AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications

Dehao Chen
Google Inc.
dehao@google.com

David Xinliang Li
Google Inc.
davidxl@google.com

Tipp Moseley
Google Inc.
tipp@google.com

Abstract

AutoFDO is a system to simplify real-world deployment of feedback-directed optimization (FDO). The system works by sampling hardware performance monitors on production machines and using those profiles to guide optimization. Profile data is stale by design, and we have implemented compiler features to deliver stable speedup across releases. The resulting performance has a geometric mean improvement of 10.5

The system is deployed to hundreds of binaries at Google, and it is extremely easy to enable; users need only to add some flags to their release build. To date, AutoFDO has increased the number of FDO users at Google by 8X and has doubled the number of cycles spent in FDO-optimized binaries. Over half of CPU cycles used are now spent in some flavor of FDO-optimized binaries.

Categories and Subject Descriptors D.3.4 [*Processor*]: Optimization; D.4 [*Performance of Systems*]: Design Studies

General Terms Performance

Keywords Feedback Directed Optimization, Sampling, System Profiling

1. Introduction

Google spent $11 billion on capital expenditures in 2014[2], the majority of which was for production equipment, datacenter construction, and real estate purchases. With gains from Moore's law and Dennard scaling tapering in recent years, there is increasing pressure for software to operate more efficiently with existing resources. Improvements in the compiler can yield big performance gains across the board, which lead to big reductions in cost.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

CGO'16, March 12–18, 2016, Barcelona, Spain
ACM. 978-1-4503-3778-6/16/03...$15.00
http://dx.doi.org/10.1145/2854038.2854044

A common approach for improving performance is feedback directed optimization (FDO), which uses information about the code's runtime behavior to guide optimization, yielding improvements commonly in the 10-15% range and sometimes over 30%. The case for using FDO for datacenter applications is especially compelling, where even small deployments can consume thousands of CPUs. However, until recently only a few dozen of Google's largest CPU consumers had adopted FDO because the release process was too difficult to maintain. Traditional FDO follows this three-step pattern:

1. Compile with instrumentation
2. Run a benchmark to generate representative profile
3. Recompile with the profile

The nature of datacenter applications makes maintaining a representative benchmark prohibitively difficult for many users (it is even challenging for simple applications like SPEC[16]). Binaries are typically large, compiled from millions of lines of C++ code, where the most performance-critical portions are often rapidly changing[18]. They typically depend on many remote services, and the data they process is often sensitive in nature, so storing and accessing logs for loadtesting is complicated by security restrictions. Further, the overhead of instrumentation causes servers to behave very differently under load because default timeouts are exceeded. Given these constraints, isolating real-world behavior in a benchmark is a substantial barrier to entry.

In this paper, we describe AutoFDO, a system to collect feedback from production workloads and apply it at compile time. To enable AutoFDO, users need only add some new flags to their build. AutoFDO works by sampling hardware performance monitors on production machines and using those production profiles to compile the next release. Because the profile is stale relative to the newest source, we have developed compiler techniques to tolerate staleness and deliver stable performance for the typical amount of code change between releases. On benchmarks, AutoFDO achieves 85% of the gains of traditional FDO, but in practice many projects have switched to AutoFDO from traditional processes and found performance to be equivalent,

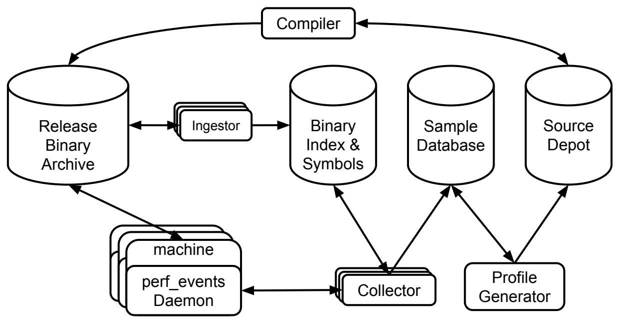


Figure 1. System Diagram.

likely because more representative input balances out less precise profiles. To date, AutoFDO has increased the number of FDO users at Google by 8X and has doubled the number cycles spent in FDO-optimized binaries. Over half of CPU cycles used are now spent in some flavor of FDO-optimized binaries.

