Feedback-Directed Switch-Case Statement Optimization

Abstract

Switch-case statements provide a concise way to express multi-way branching control flow semantics. Switch-cases are common in programs, including script parsers, compilers and virtual machines. Because they occur in widely used programs, optimization for switch-cases has been studied since the 1970's. This paper presents two new techniques: hot default case promotion (DP) and switch-case statement partitioning (SP). DP improves case dispatch performance while SP simplifies case dispatch, improves instruction layout and enables further inlining. Both DP and SP are guided by runtime feedback information. An extensive experimental study compares the runtime performance of existent switch-case transformations and these new techniques. The effectiveness of these switch-case transformations depends on how these statements are used in a program. For benchmarks that heavily use switch-case statements, we observed overall execution time improvements of up to 4.96% when compared with a straightforward jump table approach. A micro-architecture level performance study provides insights on the basis for this performance improvement.

1 Introduction

In most modern programming languages multi-way branch semantics are expressed by switch-case statements. These statements are frequently used in the implementation of script interpreters, compilers and virtual machines because these applications often use the value of a key to select one among a large collection of possible actions. A switch-case statement contains a *key expression*, a set of (*case value, case action code*) pairs and a *default action code*.

The execution of a switch-case statement has two distinct phases: case selection and case action [14]. Case selection decides which case should be executed based on the value of the key (or switch expression). If the key is equal to one of the enumerated case values, the control flow is directed to the corresponding action code. Otherwise, the default action code is executed. In this case, we say that the key falls in the default category or it is a default case.

Section 2 reviews known techniques for the generation of code for switch-cases that do not rely on feedback information. The creators of these techniques are credited in the related work section (Section 6). The main contributions of this paper are:

- Hot Default Case Promotion (DP) and Switch-Case Partitioning (SP) (Section 3). These two new techniques make use of feedback information to improve switch-cases.
- A detailed description of the implementation of DP and SP in the ORC 2.1 compiler (Section 4), including the algorithms required to handle control flow between cases.
- An experimental study (Section 5) that includes not only the new techniques but also known techniques such as branch sequences and jump tables. To our best knowledge, this is the first systematic performance study on switch-case optimizations in real applications.

2 Earlier Work: Switch-Cases Without Feedback

Early work on code generation for switch-cases focuses on case selection and does not include the use of feedback through profiling runs. This section briefly describes these techniques.

Search strategy (BR): Case selection is implemented as a series of branch-if-equal operations. The key is compared against each case value until a match is found. If no match is found, the default action is executed. Most compilers adopt either a linear or a binary search algorithm to find a matching value for the key. In single-issue processors, assuming the key is already loaded into a register, each case enumeration requires two instructions (one compare and one branch). Thus,

the total number of cycles spent in the case selection for a switch-case S is:

$$\text{TOTAL\_CYCLES}_{BR}(S) = \sum_{i=1}^{num+1} 2 \times \text{ORDER}_i \times \text{FREQ}_i \quad (1)$$

where ORDER_i is the place of case i in the sequence of branches, and FREQ_i is the frequency with which case i occurs at runtime. BR is effective for a small number of cases. For switch-cases with many cases, the search strategy may spend a long time enumerating case values even when some efficient search algorithm is used.

Jump table strategy (JT): Jump tables are used for a large number of case values with a dense distribution. For instance, the jump table in Figure 1.(b) is an array that stores the starting addresses of the action code corresponding to each case value. The table has a *hole* where a case is missing (case 9 is missing in this example). The hole is filled with the address of the default action. Before the key is used to index the table, its value is normalized and a test determines if the normalized key is *on-table*. If a key is *off-table*, the default action is executed.

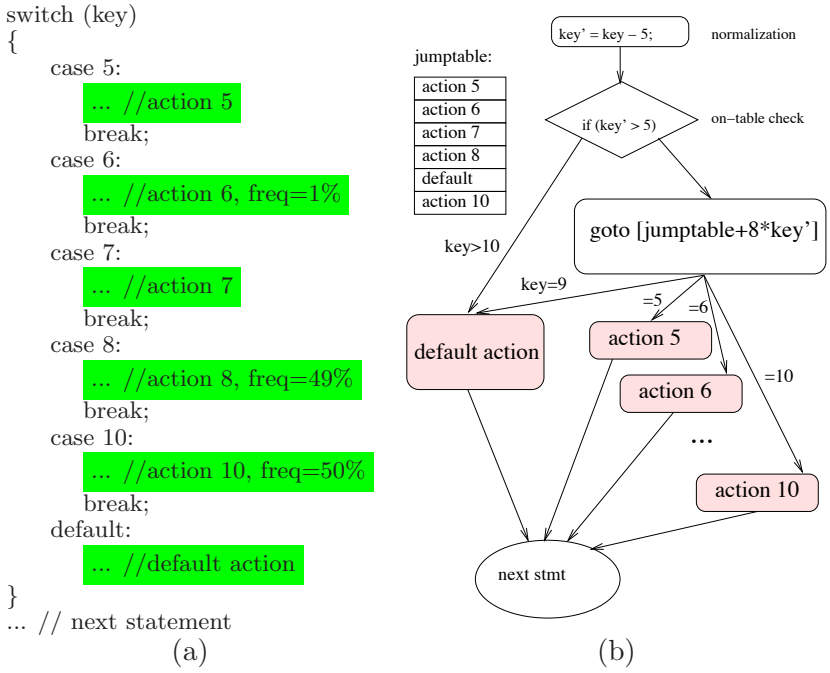


Figure 1: Jump Table Strategy (64bit Addressing)

A drawback of JT in contemporary processors is the requirement of a load from memory to

retrieve the address of the case action. The selection of the default action for an off-table key is fast because it only requires the key normalization and the on-table checking. JT is only efficient for a large number of densely distributed case values. JT applied to a switch-case with sparsely distributed case values results in inefficient memory usage (too many holes in the jumptable) and adverse cache effects.

