The Performance Impact of Exploiting Branch ILP with Tree Representation of ILP Code

SOO-MOOK MOON[1] AND KEMAL EBCIOĞLU[2]

[1]*School of Electrical Engineering, Seoul National University, Seoul, Korea*
[2]*IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA*
Email: smoon@altair.snu.ac.kr

Modern single-CPU microprocessors exploit instruction-level parallelism (ILP) by deriving their performance advantage mainly from parallel execution of ALU and memory instructions within a single clock cycle. This performance advantage obtained by exploiting *data* ILP is severely offset by sequential execution of conditional branches, especially in branch-intensive non-numerical code. Consequently, *branch* ILP must also be exploited by executing branches and data instructions in parallel. This requires compilation support for scheduling branches as well as architectural support for executing branches and data instructions in the same cycle. This paper performs a comprehensive empirical study aimed at evaluating the performance impact of exploiting branch ILP using a representation of ILP code called *tree representation*, which has been proposed by Nicolau [A. Nicolau (1985), Technical Report TR-85-678, Cornell University, Ithaca, NY] and Ebcioğlu to exploit branch ILP in the most generalized form. Our results indicate that exploiting branch ILP can enhance performance substantially (i.e., as much as a geometric mean of speedup 4.5 in the `16-ALU` machine, compared to the base speedup 3.0) and that the performance benefit comes not only from the intended parallel execution but from the decrease of useless speculative execution due to earlier scheduling of branches.

Received August 2, 1996; revised November 10, 1997

1. INTRODUCTION

Very long instruction word (VLIW) and superscalar processors derive their performance advantage mainly from parallel execution of ALU and memory instructions in a single clock cycle. After these data instructions are executed in parallel, sequential execution of branch instructions takes place. In non-numerical code, the frequency of conditional branches reaches around 20% and such sequential execution might offset the performance improvement obtained by exploiting data ILP. Therefore, ILP is often extended to the parallel execution of data instructions and branches [1, 2, 3] and to the parallel execution of multiple branches in a single cycle (*multi-way branching*) [4, 5, 6], exploiting *branch ILP*.

Although architectural extensions for branch ILP have already been provided, the issue of exploiting branch ILP has received less attention compared to that of exploiting data ILP, and there have not been many results reported on its performance effect [4, 7, 8]. While it is true that some performance gain may be expected considering the density of branches in sequential code, it is not quite obvious how many of those branches can, in actuality, be executed in parallel with independent data instructions and with themselves, and so it is unclear where the performance gain really comes from.

This paper performs a comprehensive empirical study aimed at evaluating the performance impact of exploiting branch ILP. Our study emphasizes statically scheduled machines where the compiler groups independent instructions including branches together and where appropriate architectural support is provided for their parallel execution. We employ one of the most aggressive scheduling compilers for non-numerical code [9] which, we believe, is the first one that is equipped with elaborate scheduling techniques for branch ILP as well as for data ILP. In addition, our study uses an intermediate representation of ILP code called *tree representation* which exhibits a high degree of parallelism with respect to both branches and data instructions. This aggressive experimental environment allows us to report and analyze the performance impact precisely.

The rest of the paper is organized as follows. Section 2 provides a brief background of exploiting branch ILP and describes the architectural features involved in the tree representation. Section 3 describes our scheduling compiler and discusses the performance impact of scheduling techniques for branch ILP. Section 4 shows the VLIW implementation of tree representation and describes our experimental environment. Experimental results and our analyses are described in Section 5 and a summary follows in Section 6.

2. ARCHITECTURAL FEATURES OF TREE REPRESENTATION

For statically scheduled machines, the type of architectural support for branch ILP depends on the manner in which branches and data instructions are grouped by the compiler. This Section describes two architectural features involved in the tree representation, *generalized multi-way branching* and *conditional execution*, and compares with those used in other ILP machines based on related compilation techniques.

2.1. Generalized multi-way branching

In a 'traditional' computer, branch instructions specify a condition and a target address; the program counter value is set to the target address if the condition is true. Multi-way branching does not exactly mean executing multiple traditional branch instructions in the same cycle; if so, the program counter value will be undefined when more than one condition becomes true. Multi-way branching works in the context where branch conditions are evaluated using condition registers[1] (e.g. `cc0=(r1==0); if cc0 goto X`) so that multiple branches which test different condition registers can be evaluated simultaneously. Moreover, multi-way branching requires the compiler to group those multiple branches in the form of a binary tree so that the combination of condition values determines a unique target.

For example, consider a code segment in Figure 1a which shows a group of branches explicitly scheduled by the compiler. Since branches tend to cluster together after data instructions are scheduled, multiple branches may be available for scheduling simultaneously. In this example, after scheduling `if cc0`, the compiler schedules `if cc1` and `if cc2` immediately at the true and false targets of `if cc0`, respectively. Consequently, they form a binary tree depicted in Figure 1b where internal nodes test conditions (called *test nodes*) and leaf nodes are branch targets. This entire tree is called a *condition tree* and can be represented by a single four-way branch instruction. Executing a multi-way branch instruction means selecting a branch target by traversing the tree based on condition values until a leaf node is reached. A multi-way branch unit is required to perform this traversal in a single cycle.

Multi-way branch units fall into two categories: those that implement a linear condition tree and those that implement an arbitrary condition tree. Multiflow's TRACE machine allows maximum four-way branching yet its condition tree must always be a linear tree [4]. This is due to its trace scheduling compiler which selects frequent execution paths based on branch probability and schedules a single path at a time; those branches located in the path can be grouped only linearly. For non-numerical code, however, branch probability is not always reliable (e.g. unstable between data sets or equally (50–50) distributed) and it is preferable to schedule instructions across all execution paths to cope with unpredictable branches. In this case, branches may also be grouped across all paths, forming an arbitrary condition tree such as the one shown in Figure 1b. The evaluation of this condition tree in a single cycle requires generalized multi-way branching. Fast generalized multi-way branch units that can be implemented efficiently have already been proposed [5, 6], yet are beyond the scope of this paper.[2]

2.2. Conditional execution

The other part of exploiting branch ILP is the parallel execution of independent data instructions with a (multi-way) branch instruction. There are two possible ways to execute data instructions and a branch instruction in the same cycle. A straightforward method that does not require any architectural extension is to make data instructions execute independently with the branch. A more elaborate method is to allow some data instructions to be executed conditionally pending the outcome of the branch. We call the first method *unconditional execution* and the second one *conditional execution*.

In the context of condition trees, the above alternatives determine where data instructions can be scheduled and located in the tree. Unconditional execution requires data instructions to be located at the edge above the root condition of the tree while conditional execution allows them to be located at any edge of the tree. In Figure 1, for example, let us assume that the two data instructions from targets `T0` and `T3` can also be grouped with the four-way branch instruction. Unconditional execution requires instructions to be grouped only in the form shown in Figure 2a so that all data instructions are executed when the group is executed. Conditional execution allows data instructions to be scheduled below some conditions as shown in Figure 2b, for example, so that they are executed only when those edges where they are located are taken during execution time.

The resource requirement of conditional execution is still the same as unconditional execution since a conditionally executed instruction must be fetched and take a resource even if it might not be executed (for example, the group in Figure 2b requires the same amount of resources for its execution as the group in Figure 2a, no matter which path is taken in the tree of b).[3] However, conditional execution allows the compiler to group branches and data instructions together without having to schedule data instructions above all the branches. This advantage results in better schedulability, hence better performance, which will be described in detail in Section 3.

Conditional execution is an extended form of *predicated execution*, a well-established ILP technique where an

[1] The method of condition codes constrains the ordering of instructions and prohibits parallel evaluation of multiple conditions. The method of compare-and-branch requires too much work for efficient parallel evaluation.

[2] Moon and Carson in [6] compare the two multi-way branch mechanisms in terms of flexibility of representing a given binary tree into the target selection unit. Comparing the performance of both mechanisms using benchmarks is not quite relevant because different branch mechanisms mean different scheduling algorithms; it will compare the scheduling algorithms, not branch mechanisms.

[3] However, there exist architectures where conditional execution is used to remove instructions at various stages of the pipeline, including the instruction buffer and the execution units, thus saving resources [10].

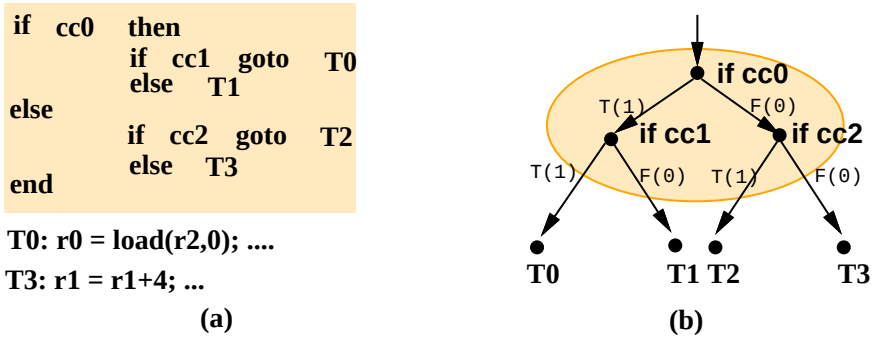

FIGURE 1. (a) An example code segment scheduled in a group; and (b) its corresponding condition tree. All instructions in the shaded region will be executed in the same cycle.

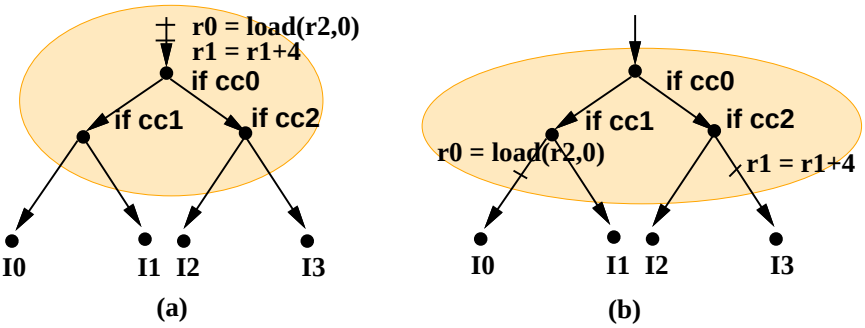

FIGURE 2. (a) Unconditional execution and (b) conditional execution.

instruction is attached to a predicate whose value decides if the instruction can commit its execution [11]. If we allow conditional execution on a single branch, a condition register corresponds to a predicate, whereas a logical expression of condition registers can be regarded as a predicate for conditional execution with multi-way branching. Commercial microprocessors also employ a limited form of predicated execution such as conditional moves [3] or skip instructions [2]. From an architectural point of view, both techniques are identical, yet compilers have exploited them quite differently.