While the concept is simple, a warehouse-scale deployment involves many interesting challenges. This paper details the novel challenges solved in developing and productionizing AutoFDO:

- Using sampled profiles to achieve performance 85% as good as with instrumentation
- Supporting iterative compilation with profiles from AutoFDO optimized binaries with stable performance
- Tolerating stale profile data
- Scaling and automation for hundreds of production binaries

2. System Overview

Figure 1 gives an overview of the system. AutoFDO leverages the Google-wide Profiler[25] (GWP) to sample hardware performance counters from production machines. Samples are stored in raw form in a Dremel[21] database, annotated with job metadata. Sampling incurs negligible overhead to the system under observation, and further processing happens offline using a relatively minuscule amount of datacenter resources.

Periodically, a batch job queries the database for aggregated raw samples for each of the top binaries. The batch job first *symbolizes* the raw samples, mapping addresses back to their source location. Finally, symbolized samples are converted to the compiler's profile format and submitted to version control[4].

During release, the compiler fetches the binary's profile from the source repository, uses the profile to annotate the intermediate representation of the compiler to drive feedback directed optimizations. The resulting binary is on average 10% faster than binaries optimized without AutoFDO.

3. Profiling System

3.1 Profiling

In order for the compiler to use the profile, it needs to be mapped onto the compiler intermediate representation (IR). We begin with a binary-level profile, then convert that to a source-level profile, which is finally converted into a standard compiler format which can be mapped onto the IR. The binary-level profile has two maps:

- A map from binary instruction addresses to their frequency. With the help of the CPU's performance monitoring unit (PMU), which is available on most of the modern processors, one can use a sampling based approach to get this profile with negligible overhead. However, due to PMU design constraints, it is not straightforward to get an accurate instruction frequency profile[9]. Experiments[10] show that by sampling the last few taken branches (e.g., LBR[24] and BHRB[22]), one can build a near-accurate instruction frequency profile.
- A map from branches to their frequency. A branch is represented as a $\{source, destination\}$ address pair. As with the instruction frequency profile, modern PMUs provide a mechanism to sample branch frequency with negligible overhead[24][22].

The binary-level profile by itself is meaningless to the compiler. Program counters need to first be mapped to source, which serves as a bridge between the binary-level profile and compiler IR. Before introducing the source-level profile, we need to first define some concepts.

An `extended source location` is a source line location annotated with inline stack information. It can be considered as a stack of source locations where the top of the stack is the source location of the sampled instruction, and other entries represent the source locations of the inlined callsites.

A `source location` is defined as a triplet:

- `function name`
- `source line offset to function start`
- `discriminator`

This triplet represents the relative source location within a function. For example, for the program in Figure 2, `foo` is inlined into `bar` at the callsite in line #7. The extended source location for the instruction at address `0x690` is `[{bar, 1, 0}]`. The extended source location for the instruction at address `0x6a2` is `[{foo, 2, 0}, {bar, 2, 0}]`.

`Discriminator`[1] is a logical term used to distinguish instructions that are mapped to the same source line but are with different basic blocks. For example, for the program in Figure 3, all instructions are mapped to the same source line, but they are distributed in 3 basic blocks thus with different discriminators.

```
Line Offset Source:       Binary:
----------------------    -------------------------
#1   #0   foo() {         foo():
#2   #1     if (cond)       0x670:  if_stmt.binary;
#3   #2       foo_stmt;     0x675:  foo_stmt.binary;
#4   #3   }
#5
#6   #0   bar() {         bar():
#7   #1     bar_stmt;       0x690:  bar_stmt.binary;
#8   #2     foo();          0x69d:  if_stmt.binary;
#9   #3   }                 0x6a2:  foo_stmt.binary;

Binary Level Profile:
Instruction Address   Sample Count
----------------------------------
       0x670               50
       0x675               10
       0x690               200
       0x69d               200
       0x6a2               100
```

Figure 2. A simple program and its binary-level profile

```
Source:        A = (foo() ? bar() : baz());
Discriminator:       D1      D2      D3
```

Figure 3. Discriminator assignment.

Using extended source location as the key for profile database lookups greatly increases profile stability because a change of one function will not affect the extended source location of other functions, nor will compiler version changes affect extended source locations.

