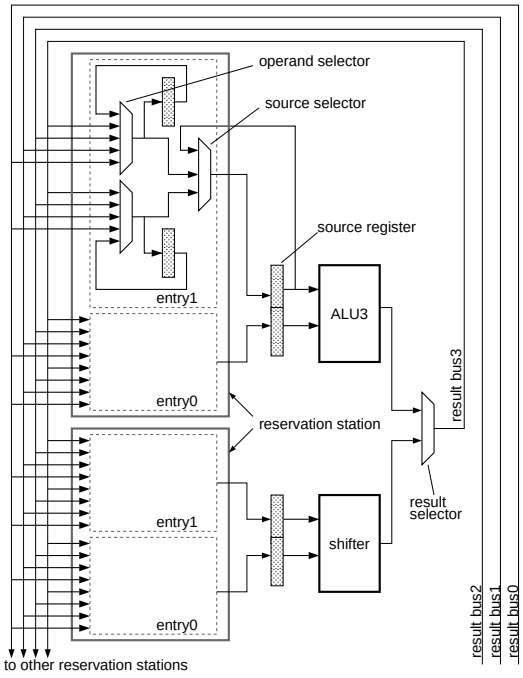
6. The delay time for the location selector is different between the SPEV path-1 and path-2 because the signal path through the location selector is different.



(a) Scalar



(b) Simple VLIW and SPEV



(c) Superscalar (SS-RS2)

Figure 12: Operand bypass network

erand bypass in the simple VLIW machine and the SPEV is a simple extension of one in the scalar machine. An operand in each pipeline stage is forwarded to a pipeline source register through a

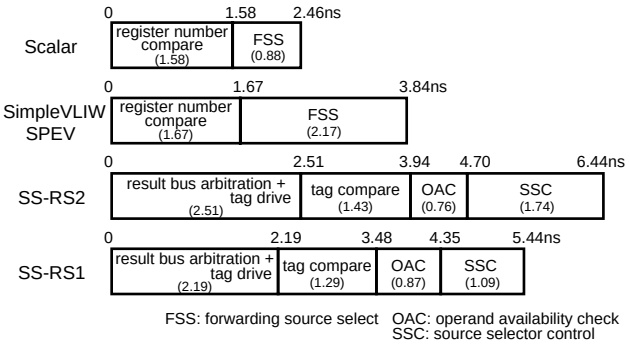


Figure 13: Delay time of bypass control with instruction issue

bus, called a *source bus*. For example, the result in the EXC stage of the ALU2 is forwarded to one of source registers for the pipeline of ALU3 and branch3 by enabling the tri-state buffer marked with the circle (the forwarding path is shown with a thick line). In the superscalar machine, results must be forwarded to all of reservation station entries. Since the superscalar machine has many forwarding destinations (48 destinations for the SS-RS2), the operand bypassing circuit is organized with selectors instead of a tri-state bus; if we organize it with a tri-state bus as in the VLIW machines, an enormous number of wires are required since each destination needs a bus. Three selectors are provided as shown in Figure 12c: the *result selector* selects a result to be forwarded according to result bus arbitration; the *operand selector* selects an operand for a reservation station entry according to tag comparison; the *source selector* selects a source operand for a function unit.

As described in the Section 3.1, the operand bypassing has two types of complexity: bypass control and result drive. Figure 13 shows the delay time of the bypass control with instruction issue time. Since an instruction waiting for an operand gets ready when that operand is forwarded and a ready instruction must be immediately issued, the operand bypassing and instruction issue must be performed in a single cycle. Notice that only the superscalar machine requires time for instruction issue; the VLIW machines do not require time for instruction issue because they are statically scheduled.

