Feedback-Directed Switch-Case Statement Optimization

Peng Zhao and José Nelson Amaral
Department of Computing Science, University of Alberta, Edmonton, Canada

E-mail: pengzhao@cs.ualberta.ca, amaral@cs.ualberta.ca

Abstract

This paper presents two new feedback-guided techniques to generate code for switch-case statements: hot default case promotion (DP) and switch-case statement partitioning (SP). DP improves case dispatch while SP simplifies case dispatch, improves instruction layout and enables further inlining. An extensive experimental study reveals up to 4.9% performance variations among different strategies. The largest performance improvement of DP and SP over existing O3 optimization in the Open Research Compiler (ORC) is 1.7%. A micro-architecture level performance study provides insights on the basis for this performance improvement.

1 Introduction

Switch-case statements are frequently used to express multi-way branch semantics in script interpreters, compilers and virtual machines. 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 phases: case selection and case action [10]. If the key does not match any enumerated case value during case selection, the default action code is executed.

Known techniques for the generation of code for switch-cases that do not rely on feedback information include: **search strategy (BR)** that implements a series of branch-if-equal operations; **jump table strategy (JT)** used for large number of case values; and a **combined strategy (COMB)** that is used when case values are distributed into several dense clusters.

The main contributions of this paper are: **(1)** Two new techniques (DP and SP described in Section 2) that use feedback information to improve switch-cases; **(2)** An implementation of DP and SP in the ORC 2.1 compiler (Section 3); and **(3)** An experimental study (Section 4) that includes not only the new techniques but also known techniques such as BR and JT.

2 Feedback-Guided Optimization

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. 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): The default action may be frequently selected by a few key values. DP creates new case values for the frequent (*hot*) values that originally fell in the default category. Hot values are identified by runtime profiling. After promotion, standard HH moves these new values to an earlier place in the branch search or hoist them before a jump table according to their frequency. For instance, in Figure 1(a), the key value 9 occurs very often. With DP, the value 9 is promoted to a separate case value (Figure 1(b)).

Existing techniques are inefficient for hot defaults. 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 enumerated 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 [3, 4, 13]. The actions of infrequent (*cold*) cases often account for a significant portion of the size of a switch-case. For instance, Figure 2 shows the breakdown of the cases in a switch-

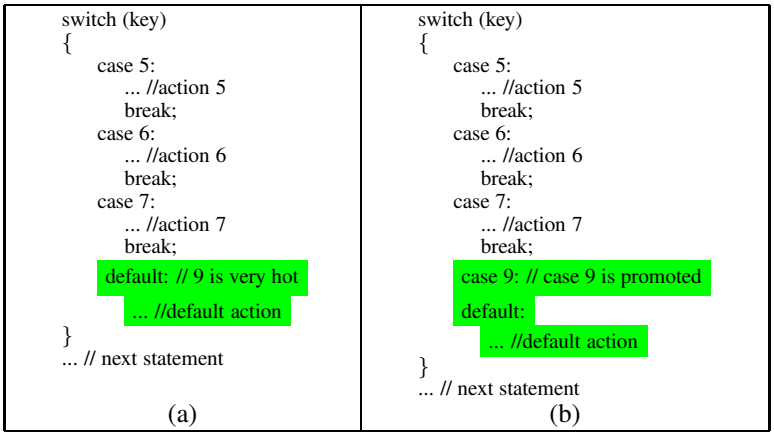


Figure 1. Hot Default Case Promotion

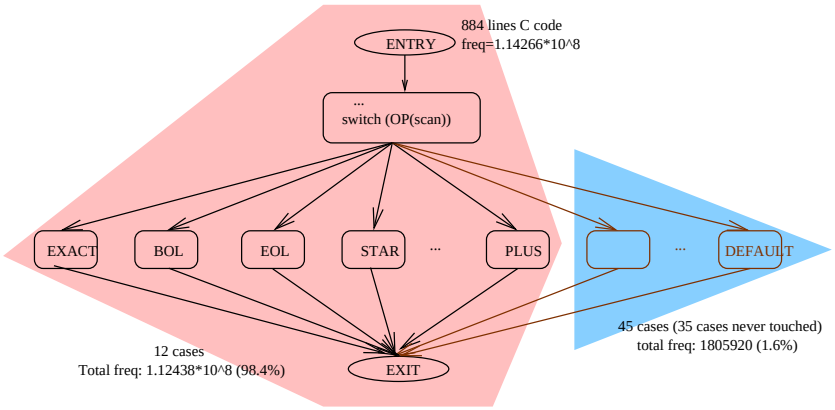


Figure 2. Annotated CFG of function $regmatch$ in `perlbmk`

case, S, in the function $regmatch$ in the SPEC2000's `perlbmk` program. This function matches regular expressions with strings in a file. S requires about 884 lines of C code evenly distributed through 57 cases. But only 12 cases are hot.