The source-level profile is organized as a forest of trees which are composed of `ProfileNode` nodes. Each profile tree represents the sampled profile data for one standalone function in the profiled binary. Inner nodes in a tree represent function inline instances. Leaf tree nodes represent source statements associated with extended source locations. An inner node has the following properties:

- `TotalCount`: the total sample counts that are attributed to the inline instance
- `Function`: the name of the function from which the inline instance originates

A leaf node has the following profile information associated with it:

- `Count`: the maximum sample count of instructions that are attributed to the extended source location
- `Targets`: a map from indirect call target function names to their frequencies. The map forms the frequency histogram of the indirect call targets for a specific indirect call

In the source-level profile database, there is a map from the extended source locations to a tree node. An actual instruction's extended source location maps to a leaf node in the source-level profile tree. If a callsite was inlined in the profiled binary, the callsite's extended source location maps to an inner node in the tree, representing the inlined instance.

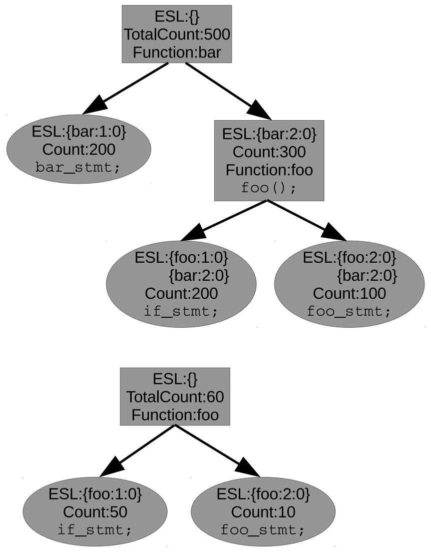


Figure 4. The source-level profile for example in Figure 2

We use standard debug info[14] embedded in the binary to convert the binary-level profile to the source-level profile. The conversion is illustrated in Algorithms 1 and 2.With this algorithm, converting the binary-level profile from Figure 2 will get the source-level profile as shown in Figure 4.

A statement at one extended source location may expand to multiple instructions. Ideally, all instructions that are mapped to the same extended source location are from the same basic block, and thus have identical frequency. In reality, debug info can be skewed in an optimized binary[10], making this assumption unsatisfied. In our implementation in GCC, we use the maximum instruction frequency as the frequency of an extended source location.

Algorithm 1 Converting Binary-level Profile To Source-level Profile

```
 1: for each address $PC$ in InstructionProfile do
 2:     esl := ExtendedSourceLocation[$PC$]
 3:     profn := Profile[esl]
 4:     count := MAX(InstructionProfile[$PC$], profn.Count)
 5:     profn.Count := count
 6:     while !esl.InlineStack.Empty() do
 7:         loc := esl.InlineStack.Pop()
 8:         Profile[esl].TotalCount += count
 9:         Profile[esl].Function := loc.Function
10:     end while
11: end for
```

Algorithm 2 Converting Binary-level Profile To Call Profile

```
1: for each branch $B$ in BranchProfile do
2:     if $B$.SourcePC is indirect call then
3:         esl := ExtendedSourceLocation[$B$.SourcePC]
4:         target := FunctionName[$B$.TargetPC]
5:         Profile[esl].Targets[target] := BranchProfile[$B$]
6:     end if
7: end for
```

3.2 Fleet-wide Collection

Each machine in a Borg[28] cluster has a standard set of daemons installed to support monitoring and system management. Among those is the `perf_events` daemon, which exposes access to a restricted set of `perf_events`[3] functionality. The GWP collector continuously crawls machines in Borg in a random order and requests a 10-second system-wide LBR [24] profile.

The AutoFDO collector covers about 10% of machines each day, so around 0.001% of observable cycles are profiled. During profiling, less than 1% overhead is observed. Coverage may seem low, but the distribution of jobs is fairly top-heavy; about 750 binaries account for 95% of CPU usage, and we obtain more than enough samples for top binaries. There are tens of thousands of binaries in total, but those further from the top are less important to optimize.

3.3 Database

For each sample the collector observes, it writes a row to a Dremel database containing the addresses of the sample and some metadata describing the context. Some examples of metadata include:

- Borg user and job identifiers
- Binary name
- Revision
- Timestamp
- Build-ID, a unique identifier for each binary usually computed by a hash of its contents at compile time.