From the evaluation results, the penalty caused by the increase in the number of forwarding sources in the VLIW machines is observed by comparing the delay times for the simple VLIW machine or the SPEV with that in the scalar machine. The penalty is 1.38 ns, or 56.1% of the scalar bypass control time. Notice that the significant portion of the delay time increase arises from the forwarding source selection because it must select one of nine candidates (eight forwarding sources and one register value) with priority. The tag drive (with result bus arbitration), the tag compare, and the operand selection (included in the tag compare in Figure 13) in the superscalar machine is semantically equivalent to the register number compare and the forwarding source selection in the scalar and the VLIW machines. In the superscalar machine, a time for instruction issue (operand availability check and source selector control) is further added. As a result, the bypass control with instruction issue in the SS-RS2 requires 6.44 ns, or 67.7% longer than the delay time in the VLIW machines. The time for this work is reduced in the SS-RS1 since the number of destinations where result tags are forwarded is reduced and a reservation station has only a single

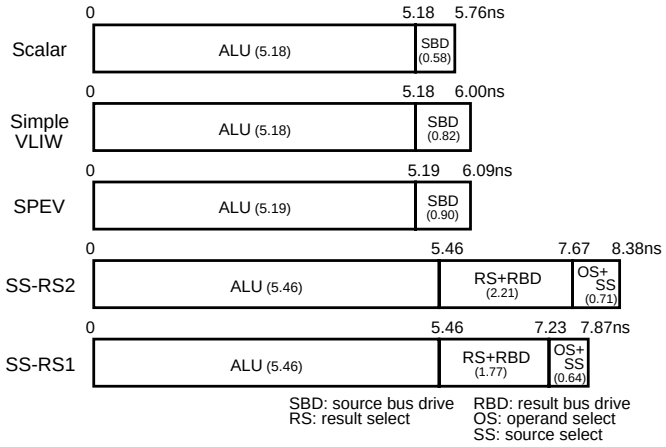


Figure 14: ALU latency with result drive time

instruction[7]. The time for the bypass control with instruction issue is reduced to 5.44 ns, but is still 1.60 ns longer than that of the VLIW machines.

Figure 14 shows the ALU latency with result drive time for operand bypassing[8]. The result drive time is combined with the ALU latency because it gives the result latency of the ALU operation. Remember that the result drive for operand bypassing is implemented with the source bus drive in the scalar and VLIW machines as shown in Figures 12a and 12b, while it is implemented with the result bus drive in the superscalar machine as shown in Figure 12c. An increase in the number of forwarding destinations increases the result drive time. As shown in Figure 14, only a small penalty is imposed on the result latency in the VLIW machines, while a significantly large penalty is imposed on the superscalar machines. Thus, the large fan-out number in the result bus drive and selectors for operand bypassing are found to seriously lengthen the result latency.

Branch Execution. Figure 15 shows the branch execution delay time. Branch execution in the simple VLIW machine is faster than that in the scalar machine. The branch-on-condition in the simple VLIW machine does not need the extra work of the two-operand comparison of compare-and-branch in the scalar machine. Instead, parallel branch execution adds the time for branch selection which selects a taken-branch instruction if multiple branches are taken. Dynamic branch prediction further adds the time for a prediction check. The complexity caused by these tasks is compensated with the complexity removal which is brought by branch-on-condition from compare-and-branch. In the SPEV, the predicate evaluation time is nearly equal to the compare time in compare-and-branch since the evaluation is done by comparing the encoded predicate with the contents of CCR. Since a branch is implemented with a predicated jump, branch execution in the SPEV does not require a condition test. This compensates for the extra time caused by a branch prediction hit check. In the superscalar machine, the branch prediction hit check increases the branch execution time by 20.7% above the scalar compare-and-branch.

7.The reservation station for the store and branch unit has two instructions, but these instructions are issued in order. Thus, the complexity of instruction issue is equivalent to that of a single instruction.

8. The signal delays of an ALU operation are slightly different among the machines because of the difference of load capacitance for the ALU output gate.

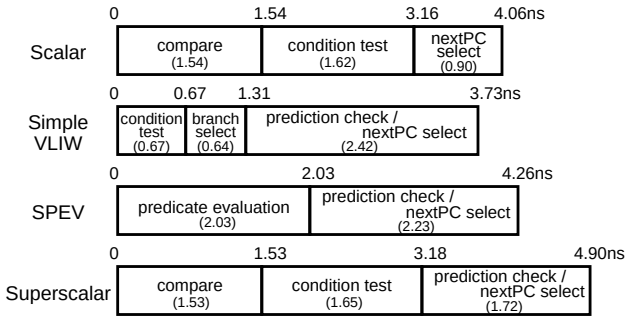


Figure 15: Delay time of branch execution

5.2.2 Cycle Time

This section estimates the cycle time for each ILP machine model using the delay time of components evaluated previously. Since the components are used in a pipeline, pipeline latches must be inserted. We assume a two-phase clock system, and so one latch is inserted at the beginning of a cycle, and another latch is inserted in the middle of a cycle[9].