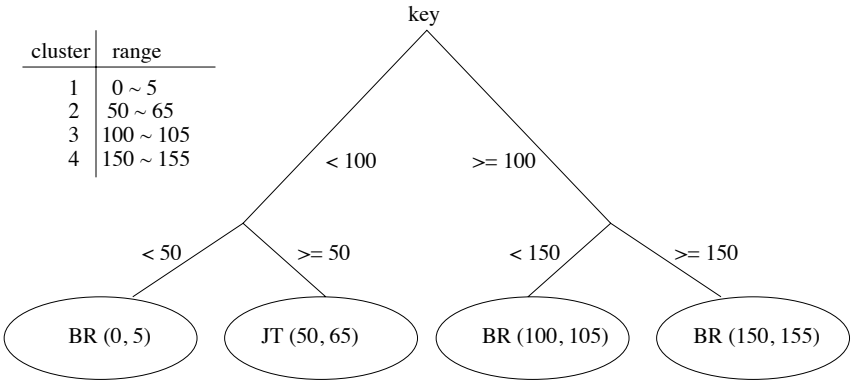


Figure 2: Combined Strategy

Combined strategy (COMB): When case values are distributed into several dense clusters, a combination of BR and JT is used. In Figure 2, the case values of a switch-case form 4 clusters. Using the search strategy may result in the execution of many compare-and-branch instructions. With COMB a binary search is first used to locate the cluster where the key falls in. Then a strategy appropriate to the cluster size, either BR or JT, is used for the case selection within the cluster.

Multi-way Radix Search Tree (MRST) strategy : Proposed for switch-cases with sparsely-distributed cases, MRST uses a contiguous bit field of the case values to create a hash table [9, 10]. Collisions are solved by examining a different set of bits. Therefore, to differentiate all the enumerated case values, a search tree of the hashing values is needed. The search tree is called MRST In practice, case values are not sparse enough to justify MRST. We are not aware of any compiler that uses MRST. Moreover, sparse cases suitable for MRST never occur in the SPEC INT2000 benchmark suite.

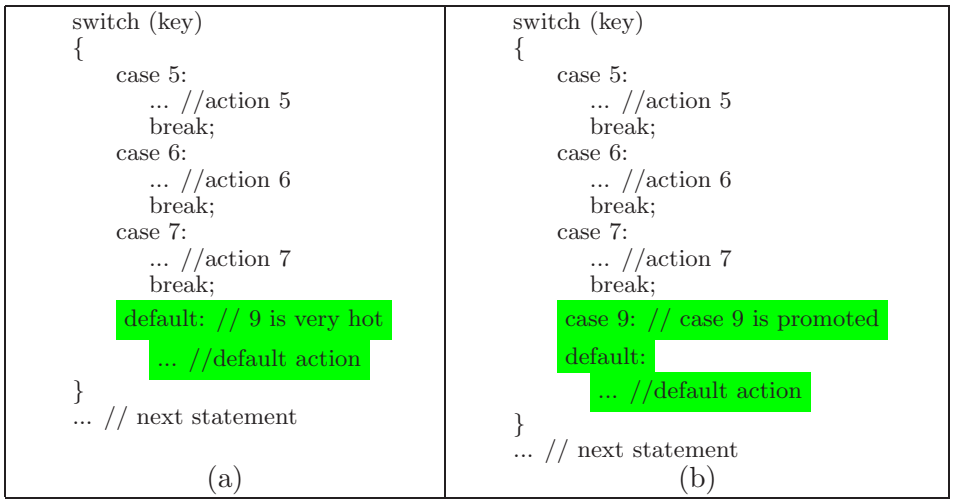


Figure 3: Hot Default Case Promotion

3 Feedback-Guided Optimization

To write robust application programs, one must handle many boundary cases, even the ones that seldom occur during program execution. As a consequence, the execution frequency of action cases is often skewed with a small fraction of cases dominating the execution of a switch-case. This section describes three techniques that make use of execution frequency information to improve the runtime performance of switch-cases.

Hot case hoisting (HH): If a few cases are known to dominate the execution, they should be tested first regardless of the case selection strategy used. For instance, see the frequency annotations in Figure 1(a). Only cases 8 and 10 are frequently executed. If the BR strategy with the enumeration order in the source code is used, 8 and 10 will be the last cases tested resulting in many wasted compare-and-branch instructions. HH tests frequent cases first. Even when the JT strategy is used, testing very frequent cases prior to indexing the table avoids the indirect load into the jump-table.

Hot Default Case Promotion (DP): DP is one of the new techniques proposed in this paper. DP stems from the observation that in some switch-cases the default action is selected very frequently. Moreover, we observed that often a few key values result in the selection of the default action. DP creates new case values for the frequent (*hot*) values that originally fell in the default category.

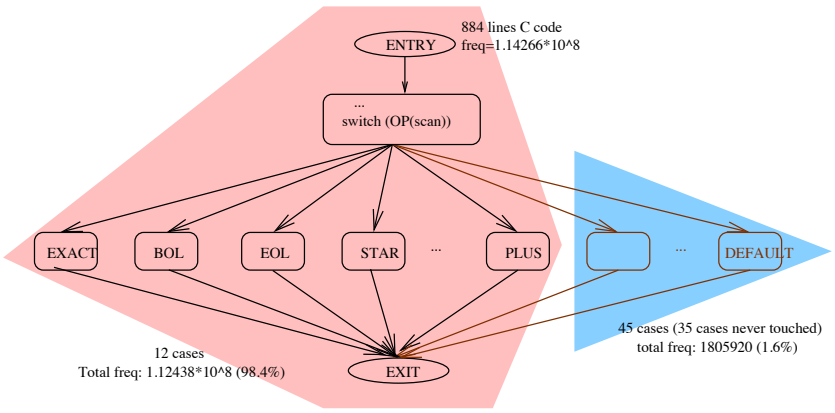


Figure 4: Annotated CFG of function *regmatch* in `perlbmk`

These hot values are identified by analyzing runtime feedback. After promotion, standard HH will move these new values to the correct place in the branch search or hoist them before a jump table according to their frequency when compared with the other cases. For instance, in Figure 3(a), the key value 9 occurs very often. With DP, the value 9 is promoted to a separate case value (Figure 3(b)).