This gives us the flexibility to do ad-hoc queries over weeks of data to instantly generate custom profiles without having to collect new data.

Ultimately, the samples need to be symbolized, and it is tempting to symbolize at this stage in the pipeline. But each sample can expand to thousands of bytes of information, and there are many billions of samples each day. Instead, we simply store sample addresses in the database and defer symbolization to a later stage.

3.4 Updating Profiles

Profiles are stored in version control so that they can be used in automated testing and to ensure that builds at the same revision will be repeatable and hermetic. Several hundred profiles are submitted weekly and are typically under a megabyte each after compression, so the growth relative to the rest of Google's repository is negligible[4].

3.4.1 Configuration

There are tens of thousands of live binaries, so we have a configuration schema to describe which binaries to generate profiles for. Configuration and profiles are stored in a separate directory hierarchy that mirrors where each original binary resides. Each binary may have one or more configuration files describing the database filters to use when selecting profile samples. The common case is one config file per binary, but some binaries are used in multiple scenarios where the code paths are drastically different, so users can filter samples to build profiles specialized for each scenario.

For quality assurance, the profile generator has some heuristics to assert that a profile is viable. For example, we typically have line numbers for a 90% of samples, and expect to have discriminators for at least 5% of samples. Some binaries exceed these constraints, so the config files are also used to relax the rules for validation after a failure is manually inspected to be a false positive. These assertions add some toil for the system operators, but have caught several instances of inadequate profile data before it reached end users.

3.4.2 Batch Job

A batch job runs weekly to submit new profiles. First, it scans the config hierarchy and loads each config file.

To avoid the complexity of merging profiles from multiple revisions, we need to choose a single revision to use samples from. There are often dozens of revisions of a binary that are live at any moment, and we aim to choose a revision that is both recent (so the source at head will be most similar) and most heavily used (so the optimizations target the most important use cases). Often, most of the revisions are for test deployments and release candidates, and only 1 or 2 are viable alternatives. The heuristic in place selects the most recent revision that has at least 20% as many samples as the most heavily used revision, looking at samples from the last two weeks.

Once a revision has been selected, the database is queried again to retrieve sample counts for that revision.

The samples are then symbolized, converted to the compiler's profile format, and submitted over the previous profile revision.

For testing, the batch job also does a dry run without submitting twice a week to alert us of problems as soon as possible.

3.5 High Performance Symbolization

When written to the database, samples need to be annotated with metadata about how the binary was built. When generating the aggregated profile, hundreds of thousands of unique addresses must be symbolized for each binary. We have developed a high-performance distributed symbolization service to fill this need for all binaries running in Google datacenters. The service is also used by profilers for ChromeOS, Chrome for Windows, and Android.

To save space, released binaries have been stripped of most or all of their metadata and symbolic debug info. Typically, to symbolize an address, we must first retrieve the original unstripped binary, load it into memory, and look up the items of interest in the DWARF[14] tables. Unstripped binaries at Google can be multiple gigabytes, so just retrieving the binary can take several minutes. Then symbolizing addresses using a tool like `addr2line` can take several minutes more and use multiple gigabytes of RAM.

An individual branch profile is system-wide, so it may contain samples from hundreds of disparate binaries and libraries. Google-wide, there are tens of thousands of unique live binaries, many of which are updated daily or weekly. The perf_events branch profile contains no metadata about the provenance of those binaries, only each binary's Build-ID, which serves as a globally unique identifier for the binary.

The symbolization service addresses the need to quickly retrieve metadata and symbols for a given Build-ID. When a release binary is built, we ingest the binary into a Bigtable[8] keyed by the Build-ID. Each row contains build metadata (e.g., the name of the binary, its timestamp, and version) and the complete symbolic line table in a format optimized for fast retrieval. During ingestion, the binary's address range is partitioned into *shards* of approximately 100,000 addresses. Each shard contains an indexed list of addresses and their corresponding function, file, line, and discriminator information.

With the symbolization service, retrieving metadata about a Build-ID or symbolizing a single address usually takes tens of milliseconds. Symbolizing all of the samples observed for a single binary takes only a few tens of seconds. For the client, it requires only enough memory to hold the data of interest, and can be done with high concurrency. Performance and client simplicity make the system easier to manage, but the symbolization service also makes debugging and manually regenerating profiles much easier for developers.