Most scheduling compilers that exploit predicated execution (except for [8]) have not used it for parallel execution of branches and data instructions; rather, it has been used for eliminating branches to convert control dependences to data dependences. Modulo scheduling employed for Cydra-5 uses predicated execution for removing branches from a loop before performing software pipelining [12]. Hyperblock scheduling of the IMPACT group also applies predication to remove those branches whose outcomes are well balanced (50–50) so as to make its trace-based technique (called superblock scheduling) work better [13]. While these techniques apply predication before scheduling, our compiler, which exploits conditional execution, applies predication after scheduling, for compact representation of branches and data instructions. One advantage of this approach is that we can avoid the problem of scheduling with worst-case data dependences and resource requirements among all execution paths, which is likely to occur when applying predication before scheduling since speculative code motion is somewhat constrained.

2.3. Tree representation

A generalized condition tree with data instructions attached on any of its edges is the tree representation which is used in our empirical study as an intermediate representation of parallelized code. By constraining the maximum number of test nodes that can be compacted in the tree and by restricting the locations where data instructions can be scheduled in the tree, we can evaluate a wide range of ILP architectures that exploit different levels of branch ILP. For example, for many superscalar processors that allow parallel execution of data instructions with a single branch, the tree will support unconditional execution with a maximum of one test node. A VLIW implementation of a full-fledged tree representation that supports conditional execution and multi-way branching will be described in Section 4.

3. COMPILATION TECHNIQUES FOR EXPLOITING BRANCH ILP

Conventional optimizing compilers generate code where little branch ILP is exposed; conditional branches are rarely grouped together and there usually exist data

dependences between a branch instruction and data instructions that precede it, preventing their parallel execution. Consequently, elaborate compiler scheduling is required to exploit branch ILP. For our performance evaluation, however, the compiler must also exploit data ILP aggressively to prevent false impact of branch ILP.

This Section describes the scheduling compiler used in our study, focusing on techniques to generate the tree representation of code where both branch ILP and data ILP are highly exploited. Scheduling techniques for branch ILP may affect the performance of data ILP and we discuss this interaction also.

3.1. The selective scheduling compiler

Our scheduling compiler is based on a software pipelining technique called 'enhanced pipeline scheduling' [14] which repetitively applies two actions: first, cut some edges of the given cyclic graph of a loop to yield a directed acyclic graph (DAG); then, schedule the DAG and collect a group of independent instructions on each edge that was cut. This process is repeated in such a way that groups generated later become increasingly compact because they can gather instructions from future iterations as well as from the current iteration across loop back edges. An attractive feature of this algorithm is that the problem of software pipelining is reduced to the simpler problem of DAG scheduling so that repetitive scheduling of DAGs automatically generates software pipelined code. More importantly, loops that have multiple execution paths can be pipelined without worst-case dependences or resource requirements [9].

The DAG obtained after cutting edges in the loop is scheduled by our DAG scheduling algorithm called 'selective scheduling' [15]. The input to the algorithm is a rooted DAG where there is an empty group just before the root node which is associated with specific resource constraints of its corresponding tree representation. The problem is to schedule the group with independent instructions until either the resource constraints are met or no more instructions can be scheduled due to data dependences.

Selective scheduling first computes the set of all available instructions (called an *availability set*) across all execution paths that can move into the group without violating any true data dependences. Non-true data dependences and true dependences on copy instructions are ignored during the computation since they can be handled by renaming and forward substitution techniques, respectively.[4]

Availability is not constrained by control dependences since instructions can be scheduled speculatively before their preceding branches or can be scheduled before control joins after generating bookkeeping code. It is also possible to compute non-speculative code motion ahead of preceding branches since instructions can be scheduled before branches without speculation. For example, an instruction located below an `if-then-else-endif` construct can be scheduled directly above the branch if there exist no dependences in both `then` and `else` parts of the code. While speculative execution is useful only when its original path is taken, this type of non-speculative execution is useful no matter which path is taken.

After computing the availability set, the algorithm classifies instructions based on their degree of speculativeness. The compiler schedules the group by non-speculative instructions first, then the remaining resources are scheduled by 'mildly speculative' instructions, thus involving fewer branches in speculation. This heuristic is valuable for obtaining high performance when branch outcomes are not always predictable since these instructions have higher chances of being useful at execution time.

Our algorithm employs aggressive techniques to compute as many non-speculative or mildly speculative instructions as possible. First, the availability set is, in actuality, collected based only on the computation part of an instruction due to renaming. Second, collecting the availability set is equivalent to moving the computation part of each instruction across execution paths towards the root of the DAG as far as data dependences permit; if the same computation is moving across both targets of a branch, they are unified into one and move above the branch non-speculatively; if a computation is available at only one target, it moves above the branch speculatively. For each computation that can reach the root, we can easily identify its degree of speculativeness, which is the number of branches where it was available at only one target. This greedy computation results in a wider availability set where more non-speculative and mildly speculative instructions are collected compared to other algorithms.

The availability set also includes available conditional branches computed across all execution paths. Since our compiler does not reorder branches in order to prevent excessive code expansion, branch code motion is always non-speculative and is given priority whenever available. Branch code motion duplicates all data instructions located on its path of code motion, and if there are incoming edges to the path of code motion an appropriate bookkeeping copy of the branch must be generated. Consequently, branch code motion may increase the code size and destroy the control structures (for example, if a branch moves across the back edge of a loop for software pipelining, the loop will no longer have a single entry) which are software overheads for exploiting branch ILP.

3.2. Performance impact of branch code motion

A code motion of a branch is performed with respect to group branches and data instructions together for their parallel execution. However, this may also include some side effects on the schedule of data instructions, since speculative code motion before the branch is obviated as a result of the code motion. In this subsection, we will investigate how this side effect influences the performance of data ILP.

[4] For example, consider a code segment [`x=load(y,4); y=z+4; store(x,y);`]. Renaming allows `y=z+4` to be scheduled before the load with a new target register `w` as [`w:=z+4; x:=load(y,4); y:=w; store(x,y)`]. Here, the store can also be scheduled before the copy `y:=w` in the form of `store(x,w)` after forward substitution.

This is closely related to the question of identifying where the performance benefit obtained by exploiting branch ILP really comes from.

Let us imagine some code that has been scheduled for low branch ILP, targeting a tree representation where the maximum number of test nodes is small. The total execution cycles of this code (C_t) are distributed as follows: those cycles where branches are executed only (C_b), those cycles where data instructions are executed only (C_d), and those cycles where both branches and data instructions are executed (C_{bd}), such that $C_t = C_b + C_{bd} + C_d$.

In order to increase its branch ILP, we reschedule the code by increasing maximum number of test nodes, which encourages additional branch code motion. We examine how C_t and its distribution changes compared to those of the original code. Especially, we are interested in the total execution cycles of branches C_{tb} ($= C_b + C_{bd}$) and those of data instructions C_{td} ($= C_{bd} + C_d$). If the rescheduling succeeds in compacting branches more tightly, C_{tb} will obviously be reduced. We are curious about what would happen to C_{td}, the total execution cycles of data instructions. There are two intuitive arguments, one based on dependence constraints and the other based on resource constraints.

The dependence-based argument suggests that a branch by itself does not impose any data dependences so that if data instructions were already scheduled tightly beyond control boundaries, branch code motion would not affect the length of the schedule of data instructions, though duplicating the schedule. That is, even if those branches executed in C_{bd} were grouped more compactly reducing the length of C_{bd}, those cycles from which branches moved out would still remain (now in C_d), maintaining the same length of $C_{bd} + C_d$. In this case, C_t can decrease only by as much as C_b decreases.

The resource-based argument states that scheduling branches early is advantageous because the sooner the branches are performed, the sooner the machine can stop executing speculative instructions that came from untaken paths, thus conserving resources. These conserved resources can be used for earlier execution of instructions from taken paths, generating a shorter schedule of data instructions. In this case, C_{td} may also be decreased, contributing to the decrease of C_t.

Let us take an example to show how branch code motion really influences the schedule of data instructions. Figure 3a shows a piece of code where a group is to be collected at the root of the DAG. We assume a group can be scheduled by a maximum of two data instructions (we do not consider the resource constraints of the branch for the time being; the parallel execution of branches and data instructions will be considered in the next subsection). We consider two cases: (1) when the branch `if cc1` is not scheduled earlier so that the group is collected with the branch fixed at its current location, and (2) when the group is collected after the branch is scheduled earlier before the root of the DAG. These cases roughly depict two possible scenarios that could occur when scheduling for low branch ILP and for high branch ILP, respectively.

In the former case, the group in Figure 3a first schedules its immediate successor `r2=load(r1)`, which is the only non-speculative instruction available. There is one more opening in the group, yet there are two available speculative code motions, one from each target of `if cc1` (`r4=r1+4` and `r5=r1+8`); among them the compiler is assumed to prefer `r4=r1+4` as scheduled in Figure 3b. The next group is also scheduled by its immediate successor `r3=r2*2`, and this time the one opening is scheduled with a non-speculative instruction `r2=r2+1` which is scheduled ahead of `if cc1` as shown in Figure 3c.

In the latter case, the branch is scheduled earlier (e.g. during the scheduling of the previous group) as shown in Figure 4a; the branch code motion duplicated both `r2=load(r1)` and `r3=r2*2`, which were located on its path. The same grouping is performed on this code, gathering two groups on each path that was duplicated, as shown in Figure 4c. Each instruction is marked with `[s]` or `[n]` if it is speculative or non-speculative, respectively. We now examine these groups compared to those in Figure 3.