Existing code generation techniques are inefficient when the default action is selected often. With BR all the cases must be tested before the default action is taken. With JT if the key value is off-table the selection is fast. However if the hot key value is in a hole, an indirect load is required to find the default action. HH cannot hoist a default value before it is promoted to a standard case.

Switch-Case Partitioning (SP): SP is the second new technique presented in this paper. SP applies to large switch-cases. Large switch-cases may limit performance improvements because their host functions are often too large to be inlined [6, 7, 8, 18]. Inlining heuristics must be conservative on the amount of code growth permitted to limit compilation time, register pressure and constraints on code placement.

SP is based on the observation that the actions of infrequent cases, (also called *cold cases*) often account for a significant portion of the size of a switch-case. For instance, Figure 4 shows the breakdown of the cases in a switch-case, S, in the function *regmatch* in the SPEC2000's `perlbmk` program. This function matches regular expressions with strings in a file. S requires 800 lines

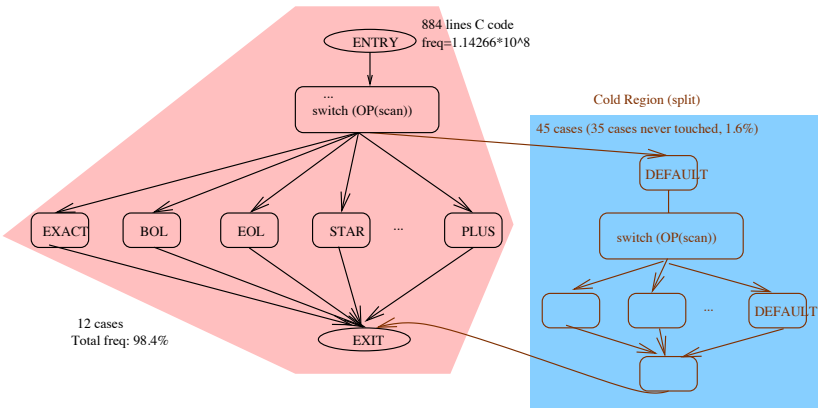


Figure 5: Partitioning *regmatch* in `perlbmk`

of C code evenly distributed through 57 cases. But only 12 cases are hot. This is because some expression matchings, such as exact matching (`EXACT`) and matching zero (or one) or more times (`STAR` and `PLUS`), occur often. However, exotic matchings, such as matching a string from n to m times and matching a string backward, seldom occur at runtime.

Cold cases introduce several problems: (1) they increase the size of the function that hosts the switch-case, which prevents inlining; (2) the code for cold case actions, which is intertwined with the hot case actions, pollutes the instruction cache; (3) cold cases may slow case selection by increasing the depth of the comparison tree or cause inefficient usage of memory by the slots for cold cases in the jump-table. Separating the cold cases and their actions from the hot cases ameliorates all these problems.

Switch-case Partitioning first partitions a large switch-case S into two: a hot switch-case S_h containing the hot case selection and the hot action code, and a cold switch-case S_c with the remainder cases and code of S. After this re-organization, a new, simple and fast, tree-based splitting technique can split S_c out of the host function. The combination of switch-code partitioning and splitting elegantly solves the problems caused by cold cases: (a) the host function becomes smaller and is more amenable to inlining; (b) the hot cases are placed together without perturbation from cold cases at runtime; and (c) the execution path for the selection of hot cases becomes shorter.

Figure 5 shows the partition of S into S_h and S_c in *regmatch*. The cold cases are clustered

```
CaseProfiling(S, key_value)
1.   index ← Search(key_value, S.Case_Values);
2.   if (index ≤ S.num)
3.      S.Freq[index] ← S.Freq[index] + 1;
4.   else // Modification to the profiling library of ORC 2.0
5.      default_idx ← Search(key_value, S.Default_Values);
6.      if (default_idx ⩾ 0)
7.         S.Default_Freq[default_idx] ← S.Default_Freq[default_idx] + 1;
8.      else
9.         default_idx ← Insert(key_value, S.Default_Values);
10.        S.Default_Freq[default_idx] ← 1;
```

Figure 6: Add Profiling for Default Cases.

into S_c and placed into the default action of S_h. After the re-organization, S_c forms a natural cold region and can be easily split out of the host function. Not all large switch-cases can be partitioned in the way shown in Figure 5, see Section 4.2.2 for details.

4 Implementation

This section introduces our extension to the existent profiling library in ORC 2.1 (Section 4.1), which is necessary for hot default case promotion. Then the detailed implementation of hot default case promotion and switch-case splitting (Section 4.2) is discussed.

4.1 Profiling for Default Cases

To enable hot default case promotion, the compiler needs precise information about the default case value distribution. A straightforward extension to the runtime instrumentation library of ORC 2.1, shown in Figure 6, records the frequency of the most frequent values that trigger the default action. Whenever a switch-case S is executed, the original instrumentation in ORC 2.1 invokes a function called CaseProfiling to update a frequency array, $S.Freq$, according to the current key_value. Our value profiling extension, shown in Figure 6, inserts instrumentation in the action code of the default case. The extension creates a new array called $S.Default\_Values$. Each element of this array contains a key value and its corresponding frequency. Whenever the default action is selected, either the frequency counter of an existing default value is incremented, or a new element is added

to the $S.Default\_Values$ array.