Before estimating the cycle time, it is convenient to know the lower limit of the cycle time. In any machine, the result latency of an ALU operation (the ALU latency with result drive time) is longer than any other component delay time. Thus, the result latency gives the lower limit of the cycle time. Using circuits that give the delay time shown in Figure 14 with inserting pipeline latches, we estimated the the lower limit. The results are listed in Table 3.

Table 3: Lower limit of the cycle time

Model	Scalar	Simple VLIW	SPEV	SS-RS1	SS-RS2
lower limit time (ns)	6.7	6.9	7.1	8.7	9.2

Models with the MIPS five-stage pipeline. Figure 16 shows the delay time for critical paths for the scalar, the VLIW-BP1, and the SPEV-BP1 models. Note that the delay time of each component is slightly increased from those evaluated in Section 5.2.1 with pipeline latches. In the scalar model, the delay time of the critical path that includes the I-cache read, the register file read, and the branch execution is 15.16 ns. Since two pipeline stages are allocated to this path, the cycle time of the scalar machine is limited to 7.6 ns (15.16 ns / 2 cycles). Notice that the bypass control does not affect the cycle time in the scalar machine.

In the VLIW-BP1, the time for the bypass control is longer than the time for the register file read. Thus, the critical path includes the bypass control, instead of the register file read. Since two pipeline stages are allocated to this path as in the scalar machine, the cycle time of the VLIW-BP1 is limited to 8.0 ns (15.99 ns / 2 cycles), which is 5.3% longer than the scalar cycle time.

In the SPEV-BP1, the same path is not a critical path since a predicated jump instruction in the SPEV does not need register values,

9. Timing constraints (setup time and hold time for a clock) and circuit constraints (for example, latch insertion into data storage cell is not allowed) may add extra delay for a path. Although we considered these constraints in our design, this fortunately had little effect on the path delay time.

i.e., there is no dependence between the register read and the branch execution. Instead, a predicated jump instruction needs CCR values which are generated by the ALU. As a result, the critical path is the path that includes the I-cache read, the register file read, the ALU operation, and the branch execution. Notice that the path-1 in the predicated register file read, a path from register number decode to data out, is included in this critical path. The path-2 in the predicated register file read, on the other hand, is not included because the path-2 starts from the beginning of the ID stage; CCR values for the path-2 are available at the beginning of the cycle, while the register numbers for the path-1 are only available in the middle of the cycle. Three cycles are allocated to this critical path. Therefore, the cycle time is limited to 8.1 ns (24.16 ns / 3 cycles), which is 6.6% longer than the scalar cycle time.

Models with the full five-stage pipeline. The full five-stage pipeline allocates the register read, the bypass control, and the branch execution in the VLIW-BP2 and the SPEV-BP2 to a full single cycle. The path that limits the cycle time in the VLIW-BP1 is allocated to three cycles in the VLIW-BP2. This gives a lower limit of 5.3 ns to the cycle time. Since this number is smaller than the lower limit of cycle time that comes from the result latency, the path no longer limits the cycle time. That is, the cycle time is limited by the result latency of 6.9 ns. This results in cycle time 13.8% shorter than the VLIW-BP1. Similarly, the path that limits the cycle time in the SPEV-BP1 is allocated to four cycles in the SPEV-BP2. Consequently, this path no longer limits the cycle time. Instead, the path which includes the path-2 in the predicated register file read limits the cycle time. That is, the critical path is the path that includes the ALU operation that generates the CCR values and the path-2 in the register file read, as shown in Figure 17. Since two cycles are allocated to this path, the cycle time is limited to 7.1 ns, which is 12.3% less than the SPEV-BP1. In the SS-RS2 and SS-RS1, the result latency limits the cycle time; the register file read, the bypass control with the instruction issue, or the branch execution are all allocated to a single cycle, and either of times for these limiting factors is shorter than the result latency. The cycle time is limited to 9.2 ns in the SS-RS2 and 8.7 ns in the SS-RS1.