4. Compiler Support

4.1 Profile Annotation

Profile annotation takes the source-level profile and IR as inputs and produces annotated IR, i.e. control flow graph with edge and node frequency. This is a 3-steps process:

4.1.1 Preparation

The source-level profile contains inline stack information for the profiled binary and is therefore hierarchical. In the hierarchical source-level profile, all instances of a given function (inlined or standalone) are represented by different subtrees. The data stored in those subtrees collectively represent the context sensitive profile for that function. The preparation step transforms the IR and makes it resemble the inline tree structure stored in the source-level profile via selective inlining. By so doing, the compiler is enabled to fully utilize the context-sensitivity embedded in the hierarchical profile. In particular, the preparation inlining step requires:

- A top-down pass of the call graph to traverse all functions to make inlining more selective. In the call graph shown in Figure 5, $bar \rightarrow baz$ is only hot when bar is called from foo (as shown by the dotted edge $foo \rightarrow bar \rightarrow baz$. With top-down traversal, baz will only be inlined through bar into foo, but not to the outline copy of bar.
- Iteratively inline each callsite that maps to a hot inline instance in the source-level profile. Note that if a callsite is inlined in the profiled binary and recorded in the source-level profile, but the instance is not hot enough, the callsite will not be inlined in this step to avoid excessive code growth. This step also needs to invoke indirect call promotion to make sure that indirect calls are also inlinable. See Section 4.2.1 for details. The algorithm is described in Algorithm 3.

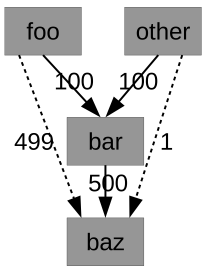


Figure 5. Context sensitive call graph

Algorithm 3 Preparation for Profile Annotation

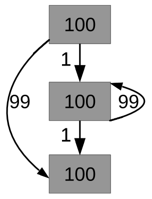
```
 1: for each node $N$ in ReversePostOrder(CallGraph) do
 2:     callsites := CollectCallSites($N$)
 3:     while !callsites.Empty() do
 4:         $C$ := callsites.Pop()
 5:         esl := $C$.ExtendedSourceLocation
 6:         if Profile[esl].TotalCount $>$ Threshold then
 7:             newsites := Inline($C$.Callstmt)
 8:             callsites.Append(newsites)
 9:         end if
10:     end while
11: end for
```

4.1.2 Annotate Basic Block Frequency

Except for the cold inline instances, all other function inline instances recorded in the source-level profile database are recreated in the IR after the preparation step. In other words, each one of the newly created inline instances has exactly one matching inner node in the source-level profile tree. The profile data for an IR statement in the inline instance can be simply retrieved through its extended source location. Algorithm 4 illustrates the algorithm to calculate the basic block frequencies from the source-level profile. In this step, the profile samples from cold instances that are not inlined during the preparation step are discarded. We can choose to combine those cold instance's profile with the standalone instance's profile, but in practice, it does not make much performance difference.

Algorithm 4 Basic Block Frequency Annotation

```
 1: for each basic block $B$ in CFG do
 2:     $B$.Count := 0
 3:     $B$.Annotated := false
 4:     for each statement $S$ in $B$ do
 5:         esl := $S$.ExtendedSourceLocation
 6:         if Profile.Exist(esl) then
 7:             $B$.Count := MAX($B$.Count, Profile[esl].Count)
 8:             $B$.Annotated := true
 9:         end if
10:     end for
11: end for
```