At the end of the instrumented execution, values from $S.Default\_Values$ must be written into the feedback file. The number of distinct default values seldom exceeds 50. Thus, the default value profiling extension writes only the 100 most frequently values into the feedback file.

4.2 Hot Default Case Promotion and Switch-Case Splitting

In preparation for switch-case splitting, a prepass summarizes the feedback information, promotes hot default cases, and clusters cases based on control flow information.

Summarize the switch-case. The total frequency of a switch-case S is:

$$S.total\_freq = \sum_{i=1}^{S.num+1} S.Freq[i] \tag{2}$$

where $S.num$ is the number of cases in S and $S.Freq[S.num + 1]$ is the frequency of the default case of S.

Promote hot default cases. The default case values are sorted by frequency from high to low. Some default cases of S are promoted, as shown in Figure 3, when they together dominate the execution of the switch-case:

$$\frac{\sum_{j=1}^{MaxPromotionNum} DefaultFreq[j]}{S.total\_freq} \geqslant PromotionThreshold \tag{3}$$

Currently, $PromotionThreshold$ is 99% and $MaxPromotionNum$ is 5.

Cluster cases according to the control flow information. If there is no transfer of control between different cases in S, identifying hot cases and cold cases is the result of a simple analysis of the feedback information. However, inter-case control flow is often found in application programs. For instance, a common programming trick is to let control fall through from case i to case $i + 1$. Moreover, an action may end with an explicit *goto* that transfers control to an arbitrary label in the program.[1] Therefore, the feedback information alone is not sufficient to identify hot actions. Consider a program where the hot action of a case A falls through to the action of another case

[1] This situation appears frequently in heavily hand-optimized applications such as `perlbmk`.

B. The action of case B is also hot, even though its frequency in the feedback information may be low. In this circumstance, A and B must be an atomic unit for the splitting analysis.

A *goto* transfers control from a *source* location to a *destination* location. Given a switch-case S, we say that a *goto* g is in S, $g \in S$, if both the source and destination locations are within S. If $g \in S$, we refer to a $g.source\_case$ and to a $g.destination\_case$. The algorithm PREPASS, shown in Figure 7, computes case groups based on control flow. First, each case is assigned to a distinct group ranging from 1 to $S.num + 1$, where $S.num$ is the number of the enumerated cases in the switch-case S (steps 2-3). Two situations are of interest: "fall-through" between subsequent cases (steps 4-7) and *goto*s that are in S and whose source and destination cases are distinct (steps 8-10).

MERGEGROUPS, called by PREPASS, merges two case groups into one. Whenever two groups are merged, their members are assigned the same new $merged\_group\_id$ (step 3). MERGEGROUPS also calculates the execution frequency of the merged group, which is the sum of the invocation frequency of its member cases (step 4).

```
PREPASS(S)
1.   merged_group_id ← S.num + 2;
2.   foreach i from 1 to S.num + 1;
3.           case[i].group ← i;
4.   foreach i from 1 to S.num
5.           if (case[i] falls through to case[i + 1])
6.              then MERGEGROUPS(S, i, i + 1, merged_group_id);
7.                   merged_group_id ← merged_group_id + 1;
8.   foreach g ∈ S such that g.source_case ≠ g.destination_case
9.           MERGEGROUPS(S, g.source_case, g.destination_case, merged_group_id);
10.          merged_group_id ← merged_group_id + 1;

MERGEGROUPS(S, i, j, new_id)
1.   S.Freq[new_id] ← 0;
2.   foreach k such that case[k].group = case[i].group or case[k].group = case[j].group
3.           case[k].group ← new_id;
4.           S.Freq[new_id] ← S.Freq[new_id] + S.Freq[k];
```

Figure 7: Case Clustering.

4.2.1 Switch-Case Partitioning Benefit Estimation

After the prepass phase, the compiler analyzes the summarized information to decide whether a switch-case should be split. If the decision is yes, the compiler also find the cases that should be

split out of the original switch-case. The input for this analysis is the switch-case S annotated with the groups formed by the prepass phase.

PartitionAnalysis, shown in Figure 8, sorts the case groups according to their frequencies from high to low. Then the algorithm scans the case groups and accumulates their execution frequency. When the accumulated frequency reaches $Freq\_Threshold$ (99% in our work), the scanning stops. The decision to split the cold cases is based on the size of the cold cases. If $ColdSize$ is larger than a set threshold, the switch-case is first reorganized into a hot switch-case and a cold switch-case as illustrated in Figure 5, then the cold switch-case is split out of the host function.

```
PartitionAnalysis(S)
1.  AccuFreq ← 0;
2.  HotSize ← 0;
3.  HotGroups ← ∅;
4.  foreach non-empty group i from the most frequent to the least frequent
5.          AccuFreq ← AccuFreq + S.Freq[i];
6.          HotSize ← HotSize + S.Size[i];
7.          if (AccuFreq / total_freq ⩽ Freq_Threshold)
8.             then HotGroups ← HotGroups ⋃ i;
9.          else break; // terminate the loop
10. ColdSize ← S.total_size − HotSize;
11. if (ColdSize > Size_Threshold)
12.    then ColdSwitch ← PartitionSwitchCaseStmt(S, HotGroups);
```

Figure 8: Partition Benefit Analysis.