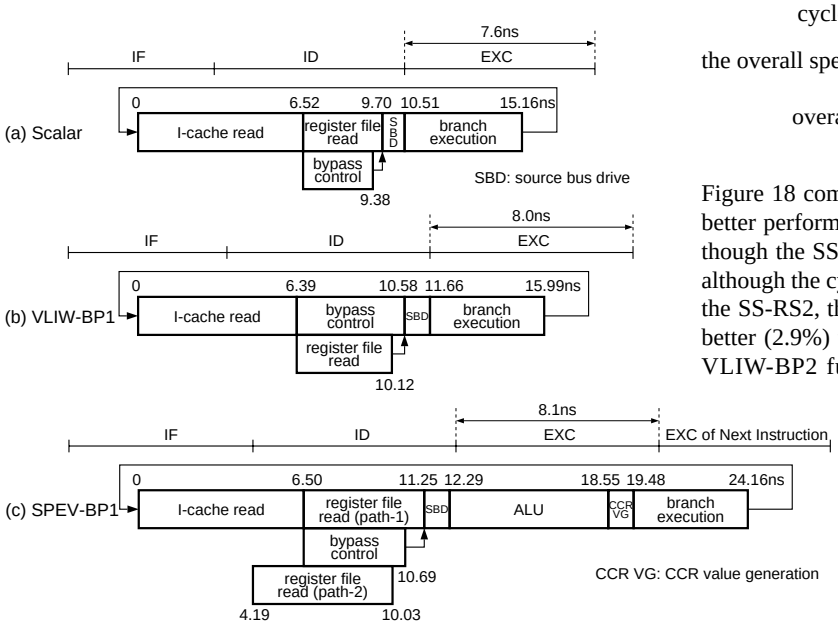


Figure 16: Critical paths in models with the MIPS five-stage pipeline

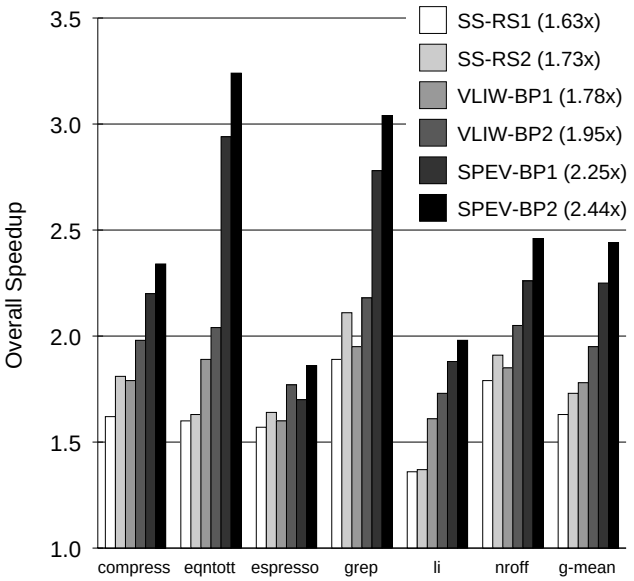


Figure 18: Overall performance comparison

5.3 Overall Performance

Performance must be measured by execution time. The execution time in a model *i* for a program is given by

$$\text{execution time}_i = \text{cycle count}_i \times \text{cycle time}_i$$

where cycle count$_i$ represents the number of cycles when model *i* executes the program, and cycle time$_i$ represents the cycle time of the model *i*. The overall speedup over the scalar machine is given by

$$\text{overall speedup}_i = \text{execution time}_{scalar} / \text{execution time}_i$$

Using the formula of cycle count speedup of the model *i* over the scalar machine:

$$\text{cycle count speedup}_i = \text{cycle count}_{scalar} / \text{cycle count}_i$$

the overall speedup is

$$\text{overall speedup}_i = \text{cycle count speedup}_i \times \frac{\text{cycle time}_{scalar}}{\text{cycle time}_i}$$

Figure 18 compares overall speedup. The SS-RS2 shows a 6.1 % better performance than the SS-RS1 through exploiting more ILP, though the SS-RS2 operates with a slower clock rate. In contrast, although the cycle count speedup of the VLIW-BP1 is smaller than the SS-RS2, the overall performance of the VLIW-BP1 is slightly better (2.9%) than the SS-RS2 because of its fast clock rate. The VLIW-BP2 further improves performance by 9.6% above the