If one examines the first group scheduled in each path of Figure 4b, one can find that no speculative code motion is performed, unlike the first group scheduled in Figure 3b; `r4=r1+4`, which is scheduled speculatively in Figure 3b is now scheduled non-speculatively in Figure 4b. Moreover, `r5=r1+8`, which has failed to be scheduled previously due to resource shortage, can now be scheduled earlier for parallel execution with `r2=load(r1)`, generating a shorter schedule of data instructions for the false path.

Earlier scheduling of a branch as shown in this example may convert otherwise speculative code motion into a non-speculative or less-speculative one, improving the usefulness of scheduled instructions. Moreover, the resource pressure is improved since a group is duplicated and is scheduled separately for each path; those resources that were previously taken up by useless speculative instructions are now available for earlier scheduling of useful instructions from taken paths. In our aggressive scheduling environment where speculative code motion is performed across all execution paths, these advantages are likely to shorten the schedule of data instructions, thus improving overall performance.

Ideally, we prefer scheduling branches as early as possible once branch conditions are evaluated instead of performing speculative code motion. However, the amount of branch code motion is strictly controlled by the resource constraint of the test nodes in the tree representation and we cannot schedule branches without resources. An interesting question that we need to investigate is how many test nodes should be provided to invoke 'enough' branch code motion to improve the performance of data ILP. Similarly, it should preferably be clarified how much of the performance benefit comes from the improved data ILP due to branch code motion and how much from the originally intended parallel execution.

Branch code motion cannot always shorten the schedule of data instructions, though. Figure 4c shows that there is no difference in the second group even after the branch

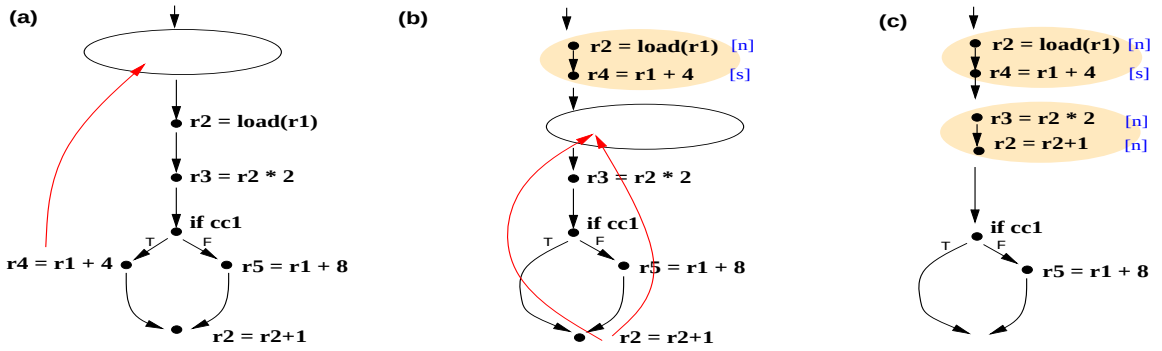

FIGURE 3. Scheduling instructions when the branch is not scheduled.

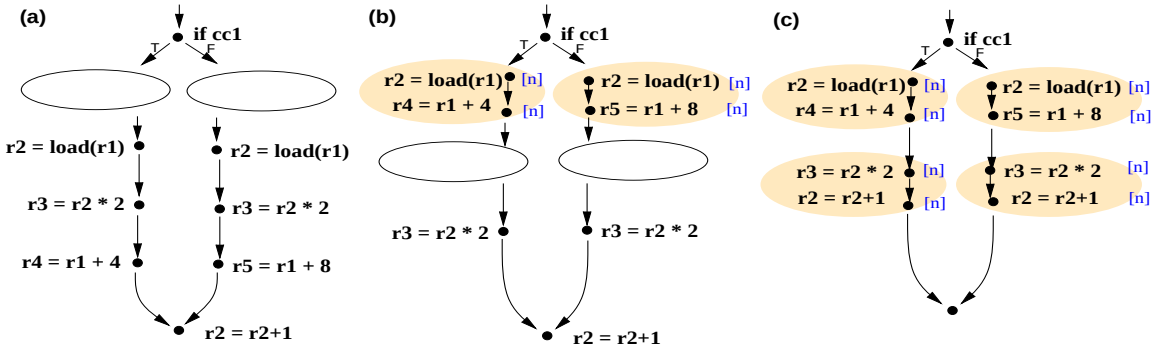

FIGURE 4. Scheduling instructions after the branch is scheduled.

code motion, no matter which path is taken. The reason is that the group in Figure 3c has already been scheduled by non-speculative instructions only and branch code motion cannot produce a better schedule. Similarly, the first group is not shortened when the true path is taken since the speculative execution in Figure 3b is useful in that case. Since our compiler attempts to schedule resources by those instructions which have higher chances of being useful, these cases are plausible especially when resources for data ILP are low.

As can be seen from the above example, both arguments can be true, yet which one is more applicable depends on the usefulness of instructions that have been scheduled in the original code. If the compiler were able to schedule resources of each cycle mainly by non-speculative or useful speculative instructions, C_{td} would remain the same even if C_{tb} were reduced. However, if the compiler is forced to schedule speculative instructions which turn out to be useless, there arise opportunities for additional branch code motion to reduce C_{td}, when the amount of branch code motion is enough to improve the usefulness of the data instructions. Our experimental results in Section 5 will provide detailed data on how many useless speculative instructions are executed when exploiting low branch ILP and on how many of them are removed when exploiting higher branch ILP (e.g. if the false path is taken in the above example, one less number of instructions are executed in Figure 4c than in Figure 3c).

3.3. Conditional execution and speculative execution

Branch code motion as described above may conserve resources for those groups that will be collected on the path of code motion since it obviates speculative code motion. Conditional execution also helps to obviate speculative code motion, yet it improves schedulability, not resource pressure.

For example, consider a code segment in Figure 5a where a group is assumed to be scheduled by a maximum of two data instructions and one branch. If unconditional execution is supported, all available instructions (`r1=r1+4` and `r1=r1+8`) can be scheduled before the branch as shown in Figure 5b. Since the target register `r1` of both instructions are live at the other target of the branch, it should be renamed, leaving a copy at the original location. This code motion is helpful since the copy may be propagated or deleted at later stages, or even when the copy is not eliminated it can be executed concurrently with any dependent instruction after forward substitution as shown in Figure 5c. The problem is that undeleted copies still take up unnecessary resources.

If conditional execution is supported, both instructions can be scheduled at the edges of the branch without leaving copies as shown in Figure 6b, which allows a more compact

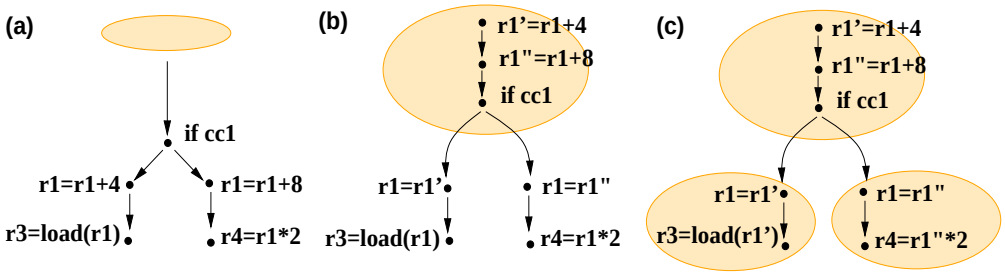

FIGURE 5. Scheduling a group when unconditional execution is used.

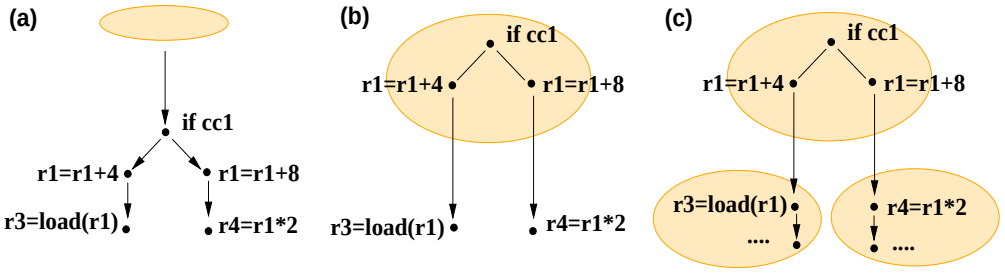

FIGURE 6. Scheduling a group when conditional execution is used.

schedule for the next groups since there are no copies, as shown in Figure 6c. The idea is that conditional execution helps the compiler to group data instructions and branches without speculative code motion. This allows data instructions to be scheduled while keeping their original target registers.

Another difference in schedulability comes from safety constraints that some types of instructions are never scheduled speculatively; for example, store instructions should never be executed speculatively. If both a conditional branch and a store instruction located after the branch are available both can be scheduled in the same group only if conditional execution is supported. Similarly, if the hardware does not support precise exception handling, conditional execution allows aggressive scheduling of unsafe instructions that might cause an exception (e.g. load or divide instructions). A related problem is that it is a desirable heuristic not to rename copy instructions during scheduling since renaming copies increases the number of copies and lowers the quality of the code. Copies can be scheduled below the branch without renaming if conditional execution is supported.

4. EXPERIMENTAL ENVIRONMENT

4.1. Tree VLIW machine

This Section describes the *tree instruction*, a VLIW implementation of the tree representation. Figure 7a shows an example of a tree instruction `L3` whose execution semantics have already been described in Section 2. For a specific implementation, there will be a finite limit on the number of distinct data operations and on the number of test nodes that can be contained in a whole tree instruction.

The execution of `L3` consists of two steps. First, using the current truth values of the condition registers `cc0` and `cc1`, the unique taken path is determined by traversing the tree from the root to a leaf. The instruction label at the selected leaf becomes the next tree instruction to be executed. Then, the instructions attached on the execution path are executed in parallel by performing all the operand 'reads' first, and then all operand 'writes'. If both `cc0` and `cc1` are FALSE, for example, `r0=r1`, `cc0=r1>0`, and `r1=r1+4` will be executed in parallel, and `L3` will branch to `L2`.

A major implementation issue of the tree instruction is performing multi-way branching and conditional execution in a single cycle. In order to reduce the critical path of the cycle time, both should be performed concurrently, with other independent tasks being performed in parallel.