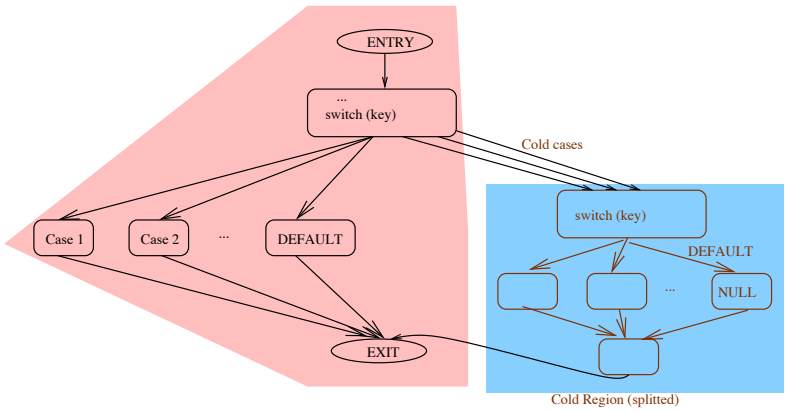


Figure 9: Partitioning a Switch-case with Hot Default Cases

4.2.2 Partitioning Switch-Cases With Hot Default

Not all large switch-cases can be re-organized in the way shown in Figure 5. When the $default$ case is seldom executed, as is the case in Figure 5, the original $default$ is simply placed into the cold switch-case. However if the original $default$ case is frequently executed, moving it into the cold switch-case is troublesome for two reasons: (1) a hot case is still mixed with the cold cases; and (2) if splitting is applied, many additional function calls will occur at runtime. Therefore two classes of switch-cases must be treated separately: *hot default* and *cold default* switch-cases. For *hot default* switch-cases the re-organization shown in Figure 9 works well. The main difference with the re-organization illustrated by the example in Figures 4 and 5 is that now the default action remains in the hot switch-case. Also the case selection of cold cases is kept intact. The actions of cold cases are replaced with *goto*s that transfer control to the new cold switch-case. The original cold actions are *moved* into the cold switch-case. After this transformation, if the *key* contains a cold value, its case selection requires: (1) the original case selection; (2) a function call; and (3) a second case selection in the cold switch-case. If the cold cases are indeed cold, this additional function call will happen infrequently and thus have a negligible cost.

```
PartitionSwitchCaseStmt(S)
1.  if (C_d is hot)
2.     ColdSwitch ← CreateSwitch(S.ColdCases, NULL);
3.     NewFunc ← SplitAndPatch(ColdSwitch);
4.     foreach (C_i, A_i) in S.ColdCases
5.              Replace(A_i, NewFunc);
6.  else
7.     OrigDefault ← (C_d, A_d);
8.     NewColdCases ← S.ColdCases − OrigDefault;
9.     ColdSwitch ← CreateSwitch(NewColdCases, OrigDefault);
10.    NewFunc ← SplitAndPatch (ColdSwitch);
11.    foreach (C_i, A_i) in S.ColdCases
12.             Delete(C_i, A_i);
13.    Replace(A_d, NewFunc);
14. RepairFeedbackInformation(S);
15. return ColdSwitch;
```

Figure 10: Partition and Split Switch-case.

PartitionSwitchCaseStmt, shown in Figure 10, partitions a switch-case into two and then splits the new cold switch-case out of the host function. The algorithm actions are different for

cold and hot default situations. In the algorithm, C_i represents case i and C_d is the default case; A_i is the action code for case i and A_d is the default action code. CreateSwitch generates a new switch-case, *ColdSwitch*, containing the cases listed on CreateSwitch first parameter and the default case that appears in CreateSwitch second parameter. When the default case is hot (steps 1-5), the default action of the cold switch-case is empty. Replace replaces the actions of the cold cases in the original switch-case with *goto*s to a call site which calls $NewFunc$.

When the default case is cold (steps 7-13), the cold switch-case is created with the cold cases, including the cold default. This cold switch-case is split into $NewFunc$. Then the cold cases and their action codes are deleted from the original switch-case. The default action of the original default case is replaced with a call to $NewFunc$.

SplitAndPatch extracts the cold switch-case into a new function $NewFunc$. To preserve program semantics, variable accesses and changes to control flow must be handled correctly. The process of function splitting is described in [1].

5 Experimental Study

This section presents an experimental investigation of switch-case optimization on standard benchmarks. The results of this investigation may be summarized as follows:

- Switch-case optimizations yield non-trivial performance improvement in some standard benchmarks — up to 4.96% compared with the straightforward jumptable (JT) approach.
- For the Itanium architecture, a linear search using branches (BR) executes a lot more instructions, in benchmarks with frequent switch-cases, when compared with the JT strategy.
- Partitioning large switch-cases effectively reduces the size of functions and significantly reduces the number of branch miss-prediction stalls (up to 16%). However, it does not improve I-cache efficiency for Itanium processors because of Itanium's large I-caches.

Experimental Framework: This research uses the Open Research Compiler 2.1 (ORC) as an experimental platform. ORC is an open-source compiler that evolved from the SGI's MIPSPro compiler, which implements a rich set of optimizations including Inter-Procedural Optimizations (IPO) and complete program analysis support. ORC generates binaries for Intel's 64-bit Itanium processors. ORC 2.1 uses BR, JT and COMB strategies for switch-cases and implements hot case hoisting (HH). We added hot default case promotion (DP) and switch-case partition (SP).

Experiment Configuration: Experimental results were obtained on an HP ZX6000 workstation with a 1.3GHz Itanium-2 processor, 1 GB of main memory, 32KB of L1 cache, 256KB of L2 Cache, and 1.5MB of on-die L3 cache. The operating system is Red Hat Linux 7.2 with a 2.4.18 kernel. This experimental study is based on SPEC2000 integer benchmarks that contain switch-cases: `bzip2`, `crafty`, `gcc`, `perlbmk`, `vortex` and `vpr`. Time is measured by the Linux *time* command and micro-architectural benchmarking is obtained with *pfmon*. All reported run-times are the average of 5 consecutive identical runs.

