Improving the Productivity of Compiler Code Quality Analysis

Hongbo Rong, Andy Ayers, David Gillies

Microsoft Corporation

{hongbor, andya, dgillies}@microsoft.com

Microsoft Research Technical Report MSR-TR-2009-18

February 22, 2009

Abstract

Producing quality code is one of the most important goals of an optimizing compiler. Analyzing code quality is therefore an essential activity in compiler engineering. By motivating new optimizations and diagnosing regressions, it takes a bottleneck position in the process. However, it has been highly empirical, and dependent on architectures and tools. This makes it a difficult and time-consuming task, and its productivity is unpredictable and usually low.

This paper proposes two novel approaches for code quality analysis. The first approach focuses on the key scenario in compiler construction, the computation-intensive benchmarks. We observed that the workload upon the processor dominates the execution time of such benchmarks. Therefore, we use the compiler to parse the workload. By doing this, the compiler applies its static analysis power to identify its own code quality issues and potentials. We have implemented a software system for this approach and integrated it with our daily testing infrastructure. It automates the code quality analysis for Spec benchmarks, and provides developers relevant information in reasonable time.

The second approach addresses code quality regressions of operating system benchmarks. We take advantage of the built-in instrumentation of the operating system to collect traces of events, and then construct a tree out of the trace. Analyzing regression is thus simplified as tree comparison. With the tree structure, this approach divides and conquers the usually huge amount of data, and effectively localizes the focus to a few leaves of the trees. It has been implemented, and addressed several difficult issues that led to visible code quality improvement.

This paper also summarizes our experiences in bringing up the code quality of a product compiler framework, establishes a few simple guidelines to achieve code quality objectives with minimized efforts, and empirically compares various analysis approaches.

0.1 Introduction

Producing quality code is one of the most important goals of an optimizing compiler. Analyzing code quality is therefore an essential activity in the compiler development. Given the binary that the compiler generated for a benchmark, *Code Quality (CQ)* measures how good the binary is, in terms of the execution time, code size, energy consumed, etc. In this paper, we follow the traditional practice and measure code quality by the execution time of the binary. However, the techniques proposed should also be applicable under other contexts.

CQ analysis serves two purposes in compiler engineering: exposing optimization opportunities and addressing regressions timely. For simplicity, we would refer to them as *potential analysis* and *regression analysis*, respectively. In regression analysis, there is a *test* and a *baseline* binary for the same benchmark, and the test binary runs slower than the baseline.

Engineering code quality is an iterative process. See Fig. 1. It is clear that CQ analysis is at a bottleneck position. The productivity of CQ analysis directly impacts the productivity of the whole progress. This is especially true at the final stage when the compiler is striving toward its performance goals.

Unfortunately, CQ analysis is difficult and time-consuming. Its productivity is unpredictable and usually low, because of its empirical nature: how fast a CQ issue can be detected is dependent on human experiences, the specific issues, architectures, and tools. In a sense, it has been something of an art form. Although there have been profiling tools with friendly graphic interfaces [15, 1], they help mainly when the issues are within the "hot spots" of the binary. It is still a painful process in exploring the binary for subtle issues if they are not from a hot spot. Effectively using them needs expertise in hardware, and skills in drilling down CQ issues. The analysis process is interactive between human and computer, not automatic.

For the above reasons, in a product environment, there can be a wide gap between the desirable and the practical productivity of CQ analysis. For instance, in our development of a product compiler framework, there are on average 3 to 4 checkins each day. Each checkin causes small CQ improvement and regressions, mingled with hardware noise, across hundreds of benchmarks. These benchmarks include standard performance suites, and various internal tests in both user and kernel mode. With the spectrum of the benchmarks, it is not realistic to expect every developer to have enough experience and time to address every CQ change. Even for a professional CQ analyst, our experience is that analyzing a benchmark takes from half an hour to several days.

To fill the gap, it is a pressing need to improve the productivity of CQ analysis. To achieve this goal, we must automate the analysis whenever possible, reduce human involvement, and lower the requirement on experiences.

The contributions of this paper are as follows:

1. First, we propose *Workload parsing* to automatically analyze the code quality of computation-intensive benchmarks, which is usually the key scenario in compiler construction.

   A fundamental observation is that for a computation-intensive benchmark, *it is the workload upon the processor that determines the overall execution time, and*

the workload is characterized by a few characteristic events of the benchmark. Therefore, we use the compiler to parse the workload and collect statistics for these characteristic events. By doing this, the compiler applies its static analysis power to identify *its own* CQ issues and potentials.

For regression analysis, the approach compares the characteristic events between the test and the baseline binary of the benchmark, and highlights the biggest differences to developers. For potential analysis, it exposes possible optimization opportunities by detecting the hottest execution paths of the benchmark via a set of *path caches*, exposing partially redundant and dead instructions via interprocedural backward slicing along these paths, and identifying the 95% instruction footprint, which is the set of static instructions that account for 95% of the dynamic instructions.

To the best of our knowledge, this is the first approach that automates CQ analysis, and the first one that uses compiler to diagnose its own problems.

This approach combines the strength of dynamic profiling, dynamic analysis, and static analysis. It collects only basic statistics during dynamic profiling and analysis, identifies hot paths with path caches instead of whole program tracing and online compression, and throws away the instruction address trace on the fly. It leaves most of the work to the compiler to do static analysis. This is the key feature that distinguishes this approach from other profiling and analysis tools [9, 18]. This is also the key design tradeoff that enables us to handle most Spec benchmarks with more realistic dataset (train input) in reasonable time, compared with whole program paths that mainly used trivial dataset (test input) [8].

2. Second, we propose *Event tree comparison* to partially automate regression analysis of operating system benchmarks, which is a key scenario when the compiler intends to build an operating system.

   The profiles of such benchmarks are radically different from those of computation-intensive benchmarks. They tend to be flat without global hot spots, involves OS kernel, multiple processes, and workload on disks. This scenario tends to be more difficult to analyze.

   We take advantage of the built-in instrumentation of the operating system to collect traces of the OS events that happen when a benchmark runs. We observed that *a stable benchmark tends to have some deterministic events in a trace: the number and order of such an event stay the same for any execution of such a benchmark*. By partitioning a trace with these events, the trace is transformed into a tree. Analyzing regression is thus simplified as comparing the trees corresponding to the traces of the baseline and the test binary.

   With the tree structure, this approach divides and conquers the usually huge amount of data in the traces, and effectively localizes the focus to a few leaves of the trees, where some local hot spots can be detected from stack-walking information. Although this approach needs human guidance to provide the deterministic events beforehand and analyze the local hot functions afterward, since

a major difficulty in analyzing this kind of benchmarks is the lack of global hot spots, being able to locate local hot spots removes a major obstacle.

3. Third, a software system, with workload parsing in its heart, has been implemented and integrated into our daily testing infrastructure. It analyzes regressions automatically when they happen, and provides developers with timely feedback. It also performs potential analysis upon request. The kernel of the software system, workload parsing, is also made an independent tool, so that a developer can conveniently run it for analyzing code quality on his or her personal computers. We are able to handle the more realistic dataset (train input) rather than the trivial dataset(test input) for most Spec2000 and Spec2006 benchmarks, and the turnaround time is reasonable. The system and the tool are proved promising and useful in the product environment.

4. Forth, event tree comparison has also been implemented. With it, we have addressed several difficult issues that led to visible code quality improvement in the entire system.

5. Finally, we summarize our experiences in bringing up the code quality of the product compiler framework, establish a few useful guidelines on how to achieve CQ objectives with the minimal efforts, and empirically compare the strength and weakness of various analysis approaches.

To avoid any confusion, we should point out the differences between CQ and *throughput* analysis, and that between CQ regression and *linear regression modeling*. CQ analysis considers the execution time of the compiler-generated code for a benchmark, while throughput analysis considers the execution time of the compiler itself. In this paper, we focus on CQ analysis only.

In later sections, we will use a statistical method, called linear regression modeling, to identify some statistical trends. It is clear that the "regression" referred in this method is a completely different concept from CQ regression. In this paper, with "regression", we refer to CQ regression by default.

The paper is organized in this way: Section 0.2 motivates and explains workload parsing, and the software system implemented. Section 0.3 describes event tree comparison. Section 0.4 evaluates the productivity of the approaches in our product environment. Then we introduce the guidelines of CQ analysis, compare various analysis techniques, and arrive at the conclusion.