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.

SP first partitions a large switch-case S into two: a hot switch-case S_h and a cold switch-case S_c. 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 3 shows the partition of S into S_h and S_c in $regmatch$. The cold cases are placed into the default action of S_h. Then S_c forms a natural cold region and can be easily split out of the host function.

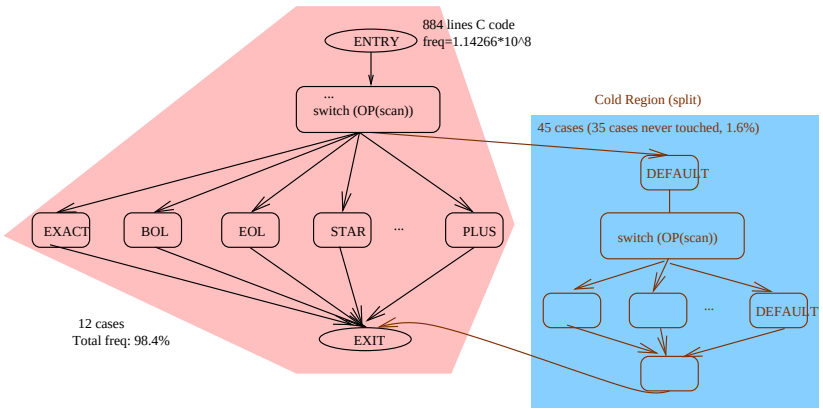


Figure 3. Partitioning $regmatch$ in `perlbmk`

```
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.    dflt_id ← SEARCH(key_value, S.Dflt_Values);
6.    if (dflt_id ⩾ 0)
7.      S.Dflt_Freq[dflt_id] ← S.Dflt_Freq[dflt_id] + 1;
8.    else
9.      dflt_id ← INSERT(key_value, S.Dflt_Values);
10.     S.Dflt_Freq[dflt_id] ← 1;
```

Figure 4. Add Profiling for Default Cases.

3 Implementation

This section introduces an extension to the existent profiling library in ORC 2.1 (Section 3.1), and the implementation of hot default case promotion and switch-case splitting (Section 3.2).

3.1 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 4, 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. This extension inserts instrumentation in the action code of the default case. Each element of a new array, $S.Dflt\_Values$, 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 $S.Dflt\_Values$. The number of distinct default values seldom exceeds 50. Thus, at the end of the instrumented execution, the extension writes only the 100 most frequently values into the feedback file.

3.2 Hot Default Case Promotion

A prepass summarizes the feedback information, promotes hot default cases, and clusters cases based on control flow information. The total frequency of a switch-case S is:

$$S.total\_freq = \sum_{i=1}^{S.num+1} S.Freq[i] \qquad (1)$$

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

To 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 1, when they together dominate the execution of the switch-case (currently the promotion threshold, PT, is 99% and $MaxPromotionNum$ is 5):

$$\frac{\sum_{j=1}^{MaxPromotionNum} DefaultFreq[j]}{S.total\_freq} \geqslant PT \qquad (2)$$

Inter-case control flow is often found in application programs. 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 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.

Given a switch-case S, a *goto* g is in S, $g \in S$, if both the source and destination locations are within S. If $g \in S$, there is a $g.source\_case$ and to a $g.destination\_case$. The algorithm PREPASS, shown in Figure 5, 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).

```
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.dest_case
9.    MERGEGROUPS( S, g.source_case, g.dest_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 5. Case Clustering.

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). The execution frequency of the merged group is the sum of the invocation frequency of its member cases (step 4).

3.2.1 Switch-Case Partitioning Benefit Estimation

PARTITIONANALYSIS, shown in Figure 6, 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. If $ColdSize$ is larger than a threshold, the switch-case is split into S_h and S_c, as illustrated in Figure 3, and S_c is moved out of the host function.

3.2.2 Partitioning Switch-Cases With Hot Default

When the $default$ case is seldom executed, as is the case in Figure 3, the original $default$ is simply placed into the cold switch-case. However if the original

```
PARTITIONANALYSIS(S)
1.  AccuFreq ← 0;
2.  HotSize ← 0;
3.  HotGroups ← ∅;
4.  foreach non-empty group i from most to 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 ← PARTITIONSTMT(S, HotGroups);
```

Figure 6. Partition Benefit Analysis.

```
PARTITIONSTMT(S)
1.  if (C_d is hot)
2.    ColdSwitch ← CREATE(S.ColdCases, NULL);
3.    NewFunc ← SPLITANDPATCH(ColdSwitch);
4.    foreach (C_i, A_i) in S.ColdCases
5.          REPLACE(A_i, NewFunc);
6.  else
7.    OrigDflt ← (C_d, A_d);
8.    NewColdCases ← S.ColdCases − OrigDflt;
9.    ColdSwitch ← CREATE(NewColdCases, OrigDflt);
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 7. Partition and Split Switch-case.

