Context-aware Code Optimization

Murat Bolat
Department of Electrical and Computer Engineering
University of Delaware
Newark, DE 19716-3130, USA
murat@udel.edu

Xiaoming Li
Department of Electrical and Computer Engineering
University of Delaware
Newark, DE 19716-3130, USA
xli@ece.udel.edu

***Abstract*—A program segment such as a function or a basic block sequence may display different behaviors during the execution of the program. For example, a basic block sequence may consistently show few cache misses during the first 10 times it is executed, while the same basic block sequence may experience high number of cache misses when the sequence is invoked in the next 100 times. The divergence within the runtime behavior of program segment implies that different optimization choices should be made even for a single code segment if it shows different behaviors under different runtime contexts. However, traditional compilation technique rarely, if ever, takes advantage of that optimization opportunity. This concept is different from the finding of hot path, which has been done in both static compilation and dynamic optimization, whose purpose is to find the most frequently executed code segment and then to apply more aggressive and more expensive optimizations on that code segment. For the most part, behavior divergence other than execution frequency is not a factor in the determination of which optimizations to be applied to the segment.**

In this paper, we propose a novel feedback-driven program optimization technique that profiles and determines the runtime behaviors of code segments in a program to find the different patterns of behavior, correlates different runtime behaviors of a program segment with its program source code, and uses an empirical search method to customize the choice of optimization for same program segments under different runtime behaviors. We implement our optimization technique in LLVM and test our approach with SPEC2000 and SPEC2006 benchmarks. The preliminary results show promising performance improvement compared with the standard optimization settings used by the benchmarks.

I. Introduction

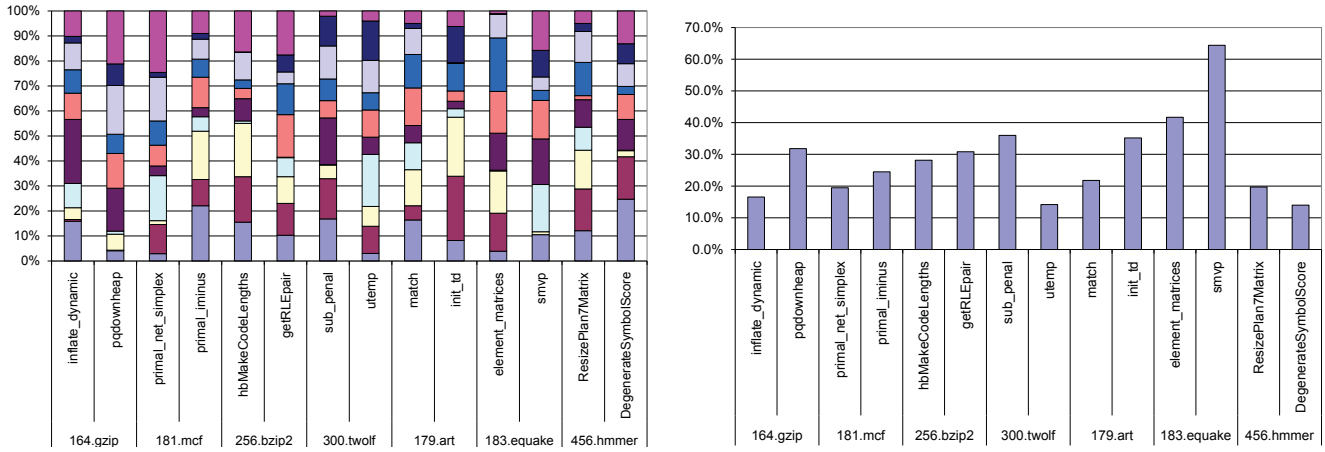
A code segment in a program may behave differently during the execution of the program when that code segment is invoked multiple times. The behavior difference is reflected in not only the time of multiple executions of the code segment, but also in the interaction between the code segment and architectural features, for example cache misses in the multiple instances of execution.

The behavior divergence of the same code segment can appear not only when the code segment is invoked by different preceding code but also can happen among multiple invocations of the code segment by the same preceding code. The fundamental reason for the behavior divergence is that the execution of preceding code of a code segment might change the execution context of the respective code segment, hence the segment is actually executed with different set of architectural resources. For example, one preceding basic block can allocate and load big chunk of memory which can result in a higher cache miss rate in the succeeding code segment, whereas another preceding basic block does not load any new memory which might lower cache miss rates of the same code segment.