0.2 Workload Parsing

As Fig. 1 shows, in the early steps of engineering compiler code quality, key scenarios need to be identified, and their workloads need to be characterized. Computation-intensive benchmarks like Spec have been the key scenarios in compiler engineering. Such benchmarks more faithfully reflect the compiler's effect upon the execution time, while the effect of other system components like disks, network, and the operating system, is negligible. Our compiler framework also builds an operating system, and

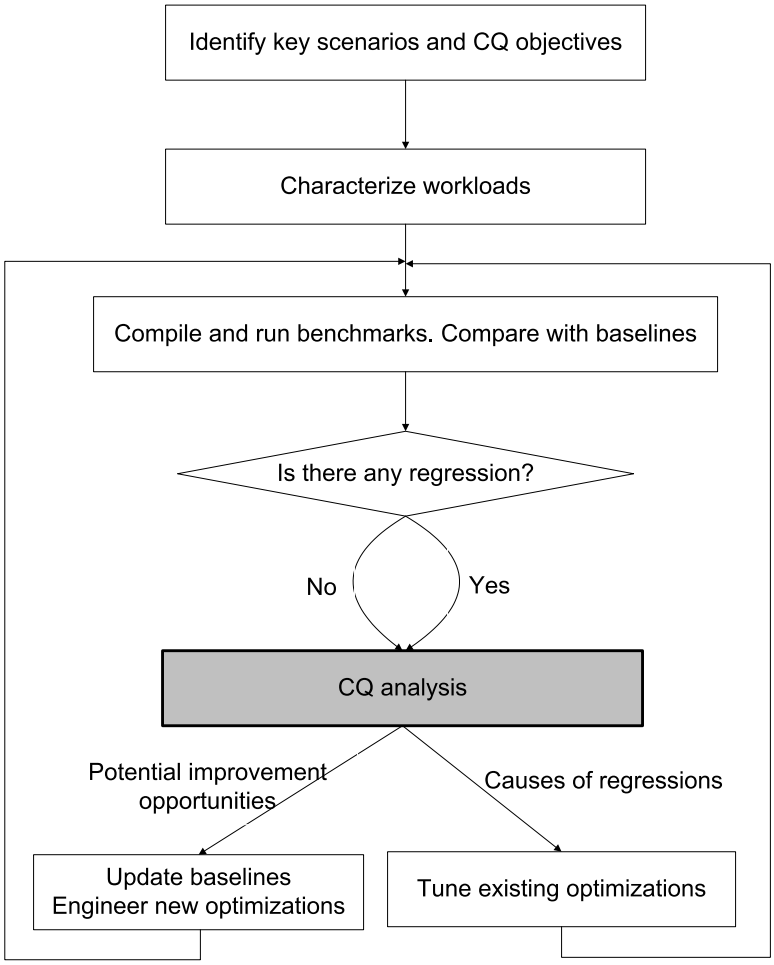
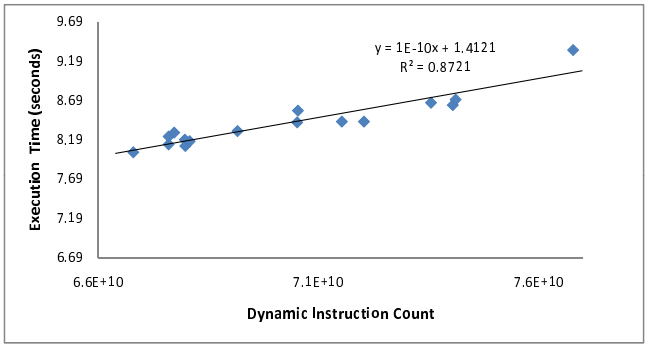


Figure 1: Compiler code quality engineering.

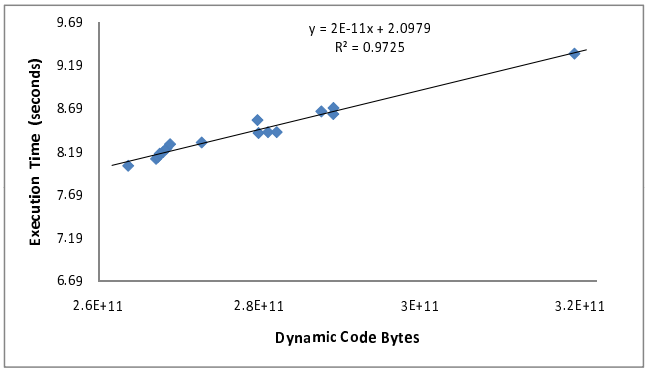
thus some OS benchmarks are also our important scenario. We will characterize them later in Section 0.3.

Below we characterize the workload of computation-intensive benchmarks. Statistics shows that such a benchmark has some characteristic events whose total numbers determine the execution time of the benchmark. These events tend to highly correlate with each other. Among them there are two key events: dynamic instruction count and dynamic code bytes. They are the workload upon the processor.

This observation has motivated the workload parsing approach. We describe the approach in detail, and introduce how it is integrated into our daily testing infrastructure.



(a) Dynamic instruction count vs. execution time



(b) Dynamic code bytes vs. execution time

Figure 2: The linear relationships between dynamic instruction count/code bytes and execution time of Winsat (Windows Assessment Tool). 18 builds are used in the experiments. They are run on an Intel Core2 Duo machine to get the execution time. They are also simulated with Nirvana simulation framework [3] to get exact instruction counts and code bytes. Repetition of the experiments on an Amd64 Dual Core machine leads to similar results.

0.2.1 Characterize the Workload of Computation-intensive Benchmarks

Fig. 2 shows a discovery of the linear relationships between dynamic instruction count/code bytes and the execution time for a computation-intensive benchmark [1]. The linear rela-

[1]In the linear relationships in Fig. 2, the annotation R^2 is Pearson's coefficient of regression. R^2=0.8721 (0.9725) means that 87.21% (97.25%) of the change in the execution time is caused by the change in the dynamic instruction count (code bytes).

Benchmarks	Characteristic events
Winsat	dynamic instruction count, dynamic code bytes
spec2000 164.gzip	retired instructions, retired branches, instruction cache fetches, instruction fetch stall cycles, retired mispredicted branches, dispatch stall cycles
spec2006 400.perlbench	retired instructions, retired branches, retired mispredicted branches, instruction cache fetches/misses/refills from L2 cache, ITLB misses at both L1 and L2, instruction fetch stall cycles, dispatch stall cycles, data cache accesses, L2 cache request
spec2006 401.bzip2	retired instructions, instruction cache fetches
spec2006 403.gcc	retired instructions, retired branches, data cache accesses
spec2006 458.sjeng	retired instructions, retired branches, data cache accesses

Table 1: Characteristic events of some computation-intensive benchmarks

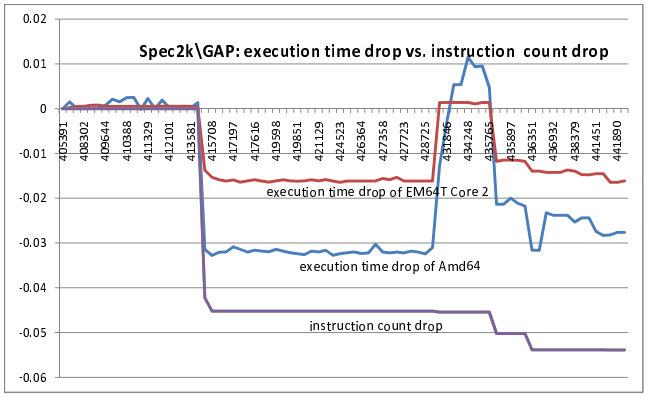


Figure 3: Statistics of Spec2000 254.Gap. The horizontal axis represents build number. The vertical axis is the ratio of the drop in execution time or dynamic instruction count, with the first build as the reference point.

tionships are surprising, considering the highly dynamic nature of the modern processor: out-of-order execution, speculation, branch prediction, prefetching, etc. Despite these, however, the dominating factors of the execution time are so simple: dynamic instruction count and code size! In other words, in order to improve the code quality, any optimization has to reduce the numer of instructions or bytes, as its first priority.