$default$ case is hot, 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. The re-organization shown in Figure 8 is used for *hot default* switch-cases. Now the default action remains in the hot switch-case and the case selection of cold cases is kept intact. The actions of cold cases are replaced with *goto*s to the new cold switch-case. The original cold actions are *moved* into the cold switch-case. After this transformation, the selection of a cold case 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.

PARTITIONSTMT, shown in Figure 7, 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. CREATE generates a new switch-case. 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

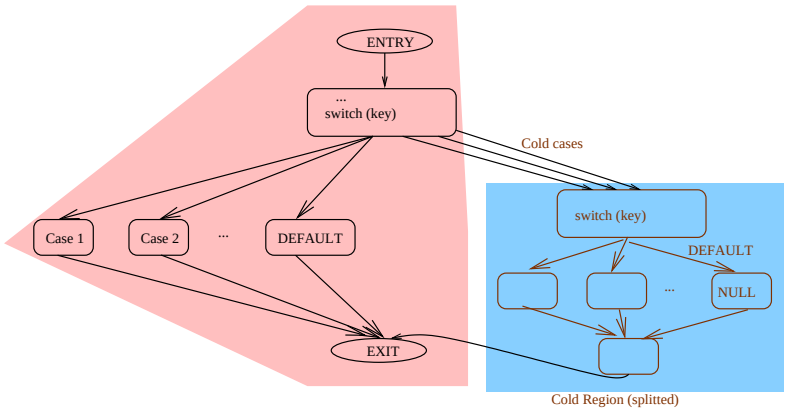


Figure 8. Partitioning a Switch-case with Hot Default Cases

*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 moved into $NewFunc$ by SPLITANDPATCH [14]. 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 $NewFunc$.

4 Experimental Study

The results of an experimental investigation using standard benchmarks may be summarized as follows:

- Switch-case optimizations yield up to 4.9% performance improvement over a straightforward jump table (JT) approach.
- In benchmarks with frequent switch-cases a linear search using branches (BR) executes many more instructions when compared with the JT strategy.
- Partitioning large switch-cases reduces the size of functions and significantly reduces the number of branch miss-prediction stalls (up to 16%).

The experimental platform is the Open Research Compiler 2.1 (ORC). ORC evolved from the SGI's MIPSPro compiler, which implements a rich set of optimizations including Inter-Procedural Optimizations (IPO). 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).

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`. Though `twolf` also contains switch statements, we don't include it because the switch statements are not executed at runtime. Micro-architectural benchmarking is obtained with *pfmon*. Run times are averaged over 5 consecutive identical runs.

4.1 Statistics for Switch-Cases

The first row of Table 1 contains a static count of the number of switch-cases in the source code of each benchmark. The *Case Number Distribution* rows show a wide variation in the number of cases in each statement. Script parsers (`perlbmk`) and compilers (`gcc`) tend to contain statements with many cases. The *Maximum Cases* row reports the maximum number of cases in a single statement. In the last row of Table 1 is the number of switch-cases that have a hot default action that is executed more than 90% of the time.

The execution frequency distribution of switch-cases using the SPEC2000 standard training input set is presented in Table 1. 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.

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

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

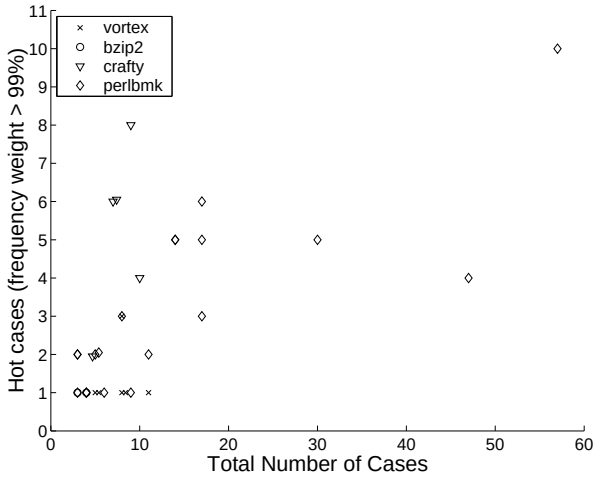


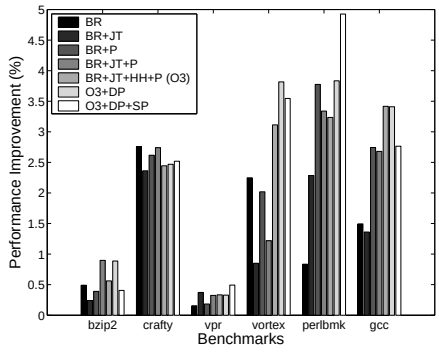
Figure 9. Case Frequency Distribution

Figure 9 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`.