The behavior divergence of code segment indeed exists in real-world programs. Figure 1 shows the profile of L2 cache misses of 14 functions, 2 functions in each of the seven SPEC benchmarks [1] [2]. The functions are all invoked more than twice during the execution of the whole benchmark. We record the L2 cache misses in each invocation of the functions. Figure 1(a) shows the distribution of those record of L2 miss on each function. The bar of each function has 10 segments, the i_{th} segment representing the percentage of the overall number of invocations of the function that has L2 cache miss in between $i * 10\%$ and $(i+1) * 10\%$ of $max\_L2\_miss - min\_L2\_miss$. All the segments add up to 100%. Figure 1(b) shows the relative difference between max_L2_miss and min_L2_miss, that is $\frac{max\_L2\_miss - min\_L2\_miss}{min\_L2\_miss}$. The figures clearly show that program segments in real-world programs not only have a widely distributed spectrum of behaviors, but also possess a large difference between behavior extremes.

The behavior divergence of code segment provides new opportunities for code optimization. Intuitively, if a code segment that shows behavior divergence could be optimized according to its different behaviors under different contexts, the overall performance would improve. However, that opportunity is unexploited in code optimization research. The most relevant code optimization technique that tries to take advantage of the behavior change in code segment is probably hotspot optimization [3], which applies more aggressive code transformations when a code segment is determined to contribute a relative large portion of overall execution time or the segment is executed frequently. However, the selection of optimization is not differentiated based on the possibly different behavior of the target segment. Trace scheduling [4] and trace-based optimization [5] are also related. This type of code optimization find out the program segments that are

978-1-4244-5736-6/09/$26.00 ©2009 IEEE



(a) Distribution of behavior divergence

(b) Maximal divergence between behaviors

Figure 1: The behavior divergence of benchmarks.

consequentially executed and enable the usage of optimizations designed for straight-line code for the traces. However, the selection of trace and the optimization are still based on execution frequency of program segments, and does not consider the difference in the behavior of those segments as a relevant factor. This paper is a first effort to systematically detect behavior divergence of code segment under different contexts and search for compiler transformation settings that perform best for the different contexts. A novel algorithm to detect hot traces is introduced and the behavior divergences of these traces are detected. Our approach to detect behavior divergences can also be applied onto pathes determined by path profiling [6] methods. We leave this application and optimizations to a future work. Program slicing method [7] is used for debugging and measuring functional cohesion. It only considers and processes statements which are depending on interested data elements. Our approach in this paper applies the modifications to the whole basic blocks and not to a subset of the basic block statements.

The behavior divergence can be exploited at different code granularity, from procedure down to basic block. In this paper, we target at the behavior of basic block sequences in a procedure. The main contribution of this paper is a systematic solution to profile program so that the behavior of basic block sequence can be determined, to select the basic block sequences that are most likely to benefit from context-aware optimization, and to empirically search for the best compiler transformation settings for a basic block sequence under different contexts. The paper presents all the three steps, including a profiler for basic block behavior that is built on top of the Low Level Virtual Machine (LLVM) compiler infrastructure [8] in Section II-A, a behavior analyzer in Section II-B and an empirical search engine for transformation setting in Section III. We apply our approach to a number of SPEC2000 [1] and SPEC2006 [2] benchmarks and achieve up to 17% speedup.

II. Basic Block Sequence (BBS) Tool

This section describes how we profile a program to gather context information and how the profile information is analyzed to determine the basic block sequences that can benefit from context aware optimization. The whole process is done in two phases: runtime profiling and off-line analyzing.

A. Profiling Tool

The profiling tool is developed using Low Level Virtual Machine (LLVM) Compiler Infrastructure [8]. Code is instrumented at the basic block level. The tool numbers each function and each basic block in a program uniquely. The profiling code is inserted into program at the intermediate representation level. The compilation process for an application with multiple source code files is done in multiple stages. The compilation stages is shown in figure 2. This tool makes use of the LLVM tools *llvm-gcc*, *llvm-link*, *opt* and *llc* [8] and *GCC* compiler [9].

The instrumentation code performs two conceptually simple tasks, namely to record the sequence of basic blocks that is executed and to book-keep the hardware counter readings at the basic block level. We implement the two tasks in a lightweight way so that the instrumentation code has minimal interference with the behavior of the target program. The profiling information can be regarded as database and is used by the analysis tool to gather runtime properties.