5.1 Statistics for Switch-Cases

Important questions when planning a code transformation in a compiler include: (i) how often do opportunities to apply the transformations appear in the source code of standard benchmarks?; and (ii) how frequently are the sites of these opportunities expected to be executed? For the code transformations studied in this paper the number of cases in switch-cases and the occurrence of hot default actions are also relevant.

Table 1 presents the answer to these questions for the six SPEC2000 benchmarks included in this study. The first row contains a static count of the number of switch-cases occurring in the source code of each benchmark. The *Case Number Distribution* rows shows that the number of cases in each switch-case varies widely. Script parsers (`perlbmk`) and compilers (`gcc`) tend to contain statements with many cases. The data in the *Maximum Cases* row reports the number of cases in the statement with most cases in each benchmark.

Optimizing case dispatch and code layout for switch-cases only pays off when these statements are frequently executed. The execution frequency of switch-cases is presented in Table 1 as a frequency distribution. This distribution is measured by running the benchmarks with the standard training input set from the SPEC2000 suite. Only a small subset of the switch-cases are frequently executed and thus relevant for the application's performance. For instance, `gcc` executes an extensive series of optimizations and transformations over the input. Although `gcc` contains 374 switch-cases, none of them are executed more than 10^6 times.

The last row of Table 1 reports the number of switch-cases that have a hot default case, *i.e.* a default action that is executed in more than 90% of the executions of its switch-case. This frequency study indicates that, if these switch-cases are executed often, hot default case promotion may improve `crafty`, `perlbmk`, `vortex` and `gcc`.

Benchmarks		`bzip2`	`crafty`	`vpr`	`vortex`	`perlbmk`	`gcc`
Number of Switches		3	42	12	37	127	374
Case Number Distribution	<6	3	17	12	11	68	199
	$7 \sim 15$	0	25	0	22	30	117
	$16 \sim 30$	0	0	0	24	15	32
	$31 \sim 100$	0	0	0	0	11	21
	>100	0	0	0	0	3	5
Maximum Cases		4	13	6	30	243	398
Frequency Distribution	$10^6 \sim 10^7$	1	2	0	6	13	0
	$> 10^7$	1	3	0	0	7	0
Hot default		0	1	0	3	25	45

Table 1: Statistics of Switch-cases

Figure 11 illustrates the unevenness of case execution frequencies. Each point in the figure represents a switch-case that is executed more than 10^6 times in the training run. In the vertical axis is the number of hot cases that collectively account for more than 99% of the invocation of the statement. Points close to the X-axis represent statements where a few cases dominate the execution time. Hot case hoisting (HH) should benefit such statements. For example, in 5 of the 6 frequently executed switch-cases in `vortex`, a single case is executed more than 99% of the time. A similar situation is observed for `perlbmk` and `bzip2`. On the other hand, the execution is almost evenly distributed for `crafty`.

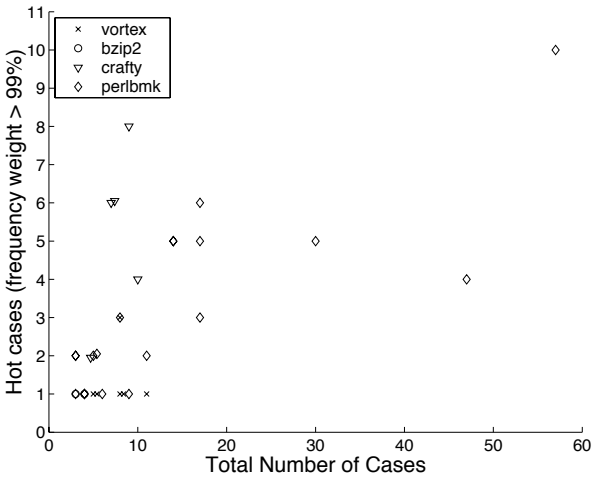


Figure 11: Skewed Frequency Distribution Among Cases

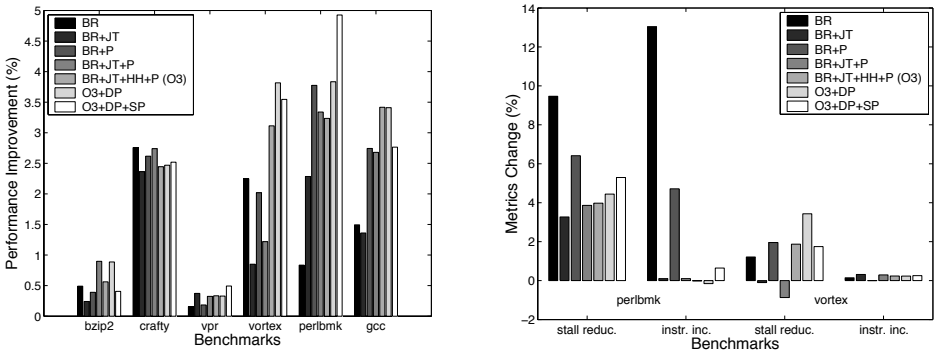
5.2 Performance Analysis

Short	Explanation
JT	Always use Jump Table Strategy, no profiling.
BR	Always use linear search (BR) strategy (linear search), no profiling.
BR+P	Always BR with profiling used to reorder cases.
BR+JT	Use BR strategy if less than 7 cases, otherwise use JT, no profiling.
BR+JT+P	BR+JT with profiling used to reorder cases.
O3	BR+JT+P with hot case hoisting (HH).
O3+DP	O3 with default case promotion (DP).
O3+DP+SP	O3 with DP and switch-case partition (SP).