So are the linear relationships coincidence? Does any other benchmark has such simple characteristics? We performed a larger-scale experiment: we gather totally 582 different versions of our compiler over 1 year, and use each of them, build several benchmarks at both O2 and Od optimization levels. These benchmarks include Spec2000 164.gzip, and 4 Spec2006 benchmarks, 400.perbench, 401.bzip2, 403.gcc, and 458.sjeng. For each benchmark, all its 1164 builds are run on a Dual-core Amd Opteron Processor 2200 machine with 2.79Ghz and 4G memory; the following hard-

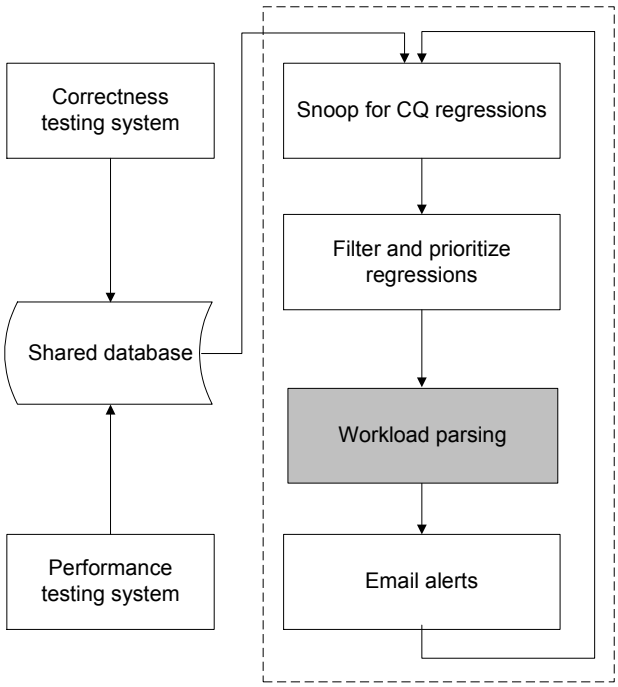


Figure 4: Integrate automatic CQ analysis into the compiler code quality engineering process. The software system we implemented for the analysis is enclosed in the dotted box.

ware events are sampled in the execution of each build with CodeAnlyst profiler [1]:

- CPU clocks;
- Retired instructions, branches, and mispredicted branches;
- Instruction cache fetches, misses, and refills from L2 cache;
- Data cache accesses and misses, and misalign accesses;
- L2 cache requests and misses;
- ITLB/DTLB misses at both L1 and L2 levels;
- Instruction fetch stall cycles, and dispatch stall cycles.

Again, linear relationships show up between execution time (CPU clocks) and dynamic instruction count (retired instructions), and between execution time and code bytes, which is approximated by instruction cache fetches. To verify, we create a linear regression model for Spec2000 164.gzip with 868 of its builds, and use the other 296 builds as a test group. With the model, we predict the execution time of the builds in

the test group; the error, defined as (predicted CPU clocks - actual CPU clocks)/actual CPU clocks*100%, was between -2.7% and 3.8%, which is pretty accurate [2].

There are several noticeable points: first, the benchmarks were built at both O2 and Od optimization levels in the experiments, and the compiler versions span a long period. This implies that the events are not dependent on specific builds or optimizations; they are characteristics of the benchmarks themselves.

Second, there are more than one characteristic event for a benchmark. They are highly correlated. Table 1 summarizes the characteristic events that dominate the experimented benchmarks. The correlation coefficient between them are no less than 0.9 [3].

From the table, it can seen that dynamic instruction count (retired instructions) appears in every benchmark as to be characteristic. Dynamic code bytes (or instruction cache fetches) appear in most of them. Both events suggest that the workload upon the processor determines the execution time. Besides, some other events are also important, including retired branches and data cache accesses.

Third, it is the *numbers* of these characteristic events, not the order of them, that determine the execution time. With software analysis, these events like dynamic instruction count and code bytes can be accurately calculated, and the others likes data cache accesses can be indirectly reflected, for example, by the total number of loads/stores, or loads/stores crossing cache lines or memory pages, etc. This suggests a software system that parses the workload to calculate statistics for the characteristic events, and from that, accomplishes code quality analysis. It needs record only the counts of these events, but not their order, which would save considerable time and space.

Finally, the linear relationships shown above have high accuracy, but are not 100% accurate. For example, dynamic instruction count is found to be a most accurate predictor of the execution time. However, there may be cases it cannot explain. For instance, Fig. 3 shows history curves of our compiler framework on the drops of the execution time and dynamic instruction count of spec2000 254.gap. The instruction count and the execution time correlates well, except in a period when the instruction count dropped slightly while the execution time increased significantly. Therefore, the software system should provide detailed information on instruction mix and Intermediate Representation (IR) to facilitate exploring CQ issues.

In Section 0.2.5, we will illustrate how this mysterious regression is diagnosed by our software system.

0.2.2 Integrate Automatic Code Quality Analysis into the Daily Testing Infrastructure

We propose a software system for CQ analysis. It is loosely coupled into the existing testing infrastructure, composed of a correctness and a performance test system, through a shared database. See Fig. 4.

[2]The best results of performance prediction we learned have the absolute relative error about 8% [12].

[3]Correlation coefficient indicates the strength and direction of a linear relationship between two random variables. The closer the absolute value of the coefficient is to 1, the stronger the correlation between the variables. Here 0.9 is a pretty high value, showing that the effects of these characteristic events on the execution time are mostly overlapped.

The analysis system works iteratively. In each iteration, it snoops the database for any new regressions logged by the performance test system. When new regressions are found, it chooses the worst unanalyzed regressions from each responsible checkin, analyzes them via workload parsing and sends out email alerts to the corresponding developers. For the regressions not chosen in this iteration, they may be picked up in the next iteration. In this way, we guarantee that the latest and the worst regressions are handled promptly as they appear.

As said before, there are two kinds of CQ analysis: regression and potential analysis. Regressions are common, and are better to be resolved early. Thus regression analysis has to respond in real-time. Identifying CQ potentials is a long-term work, and more tolerable to the analysis time. Therefore, the analysis system is configured for analyzing regressions by default, and potential analysis is performed upon request. For regression analysis only, the path caches are turned off, no inter-procedural slicing is performed, and the 95% instruction footprint is not identified.

The kernel of this software system, i.e., workload parsing, is also made an independent tool so that every developer can conveniently run it for analyzing code changes on his or her personal computers.

0.2.3 Workload Parsing

Fig. 5 shows the work flow of the workload parsing approach. It consists of a simulator, one or more online analyzers, a static analyzer, and a diff-and-translate engine.

The simulator and the online analyzer(s) cooperate to find out basic statistics, including the virtual memory layout, instruction hit counts and hot execution paths. They communicate through a shared buffer. The simulator simulates a benchmark binary and writes the IP (Instruction Pointer) of the current instruction into the shared buffer if the instruction causes a control flow transfer, and the online analyzer(s) read the IPs from the buffer. The buffer is divided logically into a number of equally-sized chunks. The simulator or an analyzer exclusively controls one chunk until it is full or empty, when it releases the chunk and tries to acquire another chunk. This is a classic single-writer multiple-readers synchronization problem. All the online analyzers are the instances of the same code and work in parallel on different chunks to maximize the speed in consuming the data in the shared buffer. Each of them has its own path caches to identify hot paths independently. After the simulation ends, the online analyzers merge their results together as the online analysis report, an XML file.

The static analyzer then raises the benchmark binary to IR, builds control flow graph, maps to IR the IPs recorded in the online analysis report, and calculates various statistics for the characteristic events at both function and basic block levels. Optionally, it detects the 95% instruction footprint, and performs backward inter-procedural dataflow analysis along the hot paths to identify redundant and dead instructions. Note that the static analyzer achieves these by orchestrating various components of the compiler framework. In this sense, the compiler uses its static analysis power to identify its own CQ issues and potentials.

When addressing a CQ regression, the above process is applied to the baseline and test binary of the benchmark, respectively. Their results are compared and translated into HTML files by the diff-and-translate engine.

Key Feature and Design Tradeoff

To make the approach useful in product environment, the analysis time must be acceptable without compromising the analysis results. To achieve this aim, the key design tradeoff is to perform most of the analysis work statically. Only the basic statistics that cannot be calculated statically are collected on the fly. This is the key feature that makes it different from other analysis tools [9, 18], and the key design tradeoff that makes it fast enough for analyzing most Spec benchmarks with the more realistic train input, instead of the trivial test input, as compared with whole program paths [8].

The major reason for the tradeoff is that dynamic profiling and analysis can be time-consuming. In our experimental data to be shown in Section 0.4, they consume almost all the time for any benchmark. The huge amount of data generated into the shared buffer also makes them memory-bound. In contrast, the static analysis time is negligible.

In this paper, we have used Nirvana simulator framework [3] for dynamic profiling. Similar to PIN and Valgrind [9, 10], Nirvana dynamically translates instructions, and allows user extension to easily collect statistics of instruction and sub-instruction level events dynamically. Although it is a state-of-art tool, the simulation speed makes it a challenge to do sophisticated analysis dynamically for real applications. Besides, it lacks the ability to build IR and control flow graph.

To further reduce the time spent in dynamic analysis, we identify hot paths with path caches instead of whole program tracing and online compression [8]. In the early stage of developing workload parsing approach, we did attempt online compression by using a commercial Xceed Zip library to dynamically compress the shared buffer. Although the compression ratio is high (around 98%), the time is long and the resulting output size is huge: for a trivial program simulated for only a few minutes, the online compression can take half an hour to finish, and produce a zip file over 100M bytes. We gave up the compressor for a few reasons: first, the decompression afterwards might take as long time. Second, it might be tough for it to handle longer programs or inputs: we noticed whole program paths [8] mainly used the smallest dataset (test input) for Spec95. Third, compression identifies the repetitive structure of the whole program, which takes considerable time and space. However, the information found is more than enough for identifying hot execution paths: such paths usually appear in loops and should not take much space in the results. In other words, most of the compressed information will not be used. For these reasons, we decided to use path caches, which dynamically consume and then throw away the IP flow, saving both time and space.