B. Analysis Tool

The goal of the Basic Block Sequence (BBS) analysis tool is to find basic block sequences that show large variance in their runtime behavior and are executed relatively frequently. The requirement for large variance in runtime behavior is a natural conclusion from the definition of context-aware

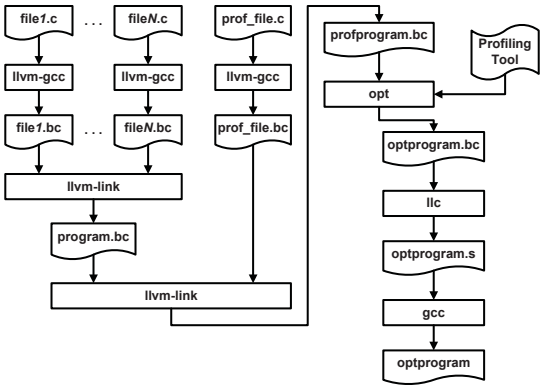


Figure 2: The instrumentation and compilation of code.

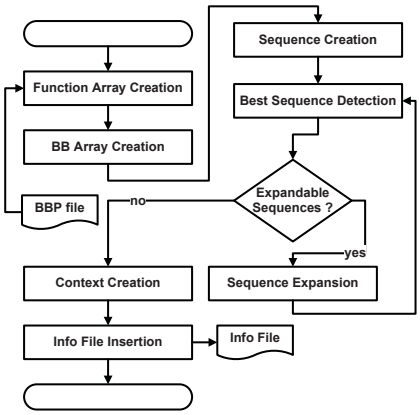


Figure 3: Flow graph of the BBS tool. This flow graph is processed separately for each function in BBP file.

optimization. If a program segment performs identically among its multiple executions, there is no reason to optimize the segment in different ways. It is necessary to require a relatively high execution frequency for a target program segment of context aware optimization because the context-aware optimization has setup overhead which must be amortized over multiple executions of the target program segment.

We analyze the profiling file and determine basic block sequences for each function of an application. Inter-procedural analysis of basic block sequence behavior is not yet supported at this stage. A length of a sequence is the number of adjacent basic blocks in that sequence. An occurrence of a sequence is the number of times that sequence is executed in the profiling file. In this paper the profiling information of a program is stored in a file called basic block profiling file (BBP file). The file contains the basic blocks executed in each function and their hardware counter readings.

The workflow of the analysis algorithm is illustrated in Figure 3. The whole process is applied to each executed function in the BBP file. The data structures and variables of each step of the algorithm are shown in figure 4. The first step of the analysis is the *Function Array Creation*

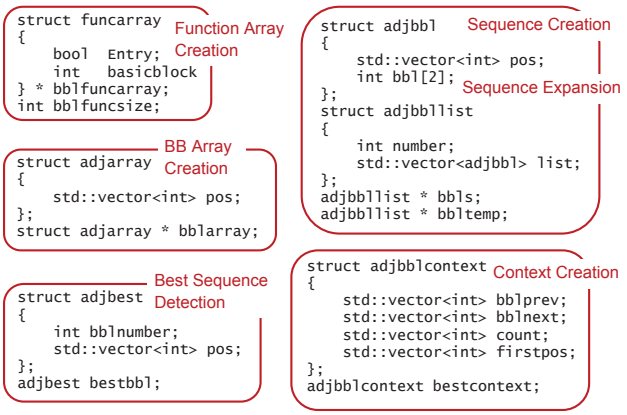


Figure 4: Structures and variables for the steps of the BBS tool.

step, which creates an array for each function and copies information about all basic blocks of the function to analyze into that array (*Function array*). This step is needed to make the processing of further steps efficient. The Function array is shown as variable *bblfuncarray* and has the size *bblfuncsize* in figure 4.

Next step in algorithm is *BB Array Creation* step. The size of the array *bblarray* in figure 4 is the number of all basic blocks present in the target program. Each array element contains vector of the positions of the basic blocks at the Function array. The order of the array element in the array *bblarray* is according to the basic block numbering, For example, the array element at the index 0 of *bblarray* contains the position vector of the basic block with number 1. The next array element contains position vector of basic block with the number 2. The array in this step is used to determine the first elements of the basic block sequences created at further steps. The sum of the position vector sizes equals the size of the Function array. This array increases the memory usage of the application and will be freed after the next step of the application.