Table 2: Optimization Techniques

This section presents a performance study of several strategies to generate code for switch-cases. Table 2 summarizes the strategies studied. The performance improvements presented in Figure 12(a) are in relation to the jump table strategy (JT), which is used as a baseline in this study.

The performance varies across benchmarks. This is due to variations in the execution frequency of individual switch-cases. For instance, switch-cases in `bzip2` and `vpr` are rarely executed, making the optimizations studied of little relevance. The study shows, however, that optimizing the code generated has a significant performance impact when switch-cases are executed frequently. In our



(a) Runtime Performance.

(b) Micro-architectural Study.

Figure 12: Runtime results.

study, `perlbmk` can be improved by 4.96%. For other benchmarks (such as `crafty`, `vortex` and `gcc`), the performance improvement ranges from 1.3% to 3.8%. All three versions of O3 produce good performance and no single combination of optimizations produces the best performance for all benchmarks. Therefore, in applications that matter, experimenting with these techniques is recommended.

Feedback-guided compilation is recommended because it consistently generates faster executables. For instance, `perlbmk` compiled under the BR+P strategy is about 3% faster than the one compiled with only the BR strategy. Moreover, all the other techniques (HH, DP and SP) depend on representative profiling to ensure effective speculations.

The new techniques described in this paper, Hot default case promotion (DP) and switch-case partition (SP), perform well in some benchmarks. DP generates the fastest code for `vortex` and SP generates the fastest code for `perlbmk`. Hot switch-cases in `vortex` with dominant default cases (Figure 11) explain DP's effectiveness. Several hot and very large switch-cases in `perlbmk` make the SP strategy successful.

A somewhat surprising observation is the limited effectiveness of the jump table strategy even when combined with linear search. This result may be particular to the Itanium family processors and to the code scheduler in this compiler. Itanium-2 processors can execute 6 instructions (2 bun-

dles) in the same cycle. The runtime of a program depends not only on the number of instructions executed, but also on the effective utilization of instruction slots. This architecture is favorable to the search strategy that executes a series of independent compare-and-branch instructions. Theoretically, six cases can dispatched in 4 bundles (2 bundles for comparing with 6 case values and 2 bundles for 6 predicated branches). On the other hand, the jump table strategy needs to calculate the address of the case entry in the jump table and then load it from memory, which is much more expensive than register-based instructions.

5.3 Micro-architecture Level Benchmarking

This section investigates how the optimization techniques studied change the micro-architecture behavior of two benchmarks: `perlbmk` and `vortex`. As evidenced in Figure 12(a), the effect of these techniques in `bzip2`, `vpr` and `gcc` is limited. The switch-cases in `crafty` are simple, with no more than 7 cases, and there is not much variation in the performance with the different techniques. Thus, the effects of case dispatching in `perlbmk` and `vortex` are of particular interest. Figure 12(b) shows the variations in two metrics, the number of retired instructions and the number of processor stalls, for `perlbmk` and `vortex` with each technique when compared with the jump table strategy baseline.

In `perlbmk` the linear search strategy without feedback (BR) results in 13.5% more retired instructions than the pure jump table strategy but it reduces the number of pipeline stalls by 9.5%. This is because linear search often needs to enumerate more case values, which requires the execution of more instructions, to hit the real targets. But these enumerations are independent of each other and thus can be executed in parallel. On the other hand, a pure jump table strategy needs a memory load operation to load the destination address. These loads cause more stalls because a memory operation might require several cycles to complete.

For `perlbmk`, while linear search with feedback information (BR+P) reduces the number of instructions executed by 13.5%, this reduction is only 4.7% for linear search without feedback information (BR). This result argues for the use of profiling feedback: it allows the compiler to

place the frequent case values earlier in the search list, thus shortening the enumeration phase.

Changes to the number of instructions executed in `vortex` are negligible when compared with those in `perlbmk`. The reason is two fold. First, `perlbmk` employs many more switch-cases, giving more weight to switch-case optimizations. Second, the switch-cases in `perlbmk` often contain more cases and larger code than those in `vortex`. Therefore, the linear search strategy executes many more compare and branch pairs to enumerate possible case values.

Most processor stalls are caused by data cache misses, branch mis-prediction stalls, or instruction cache misses.[2] Our experiments show that, for a certain benchmark, there are very small differences in instruction cache stalls and data cache stalls among the optimization techniques. Although not very obvious, this result is not entirely surprising. The Itanium family processors have extensive instruction caches that make instruction cache stalls less important than in other processors. Our data shows that instruction cache stalls account for no more than 4% of the total stalls in SPEC2000 integer benchmarks; data cache stalls account for 73% to 85% and branch mis-prediction stalls account for 7% to 22%.

Retired instructions increase a very small percentage in O3+DP+SP strategy, implying that our switch-case partition component only splits the cold cases out of the host functions. Even with an increase in retired instructions, O3+DP+SP still improves runtime performance of `perlbmk`. Because we don't invoke inlining, the benefit comes from the simplified case dispatching and processor stall reduction. Processor level benchmarking shows that SP reduces branch mis-prediction stalls by about 2.9%, while other stalls remain similar.

5.3.1 Function Body Reduction by Switch-Case Partitioning

Besides the performance improvements due to reduced stalls, switch-case partitioning has another very important advantage: it can effectively reduce the size of a function by splitting infrequently executed cases into a new function. After splitting, the host function, which is frequently invoked, is more amenable for inlining. Although inlining is out of the scope of this paper, we are interested

[2]Other factors lead to pipeline stalls, such as insufficient floating point or register stack units. However, these factors contribute no more than 5% of the total stalls and vary very little for different optimizations in our study.

in the function size reduction achieved by SP. We compared the sizes of the functions in binary format before and after SP. For `perlbmk`, where 23 switch-cases are partitioned, the function size reduction ranges from 3% to 79.7%, with an average of 36.3%. In `vortex`, where 6 partitions occur, the function size reduction ranges from 1.3% to 53.6%, with a average at 25.1%. Most of the partitioned functions are very large, some with more than 1000 lines of C code. These functions cannot be inlined in most compilers. The size reduction obtained by SP should enable further inlining. Moreover, the better cache behavior obtained by improved code placement of smaller functions should have greater impact on performance in processors with small I-caches.