For fast multi-way branching, all target VLIWs are allocated consecutively in memory so that their addresses have the same high-order bits; only the common high-order bits of the next addresses are included in the current VLIW instruction word. All target VLIWs that can be addressed by the specified high-order bits are prefetched from the I-cache at the beginning of the cycle, obtaining a fast I-cache access path with the penalty of potential duplication of VLIWs in memory. For a fast determination of the next target, each path of the tree includes a condition code mask field (a bit string), indicating which condition registers must be true, which conditions must be false, and which conditions are irrelevant for this path to be taken. The low-order bits of the target address for each path are also encoded along with the condition code mask as a target identifier. The values of the condition registers at the beginning of the

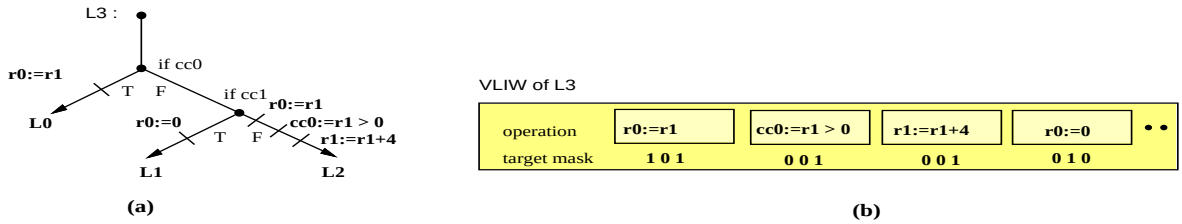

FIGURE 7. A VLIW tree instruction.

TABLE 1. Benchmarks used in the experiment.

Benchmarks		Frequency of conditional branches		Input data files
		Static (%)	Dynamic (%)	
	eqntott	9	25	int_pri_3.eqn
SPEC	espresso	7	14	bca.in, ti.in, tial.in, cps.in
Integer	li	6	15	9 queens
	gcc	10	15	19 C files
	sort	11	30	phone directory
AIX	yacc	9	17	grammar of calculator
Utility	sed	10	23	phone directory
	fgrep	10	23	phone directory
Average		9	20	

cycle and the condition code mask fields for each path allow a quick and parallel determination of the taken path with about one `and-or` gate delay (this occurs in parallel with the I-cache access and so does not affect the cycle time). The chosen target identifier then late-selects the next target VLIW among the ones that were prefetched from the I-cache, using an additional multiplexer. The VLIW machine is indeed pipelined yet has no branch penalties (stalls) as in some superscalars since the VLIW hardware is designed for worst case multi-way branching in every cycle.

For fast conditional execution, each distinct data operation contained in the tree instruction is associated with a *target mask* (a bit string) as in Figure 7b, which indicates on which paths of the tree the operation is present and on which paths it is not. The target mask is used for the late decision of commit/abort depending on the outcome of branching, so that all data operations are allowed to start execution earlier. Due to the target mask, `r0:=r1` in `L3` occupies only one resource although there are two instances. If both cc_0 and cc_1 were set to `F` in cycle n, for example, the execution of `L3` in cycle n+1 proceeds as follows:

- At the beginning of the cycle, all target VLIWs (`L0`, `L1`, and `L2`) selected by the high-order bits in `L3` are prefetched, and all four data operations in `L3` are allocated to ALUs and start execution. Concurrently, the branch unit generates a target identifier using cc_0 and cc_1 and condition masks.
- At the end of the cycle, the chosen target identifier arrives as low-order bits and selects `L2` as the branch target, and only those operations whose target masks include the chosen target `L2` (`r0:=r1`, `cc0=r1>0`, `r1:=r1+4`) commit their execution (they are called *performed* operations) while others (`r0=0`) abort.

In cycle n+2, `L2` will be executed similarly.

The critical data path of the tree VLIW pipeline is described in Appendix A and the description of the multiport register file can be found in [16]. A hardware prototype of an `8-ALU` VLIW machine based on this design has been completed at the IBM T.J. Watson Research Center.

4.2. Environment

The experiments were performed as follows. The input C code is first compiled by the PL.8 optimizing compiler which generates 801-like RISC assembly code; each instruction is assumed to take a single cycle. This sequential code is then parallelized into the VLIW code of tree instructions [9]. Finally, the VLIW code is executed on a VLIW simulator, producing outputs and execution statistics.

The benchmarks used are a subset of SPEC integer benchmarks and four AIX utilities listed in Table 1. For the SPEC integer benchmarks, the official input files were used and simulated to completion. For AIX utilities, we use our own test files.

Our model of the VLIW machine is based on the parametric resource constraints of n ALUs and m-way branching. All ALUs can perform ALU operations and half of them can also perform memory operations. However, an ALU can perform only one operation in each cycle. Branches are assumed to be handled by our dedicated branch

unit without using ALUs. Therefore, a tree instruction on an n ALU and m-way branching machine should satisfy the following resource constraints:

$\texttt{num\_ALU\_ops} \leq n$ $\quad$ $\texttt{num\_memory\_ops} \leq \frac{n}{2}$

$\texttt{num\_test\_nodes} < m$

$\texttt{num\_ALU\_ops} + \texttt{num\_memory\_ops} \leq n$

We evaluate four machines with $n = 2$, 4, 8 and 16, while varying maximum branching ways m; all these machines have 128 general-purpose registers and 16 condition registers and we assumed perfect caches in this experiment. The four machines are summarized in Table 2. The experiment covers both conditional and unconditional execution modes of parallel execution. Consequently, 32 different compilation and simulations of the whole benchmark suite were experimented, and we obtained correct results for all of them.

5. EXPERIMENTAL RESULTS

For each of the four VLIW machines, we first experimented with the case where conditional branches are not scheduled so that they are executed sequentially. This experiment reveals the extreme case where no branch ILP is exploited so that the ILP performance relies only on data ILP. The VLIW code for this case is generated as follows: if the starting instruction scheduled in a group is a data instruction the group is scheduled only by data instructions; otherwise, it is scheduled only by a single conditional branch. Consequently, a VLIW tree instruction is composed of either a set of data instructions or a single conditional branch. It should be noted that many ILP compilers do not move conditional branches as in this case, in order to maintain the original control structures [17, 18, 19]. In the next set of experiments, we exploit branch ILP while varying maximum number of branches that can be compacted in a VLIW (`num_test_nodes`).

5.1. Performance impact of exploiting branch ILP

Figure 8 shows the geometric mean of speedup, which is obtained by comparing the total VLIW execution cycles with the total sequential execution cycles of the PL.8 code. For each of the four machines, the graph includes the speedup of sequential branch execution (`seq.`) and those speedups of exploiting branch ILP with varying maximum number of branches. There is a group of two bars for each machine, the left one for conditional execution and the right one for unconditional execution, respectively.

We can make some observations from the graph. First, sequential branch execution severely limits ILP, especially in larger ALU machines. This limitation is alleviated when branch ILP is exploited. Second, there are consistent speedup increases as more branches are compacted in a VLIW, yet most speedup increases can be obtained with four-way branching which is relatively inexpensive to implement by hardware [6]. Finally, the speedup of conditional execution is consistently higher than that of unconditional execution which confirms the advantage in schedulability for conditional execution.

5.2. Analysis of the performance impact

In this subsection, we will find out where the speedup increase comes from when exploiting higher branch ILP. Figure 9a and b show the speedup and the distribution of execution cycles for the `sed` benchmark, respectively. Each bar in Figure 9b describes the total execution cycles (C_t) and is divided into three parts, C_b, C_d, and C_{bd} ($C_{tb} = C_b + C_{bd}$, $C_{td} = C_d + C_{bd}$), as discussed in SubSection 3.2.

The C_t of the sequential branch execution consists only of C_b and C_d since branches and data instructions cannot be executed in parallel. When we go from sequential branch execution to two-way branching, the graph shows that most of C_b disappear while C_{bd} constitutes a major portion of the total execution cycles for both execution modes. This means that grouping a single branch with independent data instructions by the compiler has been quite successful, and that there are ample opportunities for branch ILP that are exploitable without the support of multi-way branching or conditional execution.

The speedup increase obtained during this transition gives an abstract measure of the performance advantage for simply executing branches and data instructions in parallel. This performance advantage is almost all we can achieve by exploiting branch ILP, if the dependence-based argument holds when we compact branches more tightly; according to the argument, only C_b can possibly be reduced while C_{td} would remain the same, yet the value of C_b is already tiny as shown in the graph.

The graph of the `16-ALU` machine shows that when we go from two-way to four-way branching and from four-way to eight-way branching, C_{tb} decreases significantly, indicating that branches are grouped more compactly. During these transitions, the total execution cycles C_t also decrease significantly. This is not due to the decrease of C_b since it is tiny and unchanged. Most of the decrease comes from the reduction of C_{td}, the total execution cycles of data instructions.

The decrease of C_{td} can be explained by the resource-based argument: scheduling branches early helps to conserve resources by avoiding useless speculative execution. In order to confirm this, we measured the total number of performed data instructions; if the argument holds, the number will decrease as more branch code motion is performed since useless speculative execution decreases. This number will also show the amount of useless execution performed when exploiting low branch ILP.

Figure 10 shows the total number of performed data instructions in the `sed` benchmark for each machine configuration. Obviously, the number in conditional execution is smaller than that in unconditional execution because less copies are performed and data instructions are performed conditionally. The graph also shows the total number of useful data instructions which is the number of data instructions in the sequential execution trace.

For the two-way branching of the `16-ALU` machine, for example, the number of performed data instructions is more than twice as large as that of useful instructions,

TABLE 2. The four VLIW machines used while varying maximum branching ways.