The next *Sequence Creation* step of the algorithm goes through the *bblarray* and basic block sequences of length two is created. The result of the Sequence Creation step is all basic block sequences of length two that have been executed at least twice during the execution of the whole program. For instance, for basic block sequences starting with basic block one the function of this step goes through the position vector of this basic block and determines the basic block sequences. After this step the variable *bbls* contains list of basic block sequences of length two. This variable is modified in further steps. The structure member *number* shows the length of the basic block sequences in the member *list*. Each list element contains the structure members *pos* and *bbl*. The position vector *pos* contains the positions of the basic block sequence in the Function array. In the following steps the basic block sequences are extended with new basic blocks and the position vector is used to go through the sequence

```
Create new adjbbllist and assign to bbls
bbls->number ← 2
for each basic block bb in the Module do
  for each Function Array position PosVec1 in BB array of basic block bb do
    if basic block in position PosVec1 + 1 of function array is not Entry block then
      Create new Sequence newbbl and push in bbls->list
      Push the position PosVec1 into the vector of the sequence
      for each position PosVec2 in BB array located after PosVec1 do
        if basic block in position PosVec2 + 1 is not Entry block and basic block in position PosVec1 + 1 = basic block in position PosVec2 + 1 then
          Push the position PosVec2 into the vector of the sequence
          Delete PosVec2 from BB array
        end if
      end for
    end if
  end for
end for
FREE BB array
DELETE Sequences with Occurrence 1 from bbls->list
Function return value is the number of available sequences
```

Figure 5: Pseudo code of *Sequence Creation* step.

```
Move bbls to bbltemp, Create new adjbbllist and assign to bbls
bbls->number ← Sequence Length and I ← Sequence Length-1
for each sequence Sequence in previous list bbltemp->list do
  for each Function Array position PosVec1 in the position array of Sequence do
    if basic block in position PosVec1+I of function array is not Entry block then
      Create new Sequence newbbl and push in bbls->list
      Push the position PosVec1 into the vector of the sequence
      for each position PosVec2 in position array of Sequence located after PosVec1 do
        if basic block in position PosVec2 + I is not Entry block and basic block in position PosVec1 + I = basic block in position PosVec2 + I then
          Push the position PosVec2 into the vector of the sequence
          Delete PosVec2 from position array of Sequence
        end if
      end for
    end if
  end for
end for
FREE bbltemp
DELETE Sequences with Occurrence 1 from bbls->list
Function return value is the number of available sequences
```

Figure 6: Pseudo code of *Sequence Expansion* step.

elements efficiently. Without this vector, a search of the sequence elements in the Function array is needed. This vector eliminated this search process. The variable *bblarray* is freed after this step, because it is not needed in further steps. Pseudo code of this step is shown in figure 5.

The *Sequence Expansion* step of the algorithm is an iterative process. The step iterates until there is no expandable basic block sequence, i.e. with execution frequency at least two, left. This step uses the same structures and variables as in previous Sequence Creation step. In this step first the variable *bbls* is moved to a temporary variable *bbltemp* and recreated. The temporary variable is freed at the end of this step. The length of the basic block sequences is increased by one at each iteration. The function of this step goes through the position vector of each basic block sequence and expands it with new basic blocks. The old basic block sequence with the new inserted one has to be present at least two times for being included into the newly created basic block sequence list. The positions of the old basic block sequences with same extended basic block is copied to the newly created sequence. The newly created basic block sequences which only differ at the last basic block will be treated independently in the next iteration as they are two different basic block sequences. At the end of this step in each iteration the basic block sequences of current iteration are available and the basic block sequences of previous iterations are freed. The reason for deleting basic block sequences is to prevent high memory usage which slows down the algorithm and can cause memory overflow. Pseudo code in figure 6 shows this process. Another step to determine best basic block sequence follows this step at each iteration. The iteration stops if the next *Best Sequence Detection* determines that there are not any better basic block sequences.

The Sequence Creation and Sequence Expansion steps are always followed by *Best Sequence Detection* step. This step also iterates with the Sequence Expansion step. This step determines best basic block sequence. This step is required to be iterated because the variable *bbls* is redefined at each iteration. The criteria for best sequence is that the number of occurrence of the basic block sequence (*OoS*) times the length of the sequence (*LoS*) is maximum. The product of this multiplication will further be referred as *sequence value* (*SV*) of the basic block sequence.

$$SV = OoS * LoS \ .$$

This step goes through the basic block sequence list of the preceding steps and determines the best basic block sequence which has maximum sequence value. The sequence value of the best basic block sequence is called *best sequence value*. The variable for this step is also shown in figure 4.

After the iteration through the steps Sequence Expansion and Best Sequence Detection comes to an end, the algorithm has determined best basic block sequence for the function. The next step in the algorithm is *Context Creation*. In this step the contexts of the best basic block sequence are determined. If the basic block sequence shows large variance of behavior in hardware counter contexts, we will correlate the contexts with the preceding basic block of the selected basic block sequence. A *Context(a)* of basic block sequence *a* is defined as the occurrence of unique combinations of preceding basic block and hardware counter contexts. Ideally, the parameter that is used to distinguish and define different hardware counter contexts should be determined based on the benefit analysis, that is, the division of hardware

counter context should maximize performance improvement. In this paper, we just empirically set the parameter. The value can be different for different programs.