6 Related Work

Previous research on transformation for switch-cases focused on efficient case selection [2, 3, 9, 10, 11, 12, 14]. Our experimental platform, the Open Research Compiler (ORC), implements the methods in [3, 11, 12].

Sale summarizes and compares different case-selection strategies on the B6700/7700 series [14]. These strategies include jump table, linear search, comparison tree search and a specialized search utilizing a specific CISC instruction for searching tables.

Hennessy and Mendelsohn combine the jump table approach and comparison tree search approach to transform case statements in the Pascal★ compiler [11]. Cases are clustered according to their proximity. Case dispatch is implemented in two levels: the first level uses comparison tree search to find the cluster and the second level uses jump table to find the case action within the cluster.

Bernstein asserted that the problem of splitting the cases in a switch-case statement into a minimum number of clusters of a given density was NP-complete and devised greedy heuristics to clustering the cases [3]. Bernstein's techniques were implemented in several compilers, including the Objective Caml compiler [13]. However, Kannan and Proebsting later showed that Bernstein's problem modeling was wrong and presented an algorithm to find the solution in $O(n^2)$ time where n is the number of the cases in the switch-case statement [12]. Bernstein also suggested that cases

could be reordered according to their execution frequency.

Bumbulis and Cowan [4] discovered that when implementing scanners, it is profitable to replace large switch-cases by a series of `if` statements based on the number of distinct subranges divided by the total number of cases in the multi-way decision logic. The compiler optimization presented in this paper eliminates the need for application designers to make such decisions.

The philosophy of hot-cases-first is also used in supporting polymorphism in object-oriented programming languages. Jump-table (or virtual function table) could be used for dispatching the control to the correcting desination method. But it is also expensive. Some researchers proposed to find the destination methods that dominating the invokation of a call site and enumerat these dominating methods before turning to the jump-table approach [5].

These are some works that assume the switches have been transformed to a series of conditional branches and try to reorder these branches according to importance [16, 17] or convert them into indirect jumps [15]. Comparatively, our approach is more straightforward and natural because we optimize switches directly. Also, as our work shows, using indirect jumps should be avoided in processors with many functional units such as Itanium II.

Conclusion

This paper reviewed existent optimization techniques for switch-case statements and proposed two new speculative approaches to take advantage of the skewed execution frequency among the cases. A thorough investigation of the transformation strategies for switch-cases showed their impact on runtime performance, processor-level behavior and function sizes. This work shows that the effectiveness of the techniques depends on the program's characteristics such as the time spent on switch-case statements, the number of cases, and the case frequency distribution. When switch-case statements are frequently executed, such as in `perlbmk`, code optimization can yield significant performance gain.

References

[1] Technical Report reference ommited for double-blind reviewing.

[2] L. V. Atkinson. Optimizing two-state case statements in PASCAL. *Software – Practice and Experience*, 12(6):571–581, June 1982.

[3] R. L. Bernstein. Producing good code for the case statement. *Software – Practice and Experience*, 15(10):1021–1024, Oct 1985.

[4] P. Bumbulis and D. D. Cowan. Re2c: A more versatile scanner generator. *ACM Letters on Programming Languages and Systems*, 2(1-4):70–84, March-December 1993.

[5] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 397–408, Portland, Oregon, 1994.

[6] J. W. Davidson and A. M. Holler. A model of subprogram inlining. Technical report, Computer Science Technical Report TR-89-04, Department of Computer Science, University of Virginia, July 1989.

[7] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software - Practice and Experience (SPE)*, 18(8):775–790, 1989.

[8] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering (TSE)*, 18(2):89–102, 1992.

[9] U. Erlingsson. Lucid and efficient case analysis. Technical Report TR-95-14, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 1995.

[10] U. Erlingsson, M. S. Krishnamoorthy, and T. V. Raman. Efficient multiway radix search trees. *Information Processing Letters*, 60(3):115–120, 1996.

[11] J. L. Hennessy and N. Mendelsohn. Compilation of the Pascal case statement. *Software – Practice and Experience*, 12:879–882, 1982.

[12] S. Kannan and T. A. Proebsting. Short communication: Correction to *producing good code for the case statement. Software – Practice and Experience*, 24(2):233–233, Feb 1994. Erratum for [3].

[13] F. Le Fessant and L. Maranget. Optimizing pattern matching. In *International Conference on Functional Programming*, pages 26–37, Florence, Italy, 2001.

[14] A. Sale. The implementation of case statements in pascal. *Software – Practice and Experience*, 11(9):929–942, September 1981.

[15] G. R. Uh. *Effectively Exploiting Indirect Jumps.* PhD thesis, Florida State University, Tallahassee,FL, December 1997.

[16] M. Yang, G.-R. Uh, and D. B. Whalley. Improving performance by branch reordering. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 130–141, 1998.

[17] M. Yang, G.-R. Uh, and D. B. Whalley. Efficient and effective branch reordering using profile data. volume 24, pages 667–697, Nov 2002.

[18] P. Zhao and J. N. Amaral. To inline or not to inline, enhanced inlining decisions. In *16th Workshop on Languages and Compilers for Parallel Computing*, pages 405–419, College Station, TX, Oct 2003.