Below we introduce the approach in detail.

Simulator and Online Analyzer(s)

The simulator simulates the benchmark binary, and records the memory layout and IP flow into the shared buffer. More specifically, before simulation starts, the virtual space of the binary is queried; the layout information of all the DLLs and the binary itself are recorded. During simulation, whenever a DLL is loaded or rebased, the updated information is also recorded. When a control flow transfer happens, the IP of the instruction and the target IP are recorded. A control flow transfer is caused by an

unconditional or conditional jump, a call or a return.

An analyzer reads from the shared buffer the memory layout and the IP flow. For the memory layout, it simply copies it. For each IP, it computes its hit counts. Optionally, it creates a set of configurable path caches to identify hot paths.

Path Cache A path cache is used to identify the hottest execution paths with a specific length. It has a set of lines. Each line can contain exactly one path, and is associated with the hit count of the path.

A path is defined as a set of contiguous IPs in a chunk of the shared buffer. The number of the IPs is its length. An online analyzer reads the IPs from the chunk sequentially; the current path is the current IP and the IPs before it within the limit of the length.

The path cache works just like a hardware cache. It is queried to see whether the current path is in the cache or not. If it is, i.e., the cache is hit, the hit count of the path is incremented. Otherwise, a cache line is allocated for the path. When the cache is full, a line is evicted with the Least Frequently Used replacement policy.

To quickly decide whether a path hits or misses the cache, a cache line is also associated with two tags. In deciding a cache hit or miss, the tags are computed for the path and compared with those of the cache lines.

The first tag is the sum of all the IPs in the path. The second tag is defined as

$$((max\&0xFF) << 8|(min\&0xFF)) << bits|xorSum,$$

where max, min, and $xorSum$ are the maximum, minimum, and exclusive OR of all the IPs in the path, $bits$ is the number of the bits of an IP minus 16. In other words, the second tag is the exclusive OR of the IPs, with the highest 2 bytes replaced with the lowest bytes of max and min, respectively. This tag further improves the accuracy of matching: Since the IPs in a path are usually close in values, they mainly differ in the lower bytes. An exclusive OR of the IPs will result in 0's in the higher bytes; to add more useful information, the lower bytes of the maximum and minimum IPs are put there.

According to the above definitions, the value of a tag is not affected when the IPs in the paths are permuted. Therefore, paths due to a loop, e.g., aba and aab, have the same tags, and reside in the same cache line.

Static Analyzer

The static analyzer uses and extends the power of the compiler framework to identify CQ issues and potential. The framework has two ways to be extended. One way is that the framework serves as the "driver", and the extension serves as a "passenger". In this case, the extension is a plug-in to certain compilation phase of the framework. Multiple extensions can be plugged into the framework at the same or different phases. In the other way, the extension is the "driver": the extension is a standalone application, and it invokes some phases of the framework to perform some tasks.

The static analyzer adopts the second way. As the "driver", it invokes the framework to raise the binary to IR and build a flow graph for each function. It then reads

IPs from the online analysis report, and maps them to the IR. Then it analyzes the instruction mix at basic block and function level. It also identifies the 95% instruction footprint, which inherently includes the information of the hottest functions and basic blocks as well. Finally, it explores the hot paths via backward slicing for potential optimization opportunities.

The algorithm of processing the hot paths for potential analysis is shown in Fig. 6. First, for all the functions that have a basic block in any hot path, their alias models, originally from different scopes (functions), are merged into a single scope. This in effect transforms the inter-procedural problem into an intra-procedural problem. All possible storages, including registers, global variables, local variables, heap and stack, are described in the alias model. From the alias model, we can find out data dependences, and then recognize dead or redundant instructions.

For each hot path, its instructions are scanned backward. The data dependences between the instructions are identified on the fly. Two sets, $reachingDefs$ and $reachingUses$, are maintained for this purpose. They record the effective definitions and uses that reach the current instruction in backward direction. A dependence exists if a definition operand of the current instruction may partially cover a reaching definition or a reaching use. By "cover", we refer to the fact that according to the alias model, the storage of an operand contains that of the other operand. For example, for x86 architecture, a register EAX covers its lower half register AX.

Among the data dependences, flow dependences are used for identifying redundant instructions, while output dependences for identifying dead instructions. An instruction is redundant if removing it does not change the semantics of a path. It is dead if all its definition operands will be killed without being used in forward direction.

A flow dependence links two instructions. Redundancies may appear as one of the instructions or both. The first (the current) instruction is first checked to see if it a register-register move instruction, as this is a common source of redundancy. If the definition operand is replaceable with the source operand, we might propagate the source operand directly, and thus the instruction is redundant. Otherwise, both instructions are checked. If they are inverse instructions, like push/pop, sub/add, load/store, etc., that offset the effects of each other, and the first instruction's definition operands are not used anywhere between them, then both instructions are redundant.

If the current instruction is not redundant, its output dependences are further checked. If due to the output dependences, all its definition operands will be killed without being used, it is dead.

If the current instruction survives, the reaching definitions and uses sets are updated. We remove any definition and use from the sets, when they must be covered by the definitions of the current instruction.

Note that in the algorithm, the identification of dependences, and the updating of the two sets, are conservative: a dependence is said to exist when an operand *may* cover the other, while in updating the sets, a definition or use is killed when an operand *must* cover it.

0.2.4 Diff and Translate

This step is for comparing the analysis results of two different builds of the same benchmark. For each build, the static analysis report is produced through the process described above. The two reports in XML file format are then compared with an XML style sheet script. The results are translated into HTML files for friendly display.

The additional results from the comparison include the most different functions, and the comparison of their IR and instruction mix. Hyper links of the various results are inserted. In general, a developer addresses CQ regression from looking at the most different functions of the test and baseline binary, and exploring the hyper links among the HTML files.

0.2.5 Illustration of Workload parsing

In Fig. 3, we have shown a mysterious regression with Spec2000 254.gap when dynamic instruction count drops slightly, the execution time increases significantly (4.35%). This section illustrates regression analysis with it. We also illustrate potential analysis with history results from Spec2006 401.bzip2.

For the regression shown in Fig. 3, Fig. 7 shows some snapshots from the output of our software system. A high-level summary is shown in Fig. 7(a). It shows various statistics, including DLLs loaded, dynamic code size and instructions, floating-point instructions, flows, misaligned flows, function calls, memory accesses, misaligned accesses, and accesses crossing cache lines or memory pages. Here "flows" refer to control flow transfers. All the statistics suggest the test binary is better than the baseline, except the misaligned flows are significantly higher (9.62%).

Fig. 7(b) shows the most different functions between the test and baseline binaries, in terms of dynamic instruction count. There are two functions different. In their instruction mix, both have more `mov` instructions.

Fig. 7(c) shows 3 snapshots from the IR comparison result for the first function `CollectGarb`. The comparison is done by the diff-and-translate engine by composing the IR into text files and invoking a commercial text comparison software called Beyond Compare 2. The differences are automatically highlighted. The test binary is at the left side. The first snapshot shows that the test binary has 8M more instructions and 50M more misaligned flows. The second snapshot clearly says that there is one more instruction, `RBX=mov RCX`, which is responsible for most of the dynamic instruction increase in this function. The last snapshot compares two basic blocks between the test and baseline binary. There is no difference in the code, but there is a subtle difference in the code layout: the test binary's code starts from offset `0x2a1`, while the baseline's code starts from `0x2a0`. Because the test binary's offset `0x2a1` is not divisible by 16, each time the control flow is transferred into this block, there will be penalty for misalignment. The compiler does not align it because it needs 15 bytes to pad, while the compiler heuristically does not pad more than 8 bytes. Note this block is very hot, and the misaligned flow to it explains most of the increase in misaligned flows in the whole function.

The regression was fixed with an improvement in the heuristics of register preferencing. It exactly removed the extra `mov` instruction, which enables the block to be

aligned cheaply. After the fix, the misaligned flows dropped by 7.51%, and the execution time dropped by 3.28%, compared with the regressed binary. Other statistics remain roughly the same. This eliminated the spike in Fig. 3.

Fig. 8 illustrates potential analysis with a piece of a hot path of Spec2006 401.bzip2. Only two basic blocks are shown here for space. Read bottom up. The 2nd instruction is a redundant register-register move because EAX has only one use, and it can be replaced by R11D directly. The 5th instruction is a redundant store as it does not change the content of the memory. The 6th instruction is dead because it defines register EAX, which is overwritten by the 4th instruction without being used (Ignore the redundant 5th instruction between them). The other instructions highlighted are redundant or dead for similar reasons.