The best basic block sequence and its context information is appended into an *Info File* in the step *Info File Insertion*. This step concludes the analysis algorithm.

In summary, this section shows the process of creating best basic block sequences for each function in the program. These sequences can be understood as *hot spots* with widely different behaviors during different executions of the sequences. The resulting *Info File* contains the best basic block sequences and their context information.

III. Context Aware Code Transformation and Optimization

This section describes the technique that we develop to transform and optimize a program utilizing the context information found by the analysis tool described in the previous section. The basic idea of our context-aware code transformation and optimization is generating multiple versions of program segments for each context and empirically searching the best compiler transformation combination for each version. Like the profiling tool, the code transformation tool is also built using the LLVM framework.

The context aware code transformation and optimization process has three stages: *generate sequence functions for contexts*, *find customized optimization settings for sequence functions*, and *inline optimized sequence functions*. The details of these stages are described in following three subsubsections.

A. Generate Code Versions for Contexts

The basic block sequences and their context information that are found by the analysis tool are first grouped based on different sequence lengths. The basic block sequences hat have only one context are not processed in further steps, because it means that those sequences do not show big variance in behavior when they are executed multiple times and they will not benefit from further customized optimization.

The generation of a version of a basic block sequence for each of its contexts can be achieved simply by extracting the basic block and duplicating it the same number of times as the number of contexts. There is one additional step in the approach used in this paper. That is, instead of simply copying the basic block sequence, we first wrap each copy into a new function. This step is necessary only because we want to use many optimizations existing in the LLVM infrastructure. The LLVM compiler infrastructure contains optimization tools which can be applied onto the LLVM intermediate representations of programs. The number of optimization tools at basic block level is few. The majority of the optimization tools are at loop or function level. In these optimization tools the loop level or basic block level ones

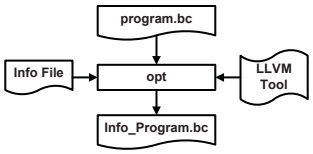


Figure 7: Application of the Info File onto the benchmark module.

can also be applied onto the functions, but the function level optimization tools cannot be applied onto the loops or basic blocks. The desire is to be able to use as many optimizations as possible. In order to have the ability of applying more optimization on the basic block sequences, the extraction of these sequences in separate functions is chosen. These functions will be referred as *Sequence Functions*. Following steps show how the extraction is performed separately for each best basic block sequence present in the benchmark. In our analysis each function contains one sequence with multiple contexts, therefore separate sequence functions for each context case is created. The figure 7 shows the input and output of the LLVM optimizer program *opt* in this tool. The input to the optimizer is the program module which has been shown in figure 2. The tool described in this section modifies it in multiple steps based on the data in *Info File* and outputs the new module with *Sequence Functions*. The steps of the LLVM tool performing this extraction is shown below.

The first step in the generation of sequence functions is the duplication of each instruction in those basic blocks for each context. The different contexts will have their own sequence. The basic blocks which are executed more than one time in each sequence is only duplicated once. For example for a sequence, like (5 10 11 10 11 3), the basic blocks 10 and 11 is only duplicated one time, so the new functions will only contain the basic blocks 5, 10, 11 and 3. After this step, there are sequence functions for each context. There will be *number-of-context* times sequence functions for each function in the analysis results.

In the second step the dependencies in these duplicated basic blocks are updated. For example, a instruction in basic block 10 depends on an instruction in basic block 5. After this step the instruction in duplicated basic block 10 will depend on instruction in duplicated basic block 5. This step also replaces the branches within the sequence basic blocks with the duplicated ones.

The dependencies of the instructions in the sequence basic blocks with the instructions outside sequence basic blocks is solved in step three using function arguments. The used function arguments are called by reference arguments and any changes made in the sequence functions can still be obtained after function execution.

Step four deals with branches to the basic blocks outside the sequence. These branches are resolved by using different return values of the sequence functions. For each branch to

```
-block-placement  -loop-reduce  -condprop  -adce
-loop-index-split -loop-rotate  -constprop -cee
-loop-unswitch    -loop-unroll  -gvn       -die
-lower-packed     -loopsimplify -gvnpre    -dse
-predsimplify     -lowerswitch  -indvars   -licm
-tailcallelim     -reassociate  -mem2reg   -sccp
-tailduplicate    -scalarrepl   -reg2mem
```

Figure 8: Optimization flags applicable on LLVM intermediate representation and used in optimization algorithm.

a basic block not in the sequence the function has different return value.