Models	ALU_ops	memory_ops	ALU_ops + Memory_ops	Branching ways
16-ALU	16	8	16	16, 8, 4, 2, seq
8-ALU	8	4	8	16, 8, 4, 2, seq
4-ALU	4	2	4	4, 2, seq
2-ALU	2	1	2	4, 2, seq

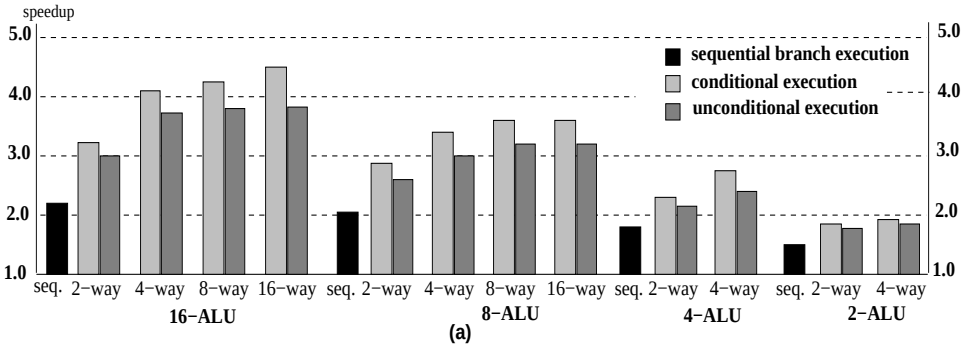

FIGURE 8. Geometric mean of speedup.

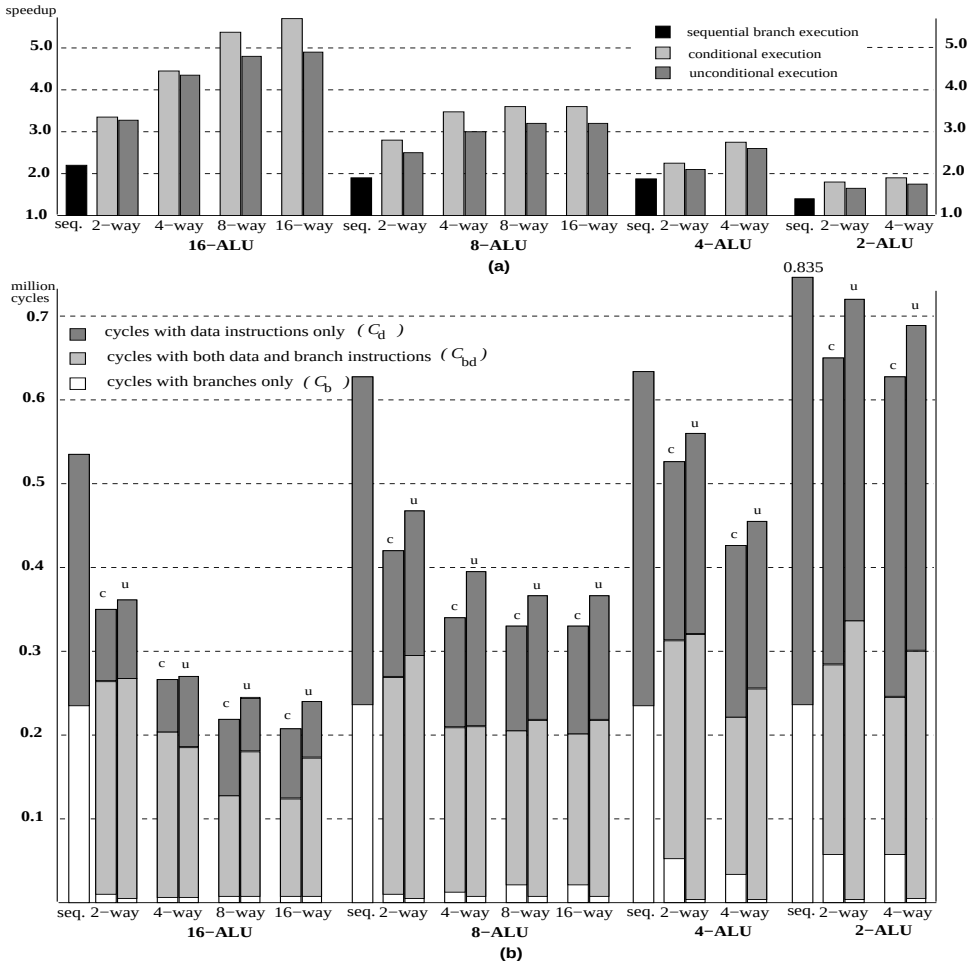

FIGURE 9. Speedup and distribution of cycles in `sed` benchmark.

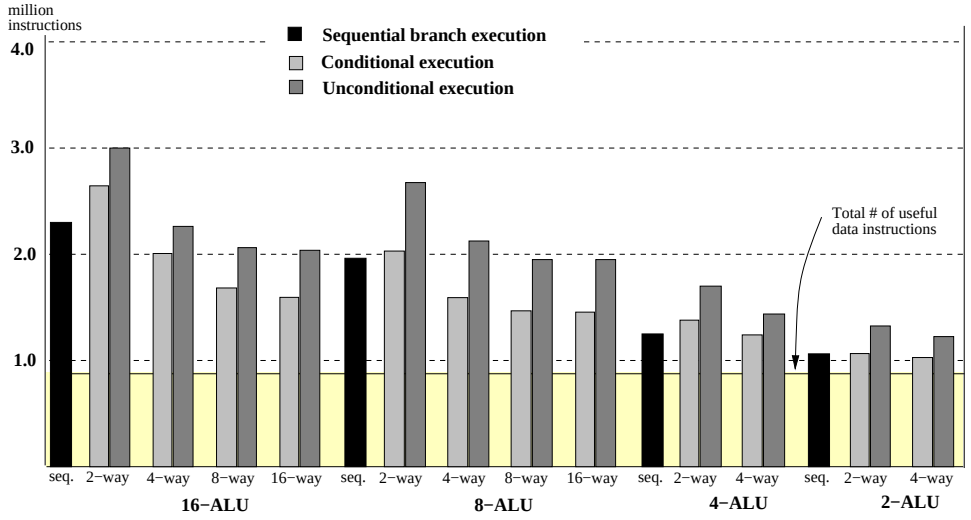

FIGURE 10. The total number of performed and useful data instructions in the `sed` benchmark.

meaning that more than half of the performed instructions are wasted due to useless speculative execution. When we go to four-way and to eight-way branching, the number decreases significantly, so does useless speculative execution. Moreover, the decrease is in proportion to the decrease of both C_{td} and C_t, confirming that the decrease is the major contributor of the performance improvement.

Figure 9 also shows that for both `8-ALU` and `4-ALU` machines, C_t decreases significantly when we go from two-way to four-way branching due to the decrease of C_{td}. This can also be explained by the resource-based argument and can be confirmed from Figure 10. For `2-ALU` machines, however, the decreases of both C_t and C_{td} when going to four-way branching are less prominent compared to the decrease of C_{tb}, unlike in other machines. We believe this is the place where the dependence-based argument starts to become effective such that the amount of branch code motion is not enough to improve the usefulness of data instructions substantially. This is also confirmed by the graph showing the total number of performed data instructions which decreases slightly when going to four-way branching.

We can observe similar and clearer behaviour in other benchmarks where C_{td} remains the same while C_{tb} decreases significantly, sometimes in large ALU machines, but mostly in `2-ALU` machines (e.g. `eqntott`, `sort`, `yacc` and `espresso`). Since the number of ALUs is only two, our scheduling compiler had a better chance to schedule ALUs by non-speculative instructions only so that many groups are already useful without branch code motion.

There are other places in Figure 9 where both C_t and C_{td} decrease little (e.g. `16-ALU` 16-way branching, `8-ALU` 8-way and 16-way branching), yet these are places where C_{tb} does not decrease either. This means that the rescheduling fails to compact branches more tightly or no further branch code motion is possible, hence no effect on the schedule of data instructions. Actually, this is typically what happens when going to more than four-way branching in large ALU machines.

We have also made consistent observations for other benchmarks which are summarized below:[5]

- Two-way branching is generally enough to group most of branches with data instructions.
- When multi-way branching increases performance, it usually comes with the decrease of useless speculative execution. Two-way branching is not enough to fully derive this performance advantage, yet in most cases higher than four-way branching is more than enough.

Let us interpret the above observations. The performance advantage of exploiting branch ILP comes from two types of parallel execution: parallel execution of branches with independent data instructions and multi-way branching. Our observation indicates that the first type of parallel execution is successfully achieved with two-way branching. Multi-way branching increases performance not only due to its parallel execution but due to additional branch code motion encouraged by the grouping of branches, which improves data ILP.

6. SUMMARY AND FUTURE WORK

In this paper, we have addressed the issue of exploiting branch ILP and performed a comprehensive empirical study

[5]One minor thing that can be observed in Figure 10 is that the number of performed data instructions of sequential branch execution is smaller than that of parallel execution. Although this number is somewhat independent from others since sequential branch execution uses a different scheduling method, the primary reason is that speculative code motion is performed less during its scheduling. That is, all data instructions that were previously scheduled with a branch in the same tree instruction (speculatively or conditionally) are now forced to be scheduled in the next cycle non-speculatively. Consequently, the final code is less speculative yet is less aggressive at the same time, thus resulting in poor performance.

aimed at evaluating its performance impact. Our results indicate that the performance benefit is substantial and that the benefit comes not only from the intended parallel execution but from the decrease of useless speculative execution due to earlier scheduling of branches. Although these results do not completely decide the optimal number of test nodes in a tree representation nor are they directly applicable to other platforms other than statically scheduled ILP machines, they do demonstrate the importance of exploiting branch ILP.

The present work has evaluated one approach of exploiting branch ILP which is based on multi-way branching and conditional execution. The other popular approach is fully resolved predicated execution as already described in SubSection 2.2. It is left as a future task to make a fair evaluation of both approaches. Actually, our approach can also be enhanced with the predication support in the middle of scheduling when there are insufficient resources to schedule branches; predication may help to remove branches on an as-needed basis while keeping the advantage of our scheduling techniques. This enhancement might be helpful since multi-way branching involves non-trivial hardware costs related to prefetching or target resolution. We also need to study the effect of multi-way branching on the I-cache performance. Since additional code expansion or hardware flags may make VLIWs larger, the I-cache might cause more cache misses. The I-cache behaviour should be understood to evaluate this additional cost.

ACKNOWLEDGEMENTS

This work was supported in part by the Electronics and Telecommunication Research Institute in Korea under the contract number 1997-321.

REFERENCES

[1] IBM (1990) A special issue on IBM RISC system/6000. *IBM J. Res. Dev.*, **34**.

[2] Asprey, T., Averill, G., Delano, E., Mason, R., Weiner, B. and Yetter, J. (1993) Performance features of the PA7100 microprocessor. *IEEE Micro.*, June, 22–35.