0.3 Event Tree Comparison

In this section, we consider how to diagnose CQ regressions with a less common, but more difficult scenario: operating system benchmarks. These benchmarks are important when the compiler intends to build an operating system. They evaluate the code quality of the OS built by the compiler.

The profiles of these benchmarks are essentially different from those of computation-intensive benchmarks. They may spend much time in the OS kernel, and have flat profiles. Kernel-mode code does not easily allow simulation. A flat profile does not have hot spots. Besides, these benchmarks are often concurrent and have multiple processes.

The operating system has built-in instrumentation for the purpose of correctness and performance debugging [13]. The instrumentation records a configurable set of OS events when a benchmark runs. These events are raised by both the benchmark and kernel-mode services. They include, for example, context switches, DLL load, disk IO, page faults, process start and end, and so on. Naturally we hope to detect CQ problems by analyzing traces of these events.

The challenges in analyzing the event traces are from the volume of the data and their dynamic nature. First, an event trace tends to be huge. For example, by tracing only the very basic events, a small test executing only for 16 seconds generates a trace of 15M bytes, with 220K events in it. If system calls are also traced, the trace grows to 60M bytes, with 2 million events. It is overwhelming, if not impossible, for human to explore the huge amount of events, let alone understanding their relationship with code quality. Second, a benchmark may start multiple processes, but we usually care only a few important ones among them. The events of the processes may be interleaved in non-deterministic order, while multi-core processors further interleave the events from different CPUs. Consequently, running the same benchmark twice produces two traces that might be different in many events in terms of their total numbers and orders.

To meet the challenges, we propose a divide-and-conquer approach. The goal is to find out the functions that are responsible for the code quality regression. A key observation is that despite the dynamic nature, a stable benchmark tends to have some *deterministic events*. These events have consistent numbers and orders regardless of the specific runs. We can construct a tree from a trace, where a node contains a set of contiguous events from the trace, and these deterministic events are the boundaries

between two sibling nodes.

To address a regression, two traces can be collected by running the baseline and the test binary. Stack-walking is turned off because it will disturb the execution of the benchmark, while we need accurate time stamps for the events in the traces.

The trees for both traces can be constructed with the deterministic events. Because these events are deterministic, and the traces are divided by them into tree nodes in the same way, the two trees have exactly the same structure. Therefore,the two trees can be compared node-wise. The nodes with the biggest difference in execution time can be found.

Then the problem is how to find out the hottest functions in these nodes. To do this, we can turn on stack-walking, re-collect the traces for the test binary, construct the tree for it with the same deterministic events, and locate the nodes corresponding to the most different ones in the trees without stack-walking. The stack-walking information records the functions in the call stack when an event happens. With it, we can find the hot functions in a node easily. The task of analyzing the functions is left to human, as the traces contain only high level OS events, not instruction-level events. However, as we said before, a major difficulty in analyzing this kind of benchmarks is the lack of global hot spots. Therefore, being able to locate local hot spots removes a major obstacle.

The algorithm for constructing and comparing the trees are shown in Fig. 9. The parameter $DEvents$ is an array; each element $DEvents[i]$ contains a set of deterministic events at tree level $i (i \geq 1)$. $FocusProcs$ is the set of the processes we care.

0.3.1 Illustration of Event Tree Comparison

Fig. 10 shows an example extracted from a history regression of a benchmark for Windows Control Panel. It extremely simplifies the original problem for easy understanding, but still keeps the flavor of it.

Fig. 10(a) shows the two traces for the test and the baseline binary. There are only 7 events, including a process start and end (PStart and PEnd), 3 image loads (ILoad), and 2 enumerating registry keys (RegEnumerateKey). we annotate two events of the same kind with numbers to differentiate them. The time stamps when these events happen in each trace are shown to the right.

Assume there are two levels of deterministic events. The first level is ILoad, and the second level RegEnumerateKey. A tree can be constructed out of each trace with these events. Because the structure of the two trees must be exactly the same, we merge the trees and show them together in Fig. 10(b) to save space. The root node of a tree representing the whole trace is divided into 4 children by ILoad events. Then the third children is further divided into 3 children by RegEnumerateKey events.

In each node, the first line shows the first and the last event; the second line is the latency between them for the test and baseline binary, respectively; and the last line is the difference between the two latencies. The bigger the difference is, the bigger a regression has happened in this node. We highlight in darker color the path along which the nodes have the biggest difference at each level. We can focus on the leaf node of this path to analyze the regression.

Although not illustrated here, we can now turn on stack walking, recollect a trace for the test binary, construct a tree for it, find out the leaf corresponding to the leaf said above, and identify hot functions from the stack-walking information. The analysis of these local hot functions is then left to human. In practice, analysis results of the regression, from which the above example is extracted, shew the necessity to develop inter-procedural constant propagation, which has led to visible CQ win across the entire benchmark system.

0.4 Evaluation of Productivity

This section evaluates the productivity of workload parsing and event tree comparison in our CQ engineering process.

The software system highlighted in the dotted box in Fig. 4, where workload parsing resides, is run on a machine with 2 Quad-core Amd processors with 8G memory. For workload parsing (Fig. 5), the shared buffer is configured as 6G bytes, divided into 12 chunks, read by totally 6 online analyzers in parallel. The configuration is experimentally decided to make full use of the modern machine's power. The system analyzes regressions by default. When potential analysis is requested, each online analyzer is configured with a path cache that has 512 lines, each line contains 15 IPs, and the analysis for footprint and slicing is turned on.

The productivity of the analysis is measured by the time it takes. Table 2 shows the time for Spec2000 and Spec2006. Each benchmark is simulated with a train input by default, except a few with test or reference inputs, as annotated by (test) or (ref) in the table.

The average time for analyzing a Spec2000 and Spec2006 regression is 18 and 26 minutes, respectively. Most of the benchmarks can be handled under half an hour. Overall, the response time is satisfying.

When both regression and potential analysis are performed, each benchmark takes about 1.5 hours and 2 hours on average, respectively. Since potential analysis is for long-term investigation and not a routine daily task, the response time is acceptable.

The implementation of the event tree comparison algorithm shown in Fig. 9 recognizes events from the trace files with an internal tool called TraceEvent, and then constructs the trees and compares them. Table 3 shows the statistics for the internal benchmarks we analyzed. Except for the benchmark on loading libraries, all the others are analyzed very fast.

0.5 Best Practices

We summarize our experiences in bringing up the code quality of our compiler framework, and establish a few general guidelines as the overall strategy to achieve CQ objectives with minimized efforts.

1. For the quality of code produced by the compiler for an operating system, start from an overall evaluation via system-wide analysis.

In system-wide analysis, the operating system is compiled and instrumented at basic block level; all the OS benchmarks are run, and the execution frequencies of the basic blocks of the OS are collected. Top functions and basic blocks in both kernel- and user- mode can be found. Solving the common problems from these functions and blocks lead to global improvement in code quality.

2. For individual benchmarks, start from trivial ones.

   It takes the shortest time with limited resources to scrutinize trivial benchmarks and flush out common issues.

3. Prioritize computation-intensive benchmarks over others.

   These benchmarks are stable in execution time, and the execution time truly reflects code quality due to the minimal hardware noise. The software system described before also makes it much more productive to work on these benchmarks.

4. Prioritize benchmarks with the worst code quality.

5. Prioritize machine-independent optimizations.

6. Focus on code generation for single processor first.

   Although multi-core architectures are becoming prevalent, it is fundamental to first achieve the best code generation for the traditional single-processor setting. Not only does this simplify the tuning efforts, but also the execution time has less noise and more directly reflects the code quality.

0.6 Strength and limitations of Analysis Approaches

For high productivity, it is essential to know under what scenarios, what approaches to use, and their strength and limitations. There might be no optimal approach due to the empirical nature of CQ analysis. But there can be certain approaches more effective than others under specific context.

During the process of engineering the code quality of our product compiler framework, we have attempted a wide range of techniques for analyzing code quality, including dynamic profiling, statistics, dynamic profiling plus dynamic and static analysis, manual inspection, micro-benchmarking, debugger extension, and system-wide analysis by instrumenting the OS. See Fig. 11. The approaches that we heavily rely on are highlighted in bold font. This section empirically compares these approaches.

Dynamic profiling collects statistics of a benchmark during its execution via software and/or hardware. Software approaches include software simulation and instrumentation [3, 11, 10, 17]. Hardware approaches include hardware simulation, and sampling with hardware performance counters [15, 1, 6]. Statistics at various granularity levels can be collected: OS events, instruction events, or sub-instruction events.

Based on dynamic profiling, there are tools helping CQ analysis with friendly graphic interfaces [15, 1]. They have been widely used for identifying global hot spots

and hardware-related performance bottlenecks. On the other hand, they are not specifically designed for comparing profiles and identifying subtle differences like workload parsing does, nor for analyzing benchmarks with flat profiles like event tree comparison does. For example, a CQ regression under 1% might be caused anywhere in the benchmark, not necessarily in a hot spot. Effectively using these tools requires experiences. The tuning process is human-computer interaction, and not automatic.