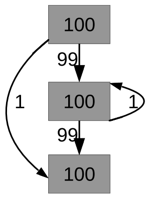


Figure 6. Basic block frequency v.s. edge frequency

4.1.3 Calculate Edge Frequency

In theory, there are cases where one cannot derive edge frequency from basic block frequency [7]. As illustrated in Figure 6, multiple edge frequency annotations could correspond to the given basic block annotation. To make things worse, the basic block frequency may not be reliable because some source may have been eliminated in the optimized binary that was used for profile collection, thus some basic blocks may not have been frequency-annotated. The heuristic used to propagate frequency across the control flow graph is illustrated in Algorithm 5. In this algorithm, an equivalence class is a set of basic blocks that are in the same loop nest, and dominate or post-dominate each other. All basic blocks in an equivalence class should have identical frequency, thus we set the equivalence class's frequency to be the maximum frequency of all basic blocks in it. We then use a flow-based heuristic to iteratively infer edge frequencies from basic block frequencies and update basic block frequencies if a profile inconsistency is detected. `AllEdgesAnnotated` returns true if all incoming or outgoing edges are annotated. `GetSingleUnannotated` returns the only unannotated incoming or outgoing edge, if one exists. Note that when there are inaccuracies, the algorithm only attempts to increase basic block frequency in order to make the algorithm terminate. For most of the cases we have seen so far, the iterative algorithm terminates in 3 iterations.

Algorithm 5 Frequency Propagation

```
changed := **true**
**for** each basic block $B$ in CFG **do**
    ec := $B$.EquivalentClass
    **if** $B$.Annotated **and** $B$.Count $>$ ec.Count **then**
        ec.Count := $B$.Count
        ec.Annotated := **true**
    **end if**
**end for**
**while** changed **do**
    changed := **false**
    **for** each basic block $B$ in CFG **do**
        ec := $B$.EquivalentClass
        **for** incoming **in** [**true**, **false**] **do**
            **if** AllEdgesAnnotated($B$, incoming) **then**
                total := GetTotalEdgeCount($B$, incoming)
                **if** !ec.Annotated **or** total $>$ ec.Count **then**
                    ec.Count := total
                    changed := **true**
                **end if**
                **continue**
            **end if**
            **if** ec.Annotated **then**
                edge := GetSingleUnannotated($B$, incoming)
                **if** !Exists(edge) **then**
                    **continue**
                **end if**
                total := GetTotalEdgeCount($B$, incoming)
                edge.count := ec.Count - total
                edge.annotated := **true**
                changed := **true**
            **end if**
        **end for**
    **end for**
**end while**
```

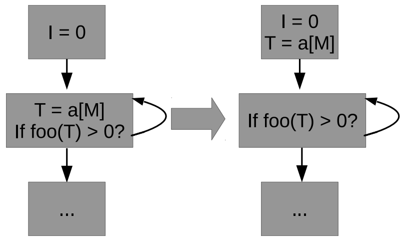


Figure 7. Loop invariant code motion

4.1.4 Sources of Profile Inaccuracies

Profile accuracy is critical to AutoFDO performance and it is known[10] that binary-level profile collected using branch traced sampling is very accurate. In fact, the main source of profile inaccuracies comes from the debug information that is used to map the binary-level profile to the source-level profile consumed by the compiler.

Since the programs profiled by AutoFDO are running in production, their binaries are fully optimized. The quality of the debug information for an optimized binary can be compromised by various compiler optimizations/transformations. The degraded debug information may in turn reduce the source-level profile accuracy and affect AutoFDO optimization. These optimizations can be roughly grouped into the following categories:

- Recoverable transformations. This type of transformations leads to profile information loss, but it can be corrected by the annotation algorithm. For example, in the CFG as shown in Figure 7, the loop invariant code motion (LCM) moves the statement `T = a[M]` outside the loop. As a result, the sampled frequency of that line would be incorrect with respect to the original source. During annotation, the basic block will still be annotated with the correct profile which comes from statement `call foo(T)`. The `MAX` operation used in Algorithm 4 and Algorithm 1 can often derive the correct basic block frequency because redundancy elimination optimizations tend to move instructions from more expensive blocks with higher frequencies to blocks with lower frequencies. In some cases, the code motion or block duplication transformation may cause basic blocks (and source statements within) to be completely eliminated. The flow-based algorithm using equivalent classes described in Algorithm 5 is designed to handle this situation. Other than LICM, partial redundancy elimination (PRE), jump threading, code hoisting, code sinking, loop unswitching, and tail duplication also fall into this category.