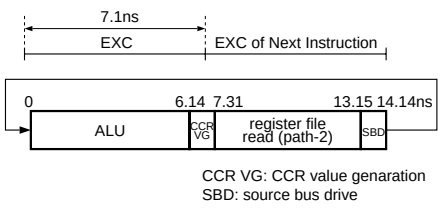


Figure 17: Critical path in the SPEV-BP2

VLIW-BP1 through cycle time improvement. This implies that a faster clock rate yields better performance though the branch misprediction penalty increases. Both the SPEV-BP1 and the SPEV-BP2 have a significantly improved performance over the other ILP models; the overall speedup of the SPEV-BP2 is 2.44x over the scalar machine, 1.41x over the SS-RS2, or 1.25x over the VLIW-BP2. Table 4 summarizes the evaluation results.

Table 4: Evaluation summary

Model	Cycle Count Speedup	Cycle Time	Overall Speedup
scalar	1.0	7.6ns	1.0
SS-RS1	1.87x	8.7ns	1.63x
SS-RS2	2.09x	9.2ns	1.73x
VLIW-BP1	1.87x	8.0ns	1.78x
VLIW-BP2	1.77x	6.9ns	1.95x
SPEV-BP1	2.40x	8.1ns	2.25x
SPEV-BP2	2.28x	7.1ns	2.44x

6 Conclusions

This paper has quantitatively discussed ILP/complexity trade-offs in ILP machines with three different architectures. The mechanism of the superscalar machine for exploiting ILP is complex. As a result, the superscalar machine achieves the smallest performance improvement over the scalar machine in all the ILP machines we evaluated. The simple VLIW machine improves performance with its fast clock rate, but the amount of improvement over the superscalar machine is not significant because of the restriction imposed on the compiler's instruction scheduling. Predicating removes this restriction using hardware support for speculative execution. Consequently, predicating achieves good cycle count speedup. Unlike the superscalar machine, the mechanism is simple enough to impose only a small penalty on the cycle time. The strength of both aspects of ILP exploitation and hardware simplicity achieves a significant performance improvement. The evaluation results show that the VLIW machine with predicating achieves a 2.44x performance improvement over the scalar machine.

Our evaluation results specifically indicate which mechanism causes the cycle time penalty and how big it is. In the superscalar machine, the result drive which broadcasts results to reservation stations inevitably increases the result latency of ALU operations, and thus directly affecting the cycle time. This gets worse if many reservation station locations are provided to aggressively exploit ILP. Since we have neglected some complex work for instruction scheduling and speculative execution by adding a minimum extra pipeline stage for them, the derived result indicates the upper limit of the speedup of the superscalar machine. On the other hand, the VLIW machine lengthens the result latency by only a small amount. Instead, the bypass control and the register file read affect the cycle time. In particular, the predicated register file in the predicating architecture is complex. The penalty caused by them, however, can be reduced by penalty transfer from the cycle time to branch misprediction. A good branch prediction mechanism effectively minimizes the penalty from the deep pipeline.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable comments. We would also like to thank Yasutaka Horiba and Tadashi Sumi for their encouragement.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] H. Ando, C. Nakanishi, T. Hara, and M. Nakaya, "Unconstrained Speculative Execution with Predicated State Buffering," In *Proc. 22nd Int. Symp. on Computer Architecture*, pp.126-137, June 1995.

[3] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," In *Proc. 13th Int. Symp. on Computer Architecture*, pp.386-395, June 1986.

[4] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[5] G. Kane, *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[6] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, Vol. 17 (1), pp.6-22, January 1984.

[7] Scott McFarling and John Hennessy, "Reducing the Cost of Branches," In *Proc. 13th Int. Symp. on Computer Architecture*, pp.396-404, June 1986.

[8] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock," In *Proc. MICRO-25*, pp.45-54, December 1992.

[9] K. Murakami, N. Irie, M. Kuga, and S. Tomita, "SIMP (Single Instruction Stream/Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture," In *Proc. 16th Int. Symp. on Computer Architecture*, pp.78-85, June 1989.

[10] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," In *Proc. 12th Int. Symp. on Computer Architecture*, pp.36-44, June 1985.

[11] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," In *Proc. 17th Int. Symp. on Computer Architecture*, pp.344-355, May 1990.

[12] R. M. Tomasulo, "An efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, 11(1):25-33, January 1967.