Statistics is the foundation of design of computer architectures and compilers. Among the statistical approaches, linear regression modeling and correlation analysis are commonly used. They can characterize the workload and expose benchmark characteristic events. As shown before, their results have motivated the two approaches in this paper. Building a reliable linear regression model needs a lot of data, and data collection can be time-consuming, though.

Workload parsing combines the strength of dynamic profiling, dynamic analysis, and static analysis. It automates analysis of computation-intensive benchmarks. As this kind of benchmarks is the key scenario for compiler development, it greatly helps the productivity. It uses simulator for dynamic profiling, which leads to the following limitations:

First, its throughput is limited by the simulator. So far, this approach is sufficient for our needs (See Section 0.4). If necessary, the limitation can be overcome by replacing simulation with instrumentation: The compiler (or any instrumentation tool like ATOM [17]) instruments the benchmark binary; and further, to reduce the volume of data written into the shared buffer, the instrumentation can be such that the data are path identifiers [8, 2], instead of IPs at each control flow change. This would be another example of using the power of the compiler itself to help generating better code. We leave this as a future subject.

Second, simulation does not provide actual execution time for functions or basic blocks. In future, we can compensate for this either by instrumentation, as described above, or by collecting actual execution statistics with sampling tools [15, 1], and teach the static analyzer to map these statistics to IR.

In workload parsing, we use path caches to detect hot execution paths, instead of online compression [8]. Eviction of cache lines due to the limited cache capacity is the major limitation to the accuracy of the results, as a hot path (cache line) may be prematurely evicted. However, this can be overcome by setting reasonable capacities for the caches. As discussed in Section 0.2.3, using path caches instead of online compression is a key choice to enable us handle Spec benchmarks with relatively realistic input within acceptable time.

In workload parsing, we have applied backward slicing to detect redundancies or deadness on hot paths. They provide hints what new optimizations are needed. However, as the analysis focuses on a path, instead of the whole control flow graph, the instructions found may be partially, not globally, redundant or dead. This is still relevant for profile-guided optimizations, though.

Event tree comparison is a static analysis based on instrumentation. It is suitable for operating system benchmarks without global hot spots. Other approaches do not address this kind of benchmarks easily. The strength of this approach is that it provides a general way to identify some local hot spots instead, which reduces human burden and limits their focus. It requires human to recognize deterministic events first. However,

this has to be done only once for every benchmark. The main drawback is that the analysis results are function level due to the limitation of the trace information.

Micro-benchmarks are unit tests for code quality, often extracted from real applications. They are convenient for identifying issues specific to certain optimizations. However, in extracting micro-benchmarks, it must be careful to keep issues relevant to the real benchmarks.

Writing an extension to the operating system's kernel-mode debugger is the only way we found to trace the OS kernel behavior instruction by instruction. With the extension, the kernel can be stepped and statistics can be collected per instruction. Unfortunately, it is extremely slow, and often affects kernel behavior by causing more time-out failures. So practically this approach is not useful.

Manual inspection of assembly code is a common activity of a compiler engineer. Often, simply reading through the assembly code can find many issues. It requires experiences with the architectures, and its productivity is human-dependent.

The above discussion is on analyzing a single benchmark. In our experience, it is helpful to perform a system-wide analysis first by instrumenting the entire operating system. This has been discussed in Section 0.5. Note that the instrumentation here is different from the built-in instrumentation of the OS. This instrumentation is to catch the execution counts of the basic blocks of the OS code, while the latter is to trace the high level OS events raised by the benchmark.

0.7 Related work

Besides dynamic profiling, statistics, online compression, which have been discussed in Section 0.6, there are several other subjects related with compiler code quality analysis: workload characterization, software performance engineering, performance modeling and prediction, and adaptive compilation.

Characterizing the workload is important for hardware architecture design, software performance engineering, performance measurement and tuning [7, 14, 16]. Ideally, with models on the relationship between the workload and the performance, performance can be accurately predicted. This enables rapid exploration of hardware design space [12]. However, it is extremely difficult to arrive at such accurate models in general, given the complexity of the modern architectures [7]. Compiler adds another dimension of complexity. Fortunately, for the purpose of analyzing compiler code generation, it is reasonable to restrict to a set of workloads that mainly stress the processor (the computation-intensive benchmarks), and in this case, simple linear relationships exist with high accuracy, which motivated the workload parsing approach, as shown in Section 0.2. The other approach we proposed, event tree comparison, is also motivated by workload characterization in that it relies on the deterministic events happening in the execution of the benchmarks. In short, this paper has demonstrated the usefulness of workload characterization for optimizing compilers.

Given the pool of the available optimizations in a compiler, adaptive compilation (or automatic performance tuning) searches for a sequence of the optimizations and their parameters that maximize the performance of a benchmark for a target architecture [5, 4]. The assumption is that the optimizations are already developed and tuned.

However, in this paper, we assume optimizations are still under development, or do not even exist. In this case, we need to address regressions quickly, and identify new optimizations to be implemented.

0.8 Conclusion

Code quality analysis is an essential activity in the compiler code quality engineering process. Improving its productivity is important for the productivity of the whole process. This paper proposes workload parsing to automate the daily CQ analysis work for computation-intensive benchmarks, and event tree comparison to address regressions of operating system benchmarks. We have implemented and applied them in our compiler development. The software system for workload parsing has been integrated to the existing daily testing infrastructure, and provides developers relevant information of their code change in reasonable time. This has greatly reduced the burden of human. We have also summarized our experiences in achieving CQ objectives with minimized efforts, and empirically compared the strength and limitations of different CQ analysis approaches.

Acknowledgements

The authors are grateful to the help of many colleagues. Darek Mihocka introduced Nirvana and wrote an extension of it, which set up a starting point for the workload parsing approach. Due to the awesome work of the Phoenix compiler team, the convenient extension API and the power of Phoenix made it possible to use the compiler as the foundation of the static analyzer. Vance Morrison generously shared the TraceEvent tool for decoding OS events, which forms the basis of the event tree comparison approach. John Lin and Xiaoru Dai worked with the authors in exploring the CQ issues and approaches and provided many inspirations. Bob Davidson and James Radigan kindly supported the publication of the work and gave useful advice in improving it. Chris McKinsey has been a great resource over the course of development. Pete Steijn helps a lot in characterizing the workloads. R. Govindarajan, Jos Nelson Amaral, and the anonymous reviewers of a previous submission provided valuable feedback. Finally, Benjamin Zorn kindly sponsored the paper to be publically available as a Microsoft Research technical report.

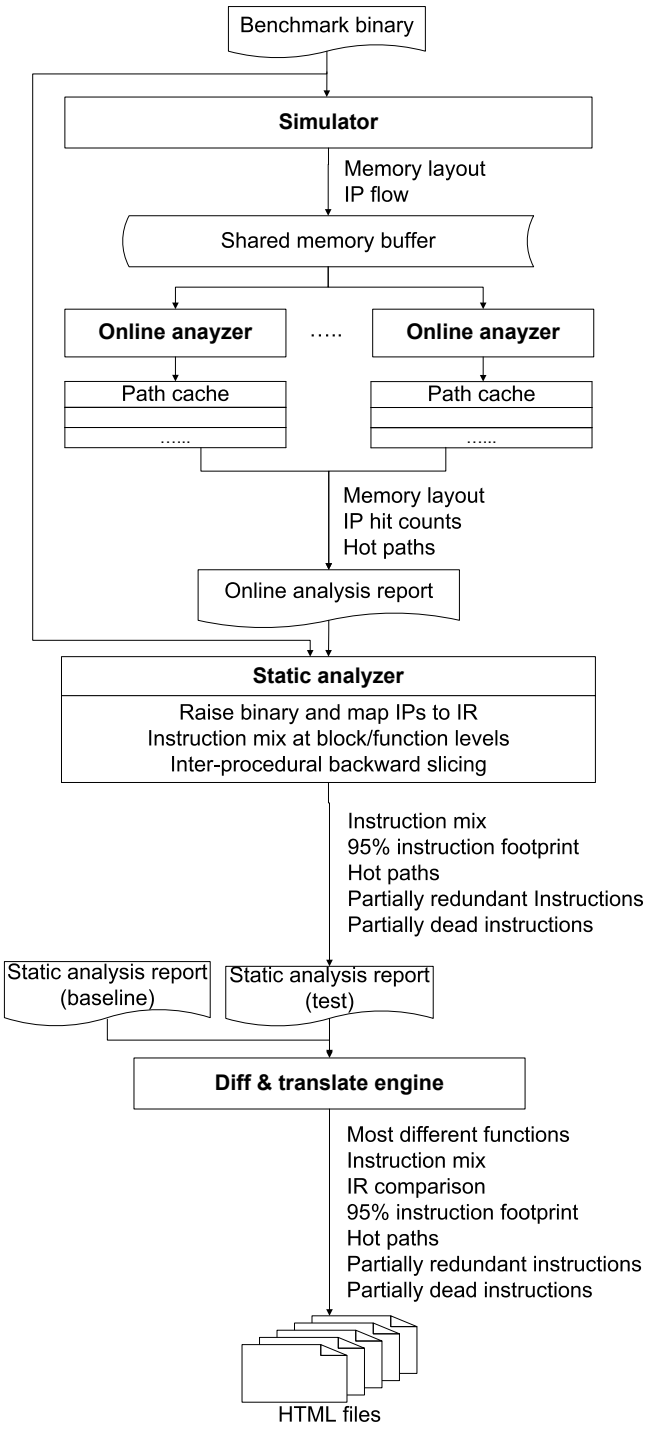


Figure 5: Workload parsing

INTER_PROCEDURAL_BACKWARD_SLICING($paths$, $functions$):