- Unrecoverable benign transformations. Some optimizations may change the profile significantly, such that the missing information cannot be easily recovered by the annotation algorithm. However, the profile loss is localized in one small isolated code region, and the optimization decisions on that region will not be affected by the inaccuracy. For example, the full loop unroller may decide to flatten out a small loop, which makes it impossible for profile to represent the original loop structure. However, the full loop unroller makes the decision based on static source information such as loop size and constant trip count, so the lost information for the loop region does not really matter. In addition, the scaled down frequency for the fully unrolled loop does not affect the frequency of its surrounding basic blocks, so optimizations outside the loop will not be affected either. Unreachable code elimination is another example of this kind.
- Unrecoverable destructive changes. Optimizations in this group will lead to information loss and the optimizer can be misled to make suboptimal decisions based on the incorrect profile. For example, the loop unroller uses profile data such as a loop's average trip count to decide how the loop should be unrolled. However if the loop was unrolled previously in the profiled binary, the average trip count computed from the profile data will be smaller, which can lead to the wrong unrolling decision for the loop. Optimizations of this kind include indirect call promotion, function cloning, if conversion, etc. This type of optimization is usually hard to deal with in general. Special handling is needed for each one of the cases. See section 4.2.1 for a description on how indirect function call promotion is handled. In general, whenever there is code duplication, the `SUM` operation should be used instead of `MAX` to derive source-level profile frequency. However when mapping the sampled addresses in the binary-level profile to extended source locations, there is no easy way to tell whether multiple addresses from the same location are clones or not. To completely solve this problem, the source discriminator support needs to be extended to distinguish instructions generated by code duplication. This solution will also increase the size of debug info.

4.2 Production Deployment

After the source-level profile is annotated on the IR, it is straight-forward for other feedback directed optimizations to kick in and use the profile data. However, in order for AutoFDO to be useful in a production environment, there are still two problems to solve:

4.2.1 Iterative Compilation

The AutoFDO profile for a target program is collected directly from an optimized binary running in production. The profile is used to further optimize the binary to be deployed to production and become the new profiling target. Certain feedback directed optimizations are so aggressive that special care needs to be taken when using profiles collected from a binary aggressively optimized with AutoFDO. For example, AutoFDO will speculatively promote a hot indirect call and inline it. If we profile from an AutoFDO-optimized binary, the indirect call will become much colder, and the inline instance of the promoted direct call will become hot. When using this profile, during annotation, the profile data for the inlined instance cannot be used in annotation because the indirect call promotion has not happened so there is no direct call to be inlined in the preparation step. And later in the indirect promotion pass (usually run after the profile annotation step), the call will not be promoted to the original hot target because it does not have the indirect call target profile any more. As a result, the performance will likely drop when using profiles collected from AutoFDO optimized binary. In the next iteration, the performance will recover because the indirect call that failed to be promoted in the previous iteration, appears again in the profile. In Section 5.2, we will show the performance impact of iterative compilation when indirect calls are not specially handled.

Algorithm 6 Preparation With Indirect Call Handling

```
1: for each node $N$ in ReversePostOrder(CallGraph) do
2:     callsites := CollectCallSites($N$)
3:     while !callsites.Empty() do
4:         $C$ := callsites.Pop()
5:         esl := $C$.ExtendedSourceLocation
6:         if Profile[esl].TotalCount $>$ Threshold then
7:             if IsIndirectCall($C$) then
8:                 dcall := Promote($C$, Profile[esl].Function)
9:             else
10:                dcall := $C$.Callstmt
11:            end if
12:            newsites := Inline(dcall)
13:            callsites.Append(newsites)
14:        end if
15:    end while
16: end for
```

To solve this problem, the preparation step is adjusted as in Algorithm 6 to integrate indirect call promotion. If an indirect call is promoted, inlined and proved to be hot in the profile, the preparation step will force promote and inline the indirect call. To support this transformation, the node that is mapped from an indirect call site is enhanced to have both the inner node's property (i.e. $total\_count$ and $function$) and the leaf node's property (i.e. $count$ and $targets$). The former represents the inlined target function the indirect call is promoted to, while the latter represents the default path with the unpromoted indirect call. To support more general indirect call promotion, a linked list of inner nodes can be used to track all promoted targets in the source-level profile.

Though iterative compilation adds more complexity to AutoFDO, it also brings benefits. Note that we use top-down inlining in the preparation stage. After profile annotation, we have another inline pass to inline hot callsites that are not inlined in the previous iteration. So in the next iteration, we will have finer grained context for the newly inlined hot callees. The benefit will diminish after a few iterations when the second inline pass has no more hot candidates to inline.