In step five context times new basic blocks inside original benchmark functions are created. Each new basic block contains one call instruction to their respective sequence function and a switch instruction. The switch instructions contain branches to different basic blocks based on the return value of the functions. The predecessor basic block in each context contains a branch to the first basic block of the sequence. These branches are replaced with branches to their respective new basic block, which calls that particular context's sequence function.

After the creation of the sequence functions, different optimization algorithms can be applied onto the sequence functions and optimize the best basic block sequences.

B. Finding Customized Optimizations

The step following the generation of sequence functions is the customization of compiler optimizations for each sequence function. Ideally, the selection of compiler optimizations can be analytically determined from the context. In this paper, we use an existing algorithm that empirically search for the best compiler optimization setting for programs. The algorithm, *Iterative Elimination (IE)*, is proposed in [10]. That algorithm is implemented and modified slightly. In the original paper the algorithm starts by turning on all optimizations and uses it as baseline. New optimization combination is found by turning off optimizations. In this paper the algorithm inverts this processing and starts by turning off all optimizations and customized optimization is found by turning on optimizations.

The LLVM compiler infrastructure contains a number of optimizations for the LLVM intermediate representation. Our implementation uses optimizations at function, loop and basic block level. Interprocedural optimizations is not used because the goal is to find customized optimizations for each sequence function uniquely. The optimizations used for the algorithm is shown in figure 8. The compilation of the benchmarks with different optimization flags is demonstrated in figure 9. The compilation process is executed for each optimization combination and a performance value is obtained. The performance of the benchmarks is measured using PAPI performance counter library [11].

The implemented algorithm gets baseline information of the particular function and at each iteration it finds combination which includes combination of previous interval plus an additional optimization that deliver better performance than the current iteration. For example, the first iteration will find single optimization which has better performance than other optimizations. In the second iteration, it finds combination of the single optimization with one more optimization which results to best performance. After second interval, two optimizations are turned on. This process continues until there is no performance improvement by adding further optimizations into the combination. This combination is referred in this text as *Customized Optimizations* for that particular function. This process is repeated for each interested function which are in our case the sequence functions and the functions calling them. The performance counter *PAPI_TOT_CYC* which measures total cycles, is used for determining the performance value. The performance value for the whole benchmark is measured, so the customized optimizations are best combination of that function that improves benchmark performance.

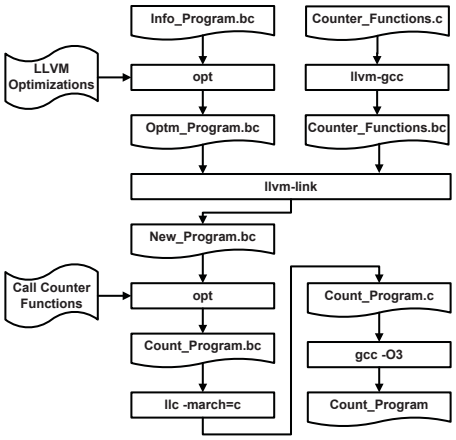


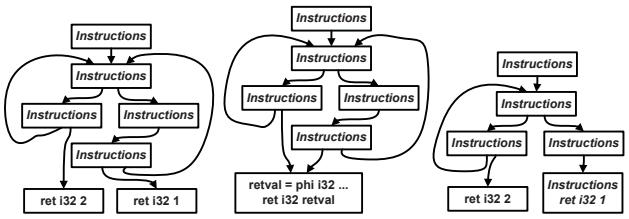
Figure 9: Finding Customized Optimizations for each function. The optimizations for the current processed function is applied at the box *LLVM Optimizations*

This algorithm is applied onto sequence functions and customized optimizations for each sequence function is determined.

C. Inline Sequence Functions

Sequence functions and the basic blocks calling these functions add overhead to the original benchmark functions. Therefore the optimized sequence functions are inlined to their original benchmarks. Application of the customized optimizations changes the structure of the sequence functions from its non-optimized structure, so the changed structure has to be considered in inlining those functions.

The form of unoptimized sequence functions can be seen in figure 10(a). In inlining of this form the branches to the basic blocks containing return instructions is replaced to the branches to the basic blocks in original functions. In the original benchmark functions there are branches in



(a) Unoptimized Form (b) Unified Ret. Block (c) Motion of Return

Figure 10: Different Forms of Sequence Functions

predecessor basic blocks to the basic blocks containing the call instructions to the sequence functions. These branches are replaced with branches to the first basic blocks in the sequence functions.