1: $aliasModel \leftarrow \{\}$
2: **for each** $f \in functions$ **do**
3: **if** f has a basic block in any $p \in paths$ **then**
4: Merge alias model of f into $aliasModel$
5: **for each** $p \in paths$ **do**
6: $reachingDefs \leftarrow \{\}$
7: $reachingUses \leftarrow \{\}$
8: $deadInstrs \leftarrow \{\}$
9: $redundantInstrs \leftarrow \{\}$
10: NextInstruction:
11: **for each** instruction $i \in p$ in backward order **do**
12: $uses \leftarrow 0$
13: **for each** $d \in \text{DEFS}(i)$ **do**
14: **for each** $u \in reachingUses$ **do**
15: //Process flow dependences
16: **if** MAYPARTIALLYCOVER(d, u, $aliasModel$) **then**
17: $uses$++;
18: **if** i is "mov d, x" and d is replaceable with x **then**
19: Add i to $redundantInstrs$
20: Goto NextInstruction
21: $uInstr \leftarrow \text{INSTRUCTION}(u)$
22: **if** i and $uInstr$ are a redundant pair **then**
23: Add i and $uInstr$ to $redundantInstrs$
24: Goto NextInstruction
25: //Process output dependences
26: $deadDefs \leftarrow \{d | d \in \text{DEFS}(i), \exists rd \in reachingDefs,$
27: $\text{MUSTTOTALLYCOVER}(rd, d, aliasModel)\}$
28: **if** $uses$ = 0 and $|\text{DEFS}(i)| = |deadDefs|$ **then**
29: Add i to $deadInstrs$
30: **else**
31: $killedUses \leftarrow \{ru | ru \in reachingUses, \exists d \in \text{DEFS}(i),$
32: $\text{MUSTTOTALLYCOVER}(d, ru, aliasModel)\}$
33: $killedDefs \leftarrow \{rd | rd \in reachingDefs, \exists d \in \text{DEFS}(i),$
34: $\text{MUSTTOTALLYCOVER}(d, rd, aliasModel)\}$
35: $reachingUses \leftarrow (reachingUses - killedUses)$
36: $\cup \text{SOURCES}(i)$
37: $reachingDefs \leftarrow (reachingDefs - killedDefs)$
38: $\cup \text{DEFS}(i)$

Notations:
DEFS(i): the definition operands of instruction i
SOURCES(i): the source operands of instruction i
INSTRUCTION(a): the instruction that contains operand a
MAYPARTIALLYCOVER(a, b, $aliasModel$):
operand a's storage may partially cover
operand b's storage, according to $aliasModel$
MUSTTOTALLYCOVER(a, b, $aliasModel$):
operand a's storage must completely cover
operand b's storage, according to $aliasModel$
$|S|$: size of set S

Figure 6: Algorithm for inter-procedural backward slicing

Summary

	Test	Base	Delta	% of Base
DLLs	0	0	0	n/a
Code bytes	24,723	24,739	-15	-0.06
Instructions	7,069	7,074	-4	-0.06
Floating-point instructions	0	0	0	n/a
Flows	1,452	1,452	0	0.0
Misaligned flows	671	613	58	9.62
Calls	147	147	0	n/a
Loads	2,031	2,035	-4	-0.2
Stores	462	465	-3	-0.73
Misaligned Loads/stores	0	0	0	n/a
Cross-cache-line Loads/stores	0	0	0	n/a
Cross-memory-page Loads/stores	0	0	0	n/a

(a) Summary of the results. Unit for absolute numbers: million

Most different functions

	Test	Base	Delta	% of Base
CollectGarb	437	429	8	1.88
Sum	126	123	2	2.33

Instruction mix and IR comparison

Function CollectGarb: IR

	Test	Base	Delta	% of Base
mov	152	145	7	5.12
add	66	60	6	11.17
sub	0	6	-6	-100.0

Function Sum: IR

	Test	Base	Delta	% of Base
dec	2	0	2	Infinity
mov	41	38	2	7.4
lea	5	8	-2	-35.89

(b) Most different functions and instruction mix. Unit for absolute numbers: million

**** CollectGarb [6 times hit. 437M instructions. 70M misaligned flows.]	<>	**** CollectGarb [6 times hit. 429M instructions. 20M misaligned flows.]

;; 6M times hit. 40M instructions.	<>	;; 6M times hit. 33M instructions.
;; 6M misaligned flows to it.	=	;; 6M misaligned flows to it.
$L26: (references=2) Offset: 321(0x0141)	<>	$L26: (references=2) Offset: 322(0x0142)
RDX = mov RAX		RDX = mov RAX
RBX = mov RCX		RAX = mov [RBX]*
RAX = mov [RCX]*		
RCX = mov RAX		RCX = mov RAX
ECX, RFLAGS = and ECX, 7		ECX, RFLAGS = and ECX, 7

;; 46M times hit. 187M instructions.		;; 46M times hit. 187M instructions.
;; 46M misaligned flows to it.	<>	;; 0 misaligned flows to it.
$L30: (references=2) Offset: 673(0x02a1)		$L30: (references=2) Offset: 672(0x02a0)
[RCX]* = mov 0		[RCX]* = mov 0
RCX, RFLAGS = add RCX, 8	=	RCX, RFLAGS = add RCX, 8
RFLAGS = cmp(IllegalSentinel) RAX, RCX		RFLAGS = cmp(IllegalSentinel) RAX, RCX
jcc(UGT) RFLAGS, $L30, $L29	<>	jcc(UGT) RFLAGS, $L30, $L29

(c) IR comparison

Figure 7: Snapshots of the analysis results illustrating how workload parsing diagnoses the CQ regression from Spec2000 254.gap in Fig. 3.

```
    ECX                 = mov [RSP+84]*
    [RSP+84]*           = mov ECX
    R13D                = mov EBX
    ECX                 = mov [RSP+108]*
    [RSP+100]*          = mov ECX
    ECX                 = mov [RSP+132]*
    R12D                = mov [RSP+112]*
    EAX                 = mov [RSP+104]*
    RBX                 = mov [RSP+304]*
                          jmp $L209

FunctionUnit BZ2_decompress:
 0x1400128cd: $L209: (references=2)
    [RSP+104]*          = mov EAX
    EAX                 = mov R12D
    [RSP+112]*          = mov EAX
    [RSP+128]*          = mov ECX
    EAX                 = mov [RSP+100]*
    [RSP+100]*          = mov EAX
    EAX                 = mov R13D
    [RSP+108]*          = mov EAX
    EAX                 = mov [RSP+84]*
    [RSP+84]*           = mov EAX
    EAX                 = mov [RSP+76]*
    [RSP+132]*          = mov EAX
    EAX                 = mov R11D
    [RSP+120]*          = mov EAX
```

Figure 8: Illustration of potential analysis. The snapshot is from a hot path of Spec2006 401.bzip2. Redundant and dead instructions are automatically highlighted in bold fonts and colors by the system. Redundant instructions are in blue, and dead instructions are in red.