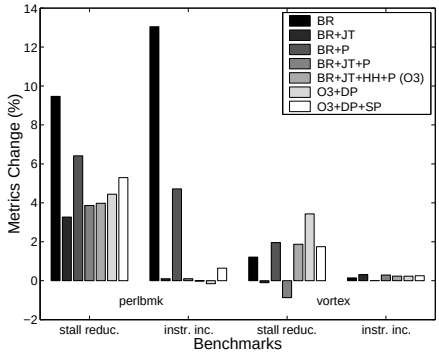
4.2 Performance Analysis

Table 2 summarizes the strategies studied. The performance improvements presented in Figure 10(a) are in relation to the jump table strategy (JT). Switch-cases in `bzip2` and `vpr` are rarely executed, making the optimizations studied of little relevance. On the other hand, more frequently executed switch cases in `perlbmk` result in improvement of 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.

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



(a) Runtime Performance.



(b) Micro-architectural Study.

Figure 10. Runtime results.

BR strategy. Moreover, all the other techniques (HH, DP and SP) depend on representative profiling to ensure effective speculations.

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 9) explain DP's effectiveness (improving performance over O3 by about 0.7%). Several hot and very large switch-cases in `perlbmk` make the SP strategy successful, improving performance over O3 by about 1.7%.

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 and to the code scheduler in this compiler. Itanium-2 processors can execute 6 instructions (2 bundles) 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 be 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.

4.3 Micro-architecture Level Benchmarking

This section investigates how the optimization techniques studied change the micro-architecture behavior of two benchmarks: `perlbmk` and `vortex`. Figure 10(b) shows the variations in the number of retired instructions and the number of processor stalls with each technique when compared with JT.

In `perlbmk` BR results in 13.5% more retired instructions than JT but it reduces the number of pipeline stalls by 9.5%. This is because BR often needs to enumerate more case values to find the desired target. But these enumerations are independent of each other and thus can be executed in parallel. On the other hand, JT needs a load for the destination address.

For `perlbmk`, while BR+P reduces the number of instructions executed by 13.5%, this reduction is only 4.7% for 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, BR executes many more compare and branch pairs to enumerate possible case values.

Most processor stalls are caused by Dcache misses, branch miss-prediction stalls, or Icache misses.[1] Our experiments show very small differences in Icache stalls and Dcache stalls among the strategies. This result is not surprising given Itanium's extensive Icaches. Icache stalls account for no more than 4% of the total stalls in SPEC2000 integer benchmarks; Dcache stalls account for 73% to 85% and branch miss-prediction stalls account for 7% to 22%.

The very small increase in retired instructions in O3+DP+SP implies that switch-case partition indeed

[1] 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.

splits only cold cases out of the host functions. O3+DP+SP improves the runtime of `perlbmk`. As there is no further inlining, the benefit comes from the simplified case dispatching and processor stall reduction. Processor level benchmarking shows that SP reduces branch miss-prediction stalls by about 2.9%, while other stalls remain similar.

4.3.1 Function Body Reduction by Switch-Case Partitioning

Switch-case partitioning reduces the size of functions. 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 better cache behavior obtained by improved code placement of smaller functions should have greater impact on performance in processors with small I-caches.

5 Related Work

Previous research on transformation for switch-cases focused on efficient case selection [1, 2, 5, 6, 7, 8, 10]. ORC implements the methods in [2, 7, 8].

In [7] cases are clustered according to their proximity in the Pascal★ compiler. Case dispatch is implemented in two levels: (1) comparison tree search to find the cluster and (2) jump table finds the case action.

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 [2]. Bernstein's techniques were implemented in several compilers, including the Objective Caml compiler [9]. 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 [8]. Bernstein also suggested that cases could be reordered according to their execution frequency.

Some approaches assume that switch cases have been transformed to a series of conditional branches and try to reorder these branches according to importance [12] or convert them into indirect jumps [11]. In contrast, our approach is more straightforward because we optimize switches directly.

Conclusion

This paper presented 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. 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.

6 Acknowledgements

This research is supported by the Natural Science and Engineering Research Council of Canada (NSERC) and could not be done without ORC. Special thanks do Sun C. Chan, Li-Ling Chen, and Shane Brewer.

References

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

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

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

[4] 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.

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

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

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

[8] 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 [2].

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

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

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

[12] M. Yang, G.-R. Uh, and D. B. Whalley. Efficient and effective branch reordering using profile data. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):667–697, Nov 2002.

[13] 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.

[14] P. Zhao and J. N. Amaral. Splitting functions. Technical Report TR04-18, Dept. of Computing Sciences, Univ. of Alberta, Edmonton, Canada, 2004.