After the application of customized optimization the function's structure changes. One example is shown in figure 10(b). In this case the values in phi instruction has to be considered. First the basic blocks in original function for those values is determined and later the branches to this return block is replaced with branches to the basic blocks in original function. The unified return block can be deleted thereafter.

In another optimized form the return instruction is moved from return basic block to the blocks which contain further instructions. This kind of example is shown in figure 10(c). In this case the basic block containing the moved return instruction is kept at the inlining of sequence function, but the return instruction is replaced with a branch to the basic block in original function.

The optimized sequence functions whose return value is determined by a memory load could not be inlined in this tool, because the value at the memory location cannot easily be determined at compile time. This condition, however, has not occurred in the SPEC benchmarks used in this paper.

IV. Experimental Results

In this section we present the results of applying our context-aware code transformation and optimization to nine SPEC CPU2000 [1] and five SPEC CPU2006 [2] benchmarks of the Standard Performance Evaluation Corporation (SPEC). All the benchmarks we use are in *C* language. The benchmarks programmed in *Fortran* are not used because the LLVM compiler infrastructure does not contain a *Fortran* frontend. As it is shown in the figures the CPU2000 benchmarks are *164.gzip*, *175.vpr*, *181.mcf*, *255.vortex*, *256.bzip2*, *300.twolf*, *179.art*, *183.equake* and *188.ammp*. The CPU2006 benchmarks are *429.mcf*, *456.hmmer*, *462.libquantum*, *470.lbm* and *458.sjeng*.

The benchmarks are tested on a multicore platform. The information of the machine is shown below:

```
Processor: Intel(R) Xeon(R) CPU 8 X 2.00GHz Cores
L1 cache size: 6,144KB      RAM: 9,015MB
Operating System: 64 Bit Linux 2.6.23
```

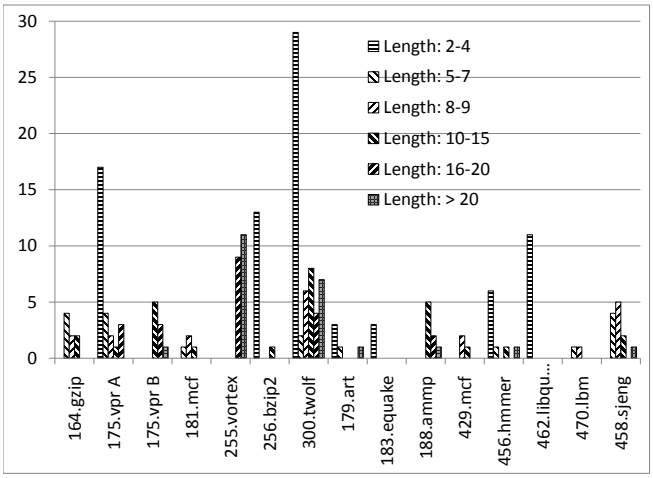


Figure 11: The number of sequences for different length ranges.

A. Result of the Analysis Tool

We first show the result of the analysis tool, that is, how many basic block sequences are found to have multiple contexts and how long are the sequences. Functions in the analysis results where the sequences have only one context are excluded, because those sequences will not be further processed. All the sequences shown in the figure 11 contain more than one contexts. It is also not shown the functions whose processing do not have effect on the overall benchmark performance. As shown in the figure, all the benchmarks have basic block sequences that have moderate length and multiple contexts.

B. Benchmark Performance

In this section we present the performance results. We call the original benchmark version without modification as *Original Version* and the second version created after using our context-aware code transformation and optimization technique is defined as the *Inlined Version*. The performance of the inlined version and the original version is compared below.

The multicore features of the test platform is not used in the experiments. Only one core is used for testing the benchmarks. PAPI performance counter library is utilized to measure the performance [11]. The process of applying the optimizations is shown in figure 9. As it is shown in chapter III-B, the optimizations are applied in the box *LLVM Optimizations* of that figure.

In the original version standard LLVM optimization *-std-compile-opts* for the original benchmark functions is applied. The performance for the optimized version is measured and is used for comparison.