EVENT_TREE_COMPARISON($test$, $baseline$, $DEvents$, $focusProcs$):
1: $trace1 \leftarrow$ TRACE($test$, FALSE)
2: $trace2 \leftarrow$ TRACE($baseline$, FALSE)
3: $tree1 \leftarrow$ Build_Tree(1, $trace1$, $DEvents$, $focusProcs$)
4: $tree2 \leftarrow$ Build_Tree(1, $trace2$, $DEvents$, $focusProcs$)
5: **for each** node $n1 \in tree1$ **do**
6: $n2 \leftarrow$ the corresponding node in $tree2$
7: $Delta[n1, n2] \leftarrow$ LATENCY($n1$) $-$ LATENCY($n2$)
8: sort the $Delta$ array
9: $trace1' \leftarrow$ TRACE($test$, TRUE)
10: $tree1' \leftarrow$ Build_Tree(1, $trace1'$, $DEvents$, $focusProcs$)
11: $localHotFuncs \leftarrow \{\}$
12: **for each** top $Delta[n1, n2]$ **do**
13: $n1' \leftarrow$ the node in $tree1'$ corresponding to $n1$ in $tree1$
14: $localHotFuncs \leftarrow localHotFuncs \cup$ HOTFUNCTIONS($n1'$)
15: return $localHotFuncs$

Notations:
TRACE($binary$, $stackWalkingOn$): The trace from running $binary$
with stack walking if $stackWalkingOn$ is TRUE
or without stack walking if it is FALSE
LATENCY(n): The time span from the first to the last event in node n
HOTFUNCTIONS($n1'$): The hot functions in the node $n1'$
according to the stack-walking information in it

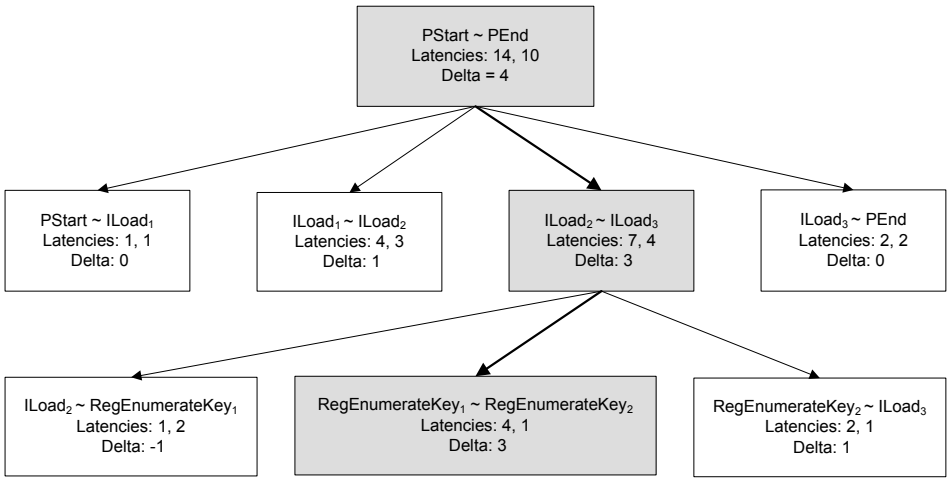
BUILD_TREE($level$, $trace$, $DEvents$, $focusProcs$):
1: create a node $root$ for $trace$
2: $firstEvent \leftarrow$ FIRSTEVENT($trace$)
3: **for each** event $e \in trace$ **do**
4: **if** $e \in DEvents[level]$ **then**
5: **if** PROCESS(e) $\in focusProcs$ **then**
6: $subTrace \leftarrow$ SUBTRACE($firstEvent, e$)
7: $subTree \leftarrow$ Build_Tree
8: ($level + 1$, $subTrace$,$DEvents$, $focusProcs$)
9: append $subTree$ as a child of $root$
10: $firstEvent \leftarrow e$
11: return $root$

Notations:
FIRSTEVENT($trace$): The first event of $trace$
PROCESS($event$): The process that generated $event$
SUBTRACE($firstEvent$, $lastEvent$):
The sub-trace starting from $firstEvent$ and ending at $lastEvent$

Figure 9: Algorithm for event tree comparison.

Event	Time stamp of the event in the test binary's trace	Time stamp of the event in the baseline binary's trace
PStart	0	0
$ILoad_1$	1	1
$ILoad_2$	5	4
$RegEnumerateKey_1$	6	6
$RegEnumerateKey_2$	10	7
$ILoad_3$	12	8
PEnd	14	10

(a) Two traces.



(b) Tree comparison.

Figure 10: Illustrating the concept of event tree comparison

	Benchmark	Regression analysis time (in minutes)	Regression and potential analysis time (in minutes)
Spec2000	164.gzip	39	234
	175.vpr	10	48
	176.gcc	6	31
	177.mesa	52	259
	179.art	4	19
	181.mcf	7	47
	183.equake	15	33
	186.crafty	23	109
	188.ammp	33	148
	197.parser	8	55
	252.eon	4	11
	253.perlbmk	31	169
	254.gap	7	39
	255.vortex	15	82
	256.bzip2	31	230
	300.twolf	9	52
	MIN	4	11
	MAX	52	259
	AVERAGE	18	87
Spec2006	400.perlbench	35	142
	401.bzip2	20	111
	403.gcc	14	34
	429.mcf	18	115
	433.milc	26	58
	444.namd	48	122
	450.soplex	9	66
	453.povrary	31	134
	445.gobmk	6	25
	482.sphinx3	14	75
	470.lbm	65	82
	447.dealII (test)	104	593
	456.hmmer (test)	13	82
	458.sjeng (test)	17	104
	464.h264ref (test)	67	246
	473.astar (test)	22	119
	483.xanlancbmk(test)	11	18
	998.specrand (ref)	1	7
	999.specrand (ref)	1	7
	MIN	1	7
	MAX	104	593
	AVERAGE	26	112

Table 2: Analysis time of workload parsing.

Benchmark	Total size of traces (in MBytes)	Total events (in Millions)	Time (in minutes)
Computer management	147	3.0	1
Control panel	166	3.0	1
Event Viewer	220	4.6	1
Game Chess	126	2.7	1
Notepad	123	2.5	1
Media player for audio	285	6.1	2
Media player for video	247	5.6	2
Media player for WMV	324	7.0	2
Internet explorer	160	3.2	1
Command shell	447	4.9	3
Load libraries	749	13.5	18

Table 3: The event tree comparison time.

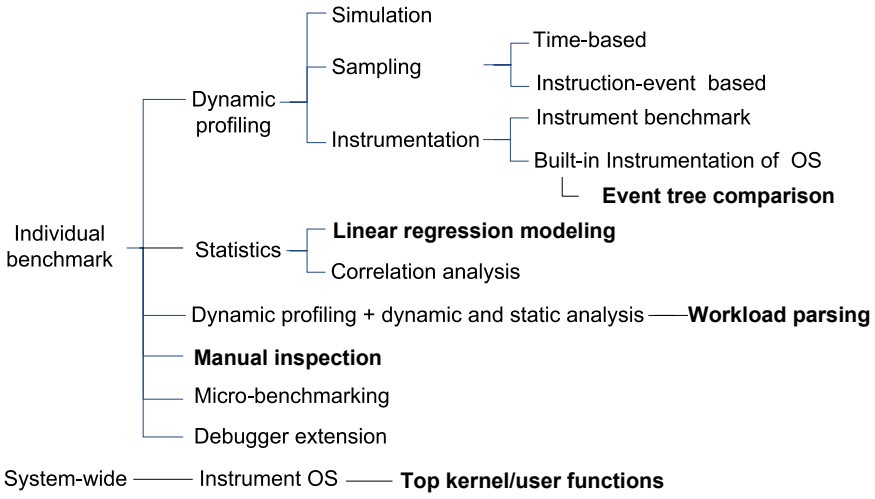


Figure 11: Techniques for analyzing CQ.

Bibliography

[1] Amd codeanalyst performance analyzer. http://developer.amd.com/CPU/Pages/default.aspx.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO'29*, pages 46–57, 1996.

[3] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE'06*, pages 154–163, 2006.

[4] C. Chen, J. Chame, Y. L. Nelson, P. Diniz, M. Hall, and R. Lucas. Compiler-assisted performance tuning. *Journal of Physics: Conference Series 78 (2007) 012024*, pages 1–10, 2007.

[5] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.

[6] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *IPDPS'03*, page 289.2, 2003.

[7] L. K. John, P. Vasudevan, and J. Sabarinathan. Workload characterization: Motivation, goals and methodology. In *WWC'98*, 1998.

[8] J. R. Larus. Whole program paths. In *PLDI'99*, May 1999.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05*, pages 190–200, 2005.

[10] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07*, pages 89–100, 2007.

[11] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *PACT'01*, pages 15–24, 2001.

[12] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. Doshi, and S. Abraham. Using model trees for computer architecture performance analysis of software applications. In *ISPASS'07*, pages 116–125, 2007.

[13] I. Park and R. Buch. Event tracing: Improve debugging and performance tuning with etw. http://msdn.microsoft.com/en-us/magazine/cc163437.aspx.

[14] R. Pooley. Software engineering and performance: a roadmap. In *ICSE'00: Proc. Conf. on The Future of Soft. Engineering*, 2000.

[15] J. Reinders. *VTune Performance Analyzer essentials: measurement and tuning techniques for software developers*. Intel Press, 2005.

[16] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, 1996.

[17] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94*, pages 196–205, 1994.

[18] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *IVME'03*, pages 50–57, 2003.