[3] Sites, R. (1993) Alpha AXP architecture. *Commun. ACM*, **36**, 33–44.

[4] Colwell, R., Nix, R., O'Donnel, J., Papworth, D. and Rodman, P. (1988) A VLIW architecture for a trace scheduling compiler. *IEEE Trans. Comput.*, **37**, 967–979.

[5] Ebcioğlu, K. (1988) Some design ideas for a VLIW architecture for sequential natured software. In Cosnard, M. *et al.* (eds), *Parallel Processing*, April, 3–21.

[6] Moon, S.-M. and Carson, S. (1995) Generalized multi-way branch unit for VLIW microprocessors. *IEEE Trans. Parallel Distributed Systems*, **6**, 850–862.

[7] Mahlke, S. (1996) *Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches*. Ph.D. Thesis, University of Illinois, Urbana, IL.

[8] Steven, F., Steven, G. and Wang, L. (1995) Using a resource-limited instruction scheduler to evaluate the iHARP processor. *IEE Proc., Part E, Comput. Digital Techniques*, **142**, 23–31.

[9] Moon, S.-M. and Ebcioglu, K. (1997) Parallelizing non-numerical code with selective scheduling and software pipelining. *ACM Trans. Programming Languages Systems*, **19**, 853–898.

[10] Steven, G., Adams, R., Findlay, P. and Trainis, S. (1992) iHARP: a multiple instruction issue processor. *IEE Proc., Part E, Comput. Digital Techniques*, **139**, 439–449.

[11] Hsu, P. Y. T. and Davidson, E. S. (1986) Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, June, pp. 386–395.

[12] Dehnert, J., Hsu, P. and Bratt, J. (1989) Overlapped loop support in the Cydra 5. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-3)*, pp. 26–38.

[13] Mahlke, S., Lin, D., Chen, W., Hank, R. and Bringmann, R. (1992) Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (Micro-25)*, December, pp. 45–54.

[14] Nakatani, T. and Ebcioglu, K. (1993) Making compaction based parallelization affordable. *IEEE Trans. Parallel Distributed Systems*, September, pp. 1014–1529.

[15] Moon, S.-M. and Ebcioğlu, K. (1992) An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (Micro-25)*, December, pp. 55–71. Also *available as* IBM Research Report RC 17962.

[16] Chappell, B., Schuster, S., Chappell, T. and Ebcioğlu, K. (1993) Virtual Multi-Port RAM. *US Patent No. 5204841*.

[17] Hwu, W. -M. *et al.* (1993) The superblock: an effective technique for VLIW and superscalar compilation. *J. Supercomputing, special issue on Instruction-Level Parallelism*, **7**, 229–248.

[18] Bernstein, D. and Rodeh, M. (1991) Global instruction scheduling for superscalar machines. In *Proceedings of the SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 241–255.

[19] Smith, M., Horowitz, M. and Lam, M. (1992) Efficient superscalar performance through boosting. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-5)*, pp. 248–259.

APPENDIX A. THE CRITICAL DATA PATH OF THE TREE VLIW PIPELINE

This appendix will only appear in the electronic version.

APPENDIX B. DETAILED EXPERIMENTAL RESULTS

See Figures B.1 to B.7 on the following pages for detailed experimental results.

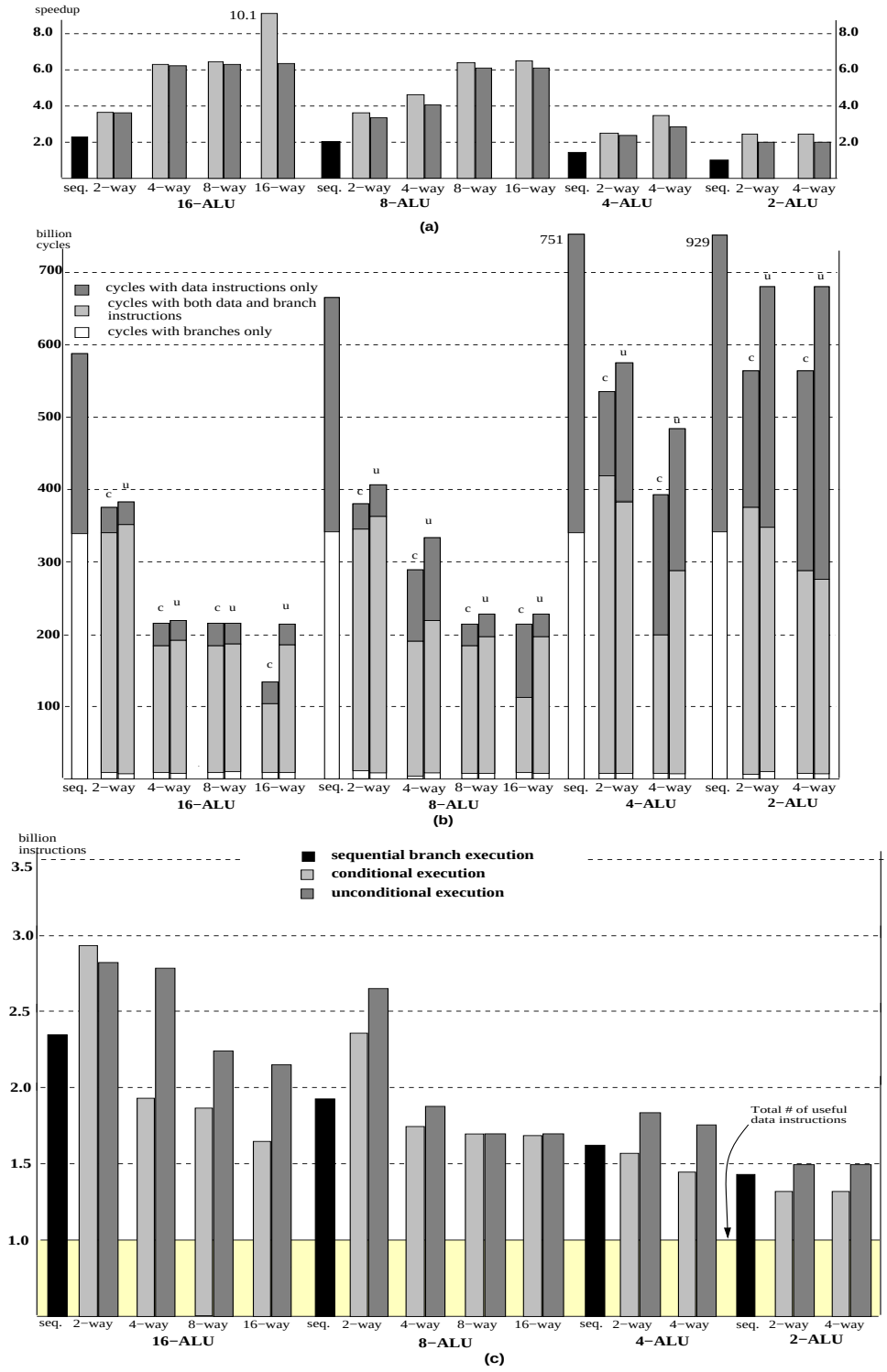

FIGURE B.1. Experimental results for the `eqntott` benchmark; the 16-way of the `16-ALU` machine shows a higher speedup due to a critical inner loop in the subroutine 'cmppt' which executes at a rate of one cycle/iteration; the tree instruction is compacted with 15 data instructions and 11 test nodes.

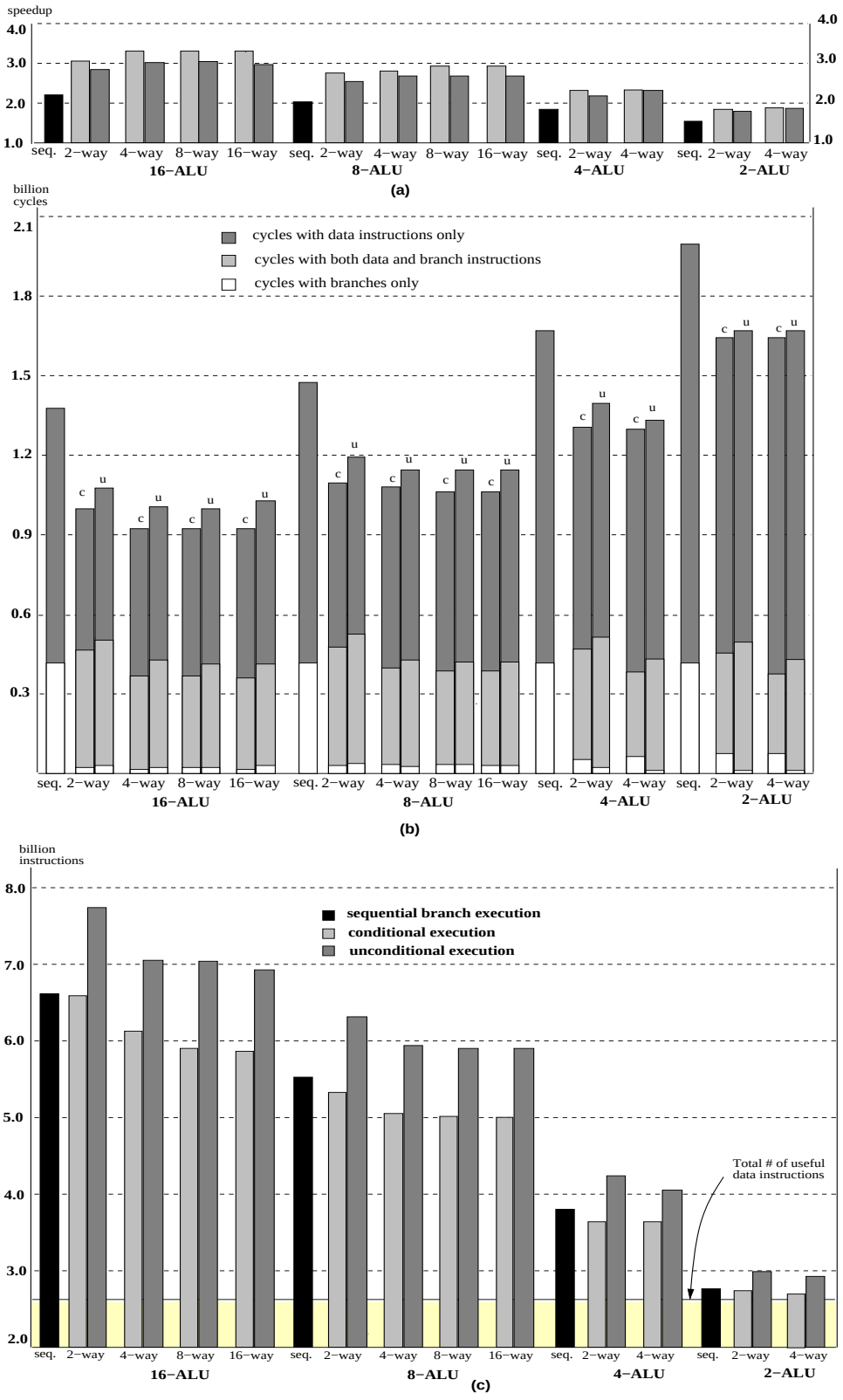

FIGURE B.2. Experimental results for the `espresso` benchmark.