In the inlined version first customized optimizations are found and applied to the sequence functions. At the second step the sequence functions are inlined to the original benchmark functions. Third step is determination and application

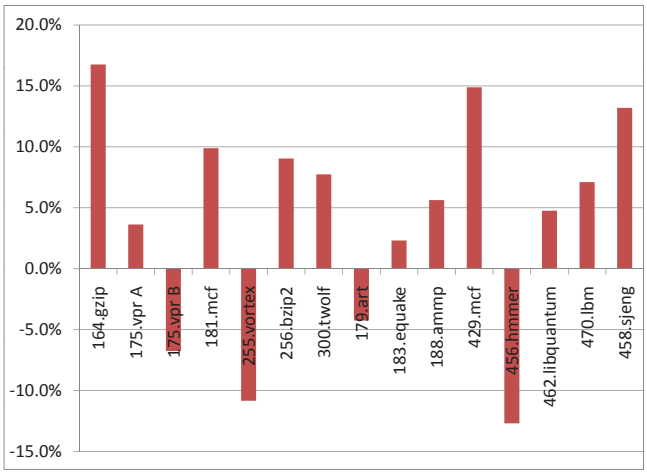


Figure 12: The Relative Performance of the Inlined Versions with applied customized optimizations compared to Original Versions with standard LLVM optimization.

of customized optimizations to the original benchmark functions.

The performance counter *PAPI_TOT_CYC* is used to determine performance information. A relative performance increase of the inlined version compared to the original version is shown in figure 12. In this figure the customized optimizations are applied to the inlined version of the benchmarks and their performance measured. The standard optimization is applied onto the original version. The Relative Performance Increase of the inlined version compared to the original version is shown in the figure. The formula used for the relative performance increase *RI* is

$$RI = \frac{SOOV}{COIV} - 1 \ ,$$

where *SOOV* is the performance of the standard optimized original version and *COIV* is the performance of the customized optimized inlined version.

The test result in figure 12 shows that optimizations based on the proposed technique have performance improvement in 10 of the 14 tested SPEC benchmarks. The benchmarks with negative performance effect are *255.vortex*, *183.equake* and *456.hmmer*. The functions in these benchmarks have much longer basic block sequences compared to the benchmarks with positive performance. They have more branch instructions. Another feature of theses sequences are that the performance difference of different contexts have lower variance. The sequence of one context usually dominates the other context cases. Hence the intrinsic overhead of context aware optimization is not amortized well.

Another reason for the negative performance of the benchmark *255.vortex* is that it was not possible to use the GCC flag *-O3* in this benchmark, because it lead to obsolete output. The optimization at the last level of our compilation was modified for this benchmark. The optimization flag used was *-O1*. Therefore, global, interprocedural optimizations could not be applied onto the inlined version. The standard optimization of the LLVM *-std-compile-opts* contains interprocedural optimization, so these optimizations were applied onto the original version. In order to conceive the effect of these optimization, another step of optimization is included into the inlined version. After the customized optimizations are applied onto the duplicated and original functions, the standard optimization of the LLVM is applied onto the whole benchmark. The result show a 1.87% performance improvement compared to the original version.

V. Conclusion

In this paper, we proposed a novel feedback-driven program optimization technique that customizes the optimization of a program according to its behavior divergence under different contexts. Our technique determines frequently executed basic block traces that have different runtime behaviors, and optimizes the traces individually for different contexts. We applied our method onto 14 SPEC [1] [2] benchmarks. The contexts are optimized using an empirical optimization algorithm. The preliminary result we gathered in using our method show that we get performance increase in 10 of the 14 benchmarks. Our future work will include more precise benefit analysis for our optimization technique, the detection of behavior divergence at different granulities, and an analytical model to select optimizations based on different contexts.

References

[1] *SPEC CPU2000 V1.3*. http://www.specbench.org/cpu2000
[2] *SPEC CPU2006*. http://www.spec.org/cpu2006
[3] J. Lau, M. Arnold, M. Hind, and B. Calder, "Online performance auditing: using hot optimizations without getting burned," *SIGPLAN Not.*, vol. 41, no. 6, pp. 239–251, 2006.
[4] J. Fisher, "Trace scheduling: A technique for global microcode compaction," *Computers, IEEE Transactions on*, vol. C-30, no. 7, pp. 478–490, July 1981.
[5] P. P. Chang and W. W. Hwu, "Trace selection for compiling large c application programs to microcode," in *MICRO 21: Proceedings of the 21st annual workshop on Microprogramming and microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 21–29.
[6] T. Ball and J. R. Larus, "Efficient path profiling," in *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 46–57.
[7] M. Weiser, "Program slicing," in *ICSE '81: Proceedings of the 5th international conference on Software engineering*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.
[8] *The LLVM Compiler Infrastructure*. http://llvm.org/
[9] *GCC, the GNU Compiler Collection*. http://gcc.gnu.org/
[10] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 319–332.
[11] *Performance Application Programming Interface (PAPI)*. http://icl.cs.utk.edu/papi/