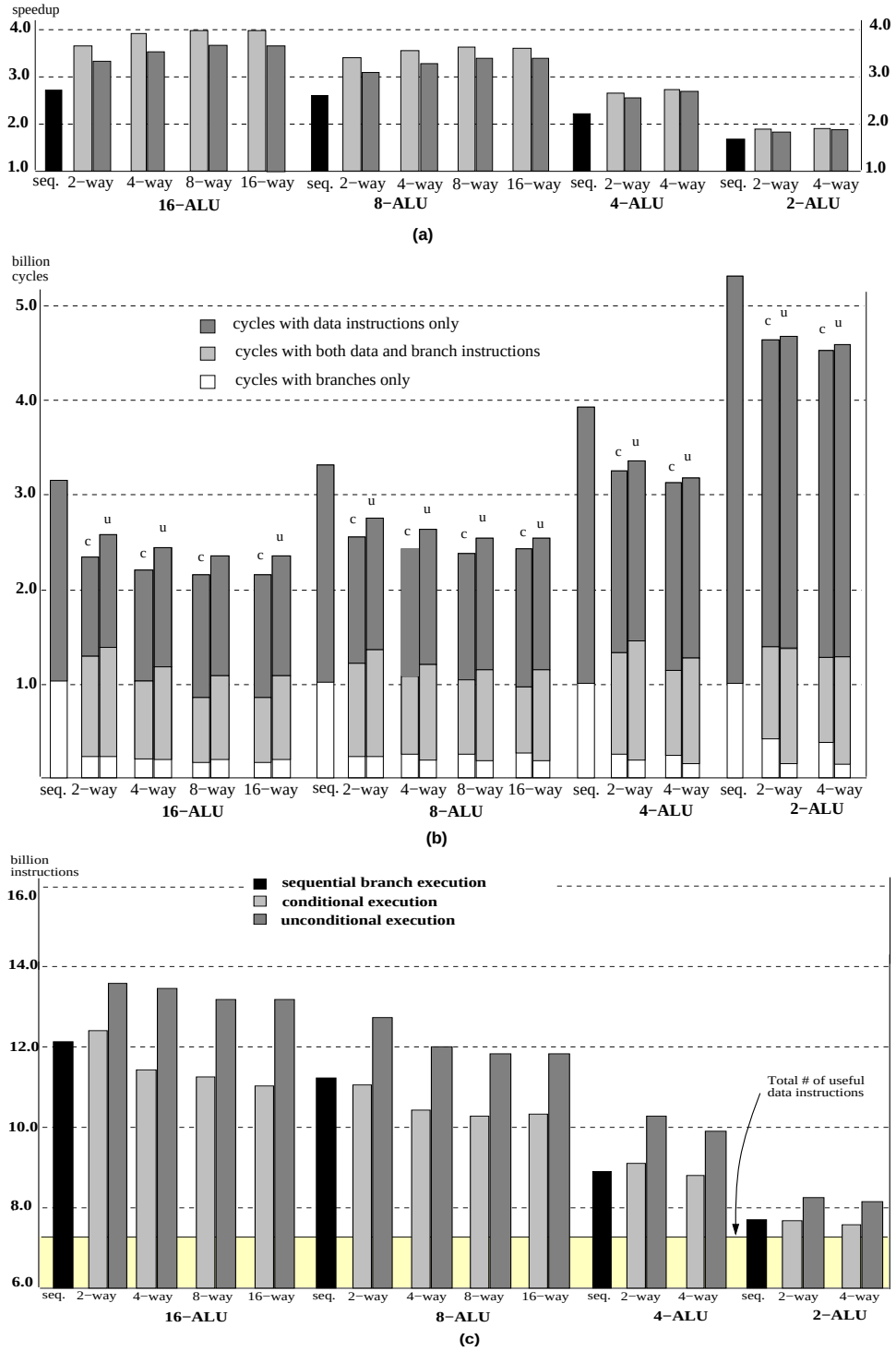

FIGURE B.3. Experimental results for the `li` benchmark.

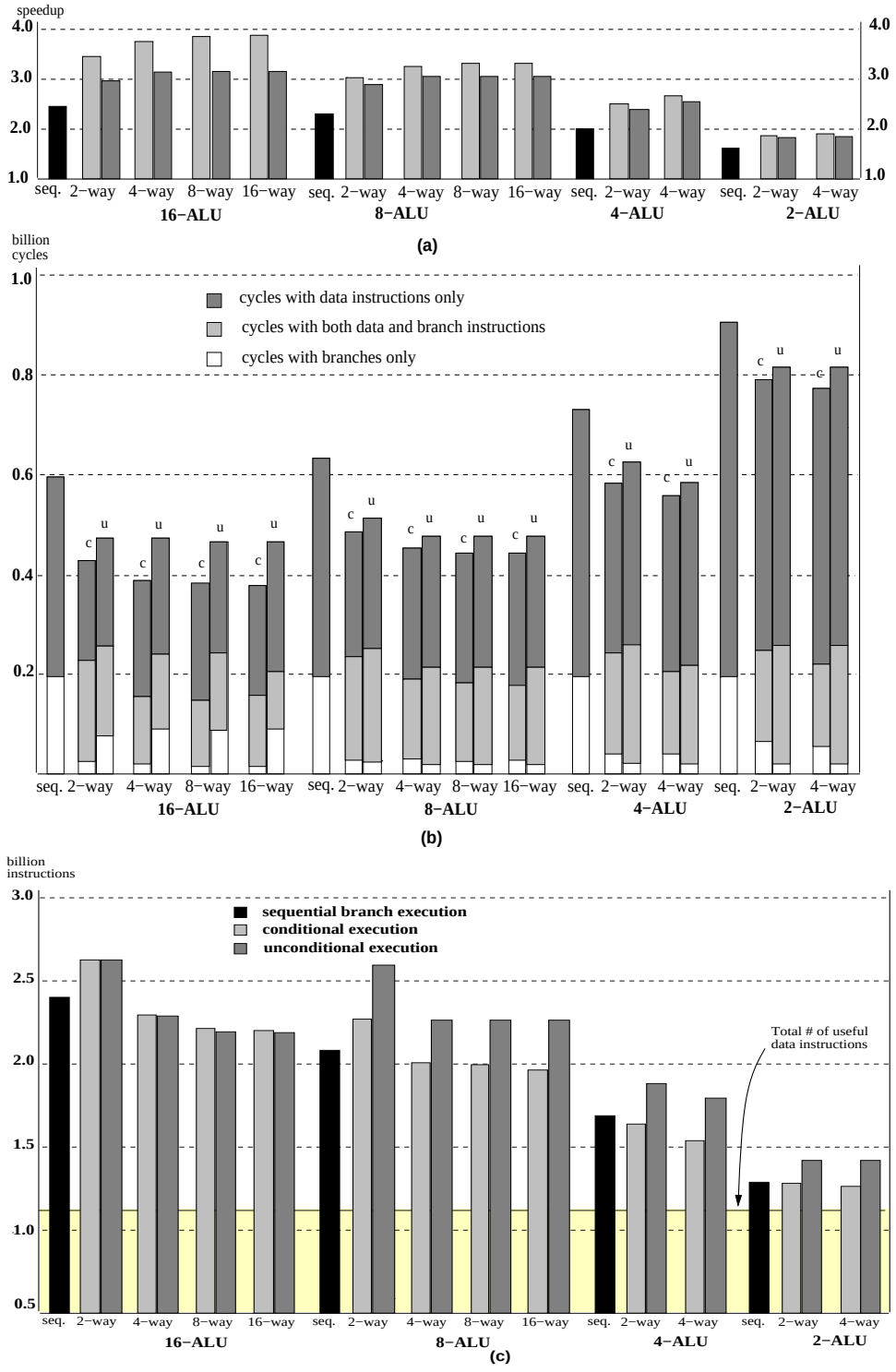

FIGURE B.4. Experimental results for the `gcc` benchmark.

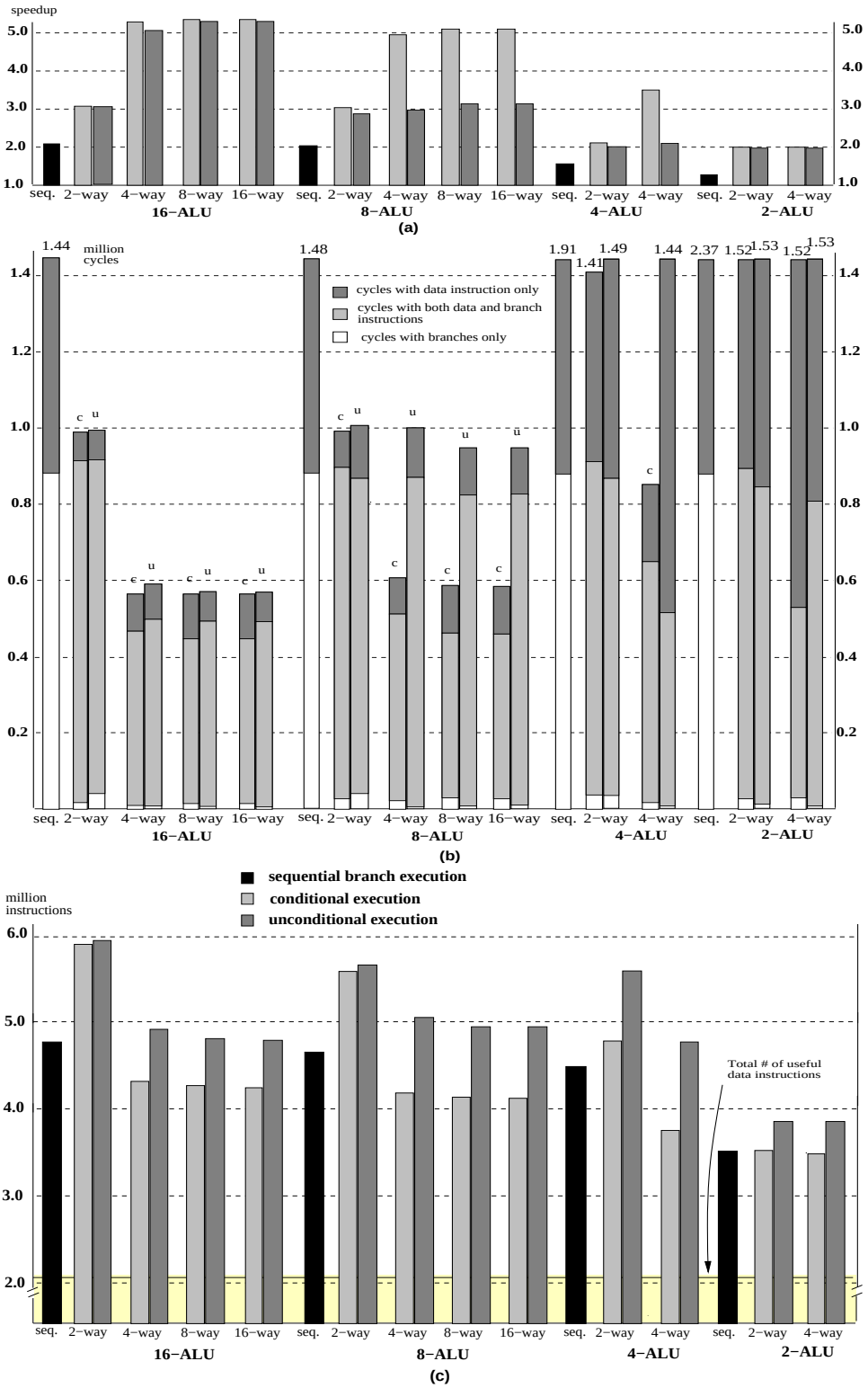

FIGURE B.5. Experimental results for the `sort` benchmark.

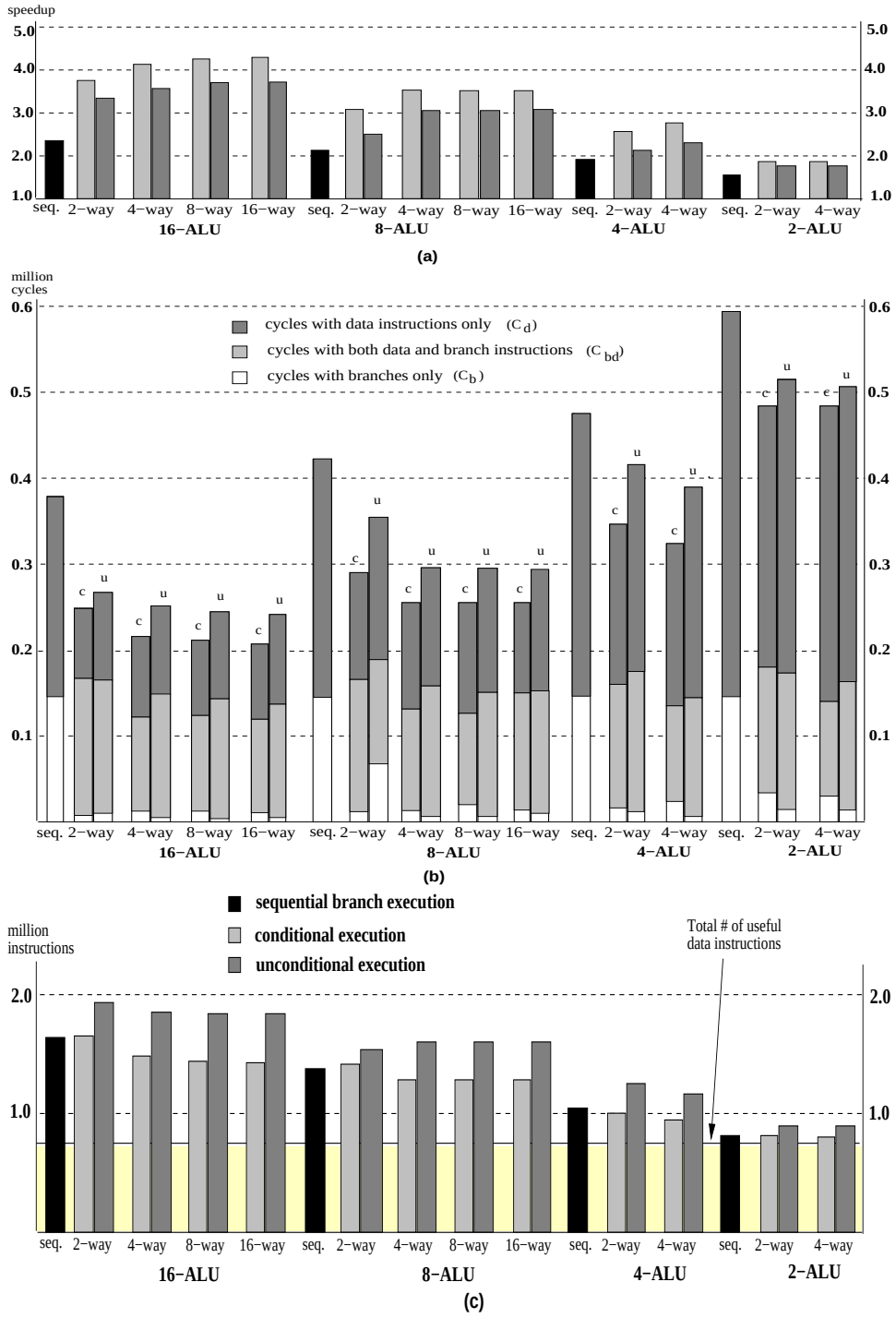

FIGURE B.6. Experimental results for the `yacc` benchmark.

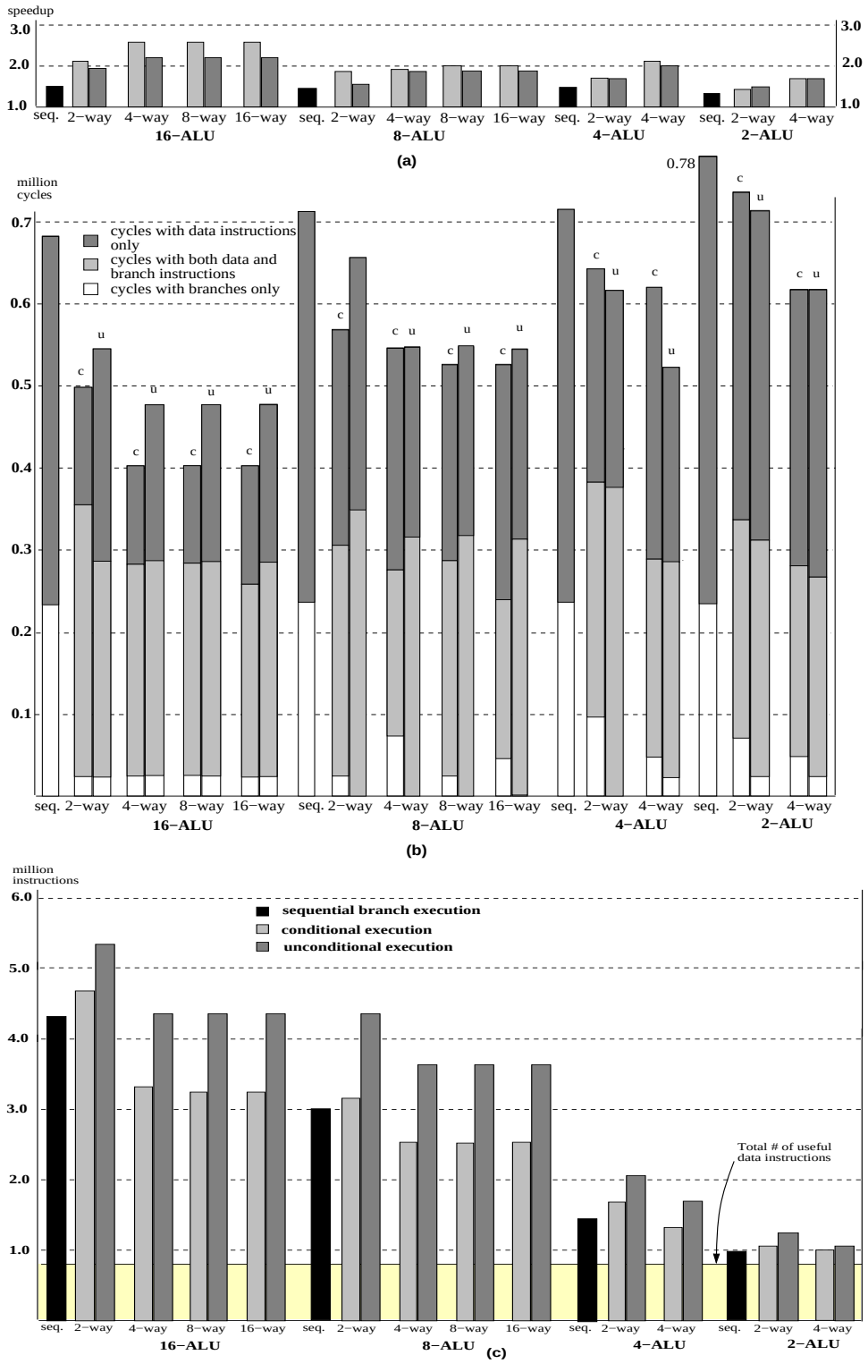

FIGURE B.7. Experimental results for the `fgrep` benchmark.