4.2.2 Stale Source Code

AutoFDO collects profiles from production binaries, then uses these profiles when building new releases in which the source code for a binary may have changed relative to the profiled binary. As a result, it is important to design the system to be tolerant to source changes. Instead of using {`file_name, line_number, discriminator`}[10], we use {`function_name, line_offset, discriminator`} triplet to represent the source location. This can effectively tolerate file name/path changes, and can also lock the source change within one function. I.e. if function `foo()` has some modification, even if `bar()` is in the same source file, as long as `bar()` is unchanged, the profile of `bar()` can still be used.

In practice, this approach is very effective in tolerating source changes when using AutoFDO profile. As section 5.3 show, the performance penalty is small on major data center applications. This is because their profiles are mostly flat, and it is not likely that all hot functions have source changes at the same time. Instead, source change tends to happen gradually and has minor impact between two consecutive releases. Thus for design simplicity, this is the only approach we use to tackle the source code staleness. We do not detect source changes, nor do we try to rectify the profile. We do provide a migration path for user to recover quickly from potential performance loss due to stale profile. User can quickly push a new suboptimal version to production for profile collection. Once we have collected enough samples, a new release is built with the new profile to recover from the performance loss. This migration has not been activated for a single project so far due to high performance in tolerating source change.

5. Performance Evaluation

5.1 AutoFDO vs FDO

We randomly chose some important Google internal applications, and use their benchmarks to compare performance between AutoFDO and traditional instrumentation based FDO. As we cannot collect instrumented profile from production, for both FDO and AutoFDO, profiles are collected from the benchmark itself. As can be seen from Table 1, for most benchmarks, AutoFDO can achieve >90% of the FDO speedup. The gap is mostly due to inaccurate debug info that is used to represent the AutoFDO profile. AutoFDO tends to be less effective where loop nesting is tight and complicated because:

- Feedback directed loop optimization usually requires very accurate profile info (e.g. loop trip count, etc).
- Debug info is usually polluted by aggressive loop optimizations.

For example, media encoding/decoding such as that in the encoder application usually benefits less from AutoFDO.

Application	FDO	AutoFDO	Ratio
server	17.61%	15.89%	90.23%
graph1	14.68%	14.04%	95.65%
graph2	7.16%	6.27%	87.50%
machine learning1	8.92%	8.46%	94.85%
machine learning2	7.09%	6.60%	93.06%
encoder	8.63%	3.31%	38.37%
protobuf	16.96%	14.40%	84.94%
artificial intelligence1	10.12%	10.12%	100.00%
artificial intelligence2	13.24%	11.33%	85.61%
data mining	20.48%	15.54%	75.86%
mean	12.40%	10.52%	84.84%

Table 1. AutoFDO and FDO performance (speedup against O2 binary) comparison on Google internal applications

Application	FDO	AutoFDO	Ratio
400.perlbench	15.27%	14.99%	98.17%
401.bzip	1.35%	1.00%	74.07%
403.gcc	7.73%	7.52%	97.28%
429.mcf	0.04%	2.75%	100.00%
445.gobmk	3.67%	3.23%	88.01%
456.hmmer	-0.73%	1.90%	100.00%
458.sjeng	6.19%	6.03%	97.42%
462.libquantum	-10.41%	-0.61%	100.00%
464.h264ref	1.61%	-1.75%	0.00%
471.omnetpp	4.03%	1.31%	32.51%
473.astar	8.86%	10.12%	114.20%
483.xalancbmk	14.44%	11.98%	82.96%
mean	4.40%	4.87%	112.33%

Table 2. AutoFDO and FDO performance (speedup against O2 binary) comparison on SPECCPU 2006 integer benchmarks

We also evaluated AutoFDO performance on SPEC-CPU 2006[16] integer benchmarks, as shown in Table 2. The performance are evaluated on Intel Westmere platform. The compiler is GCC (r231122 from google/gcc-4_9 branch). AutoFDO has shown similar speedup with instrumentation based FDO. Note that for some applications (like libquantum), due to lack of tuning and mis-optimization, FDO/AutoFDO has introduced negative speedups, which we have not observed from our production applications.

5.2 Iterative AutoFDO

As illustrated in Section 4.2.1, if the profile is collected from an AutoFDO-optimized binary, some feedback based optimizations would affect debug info, which will prevent the same optimization from happening in the next iteration of the AutoFDO build. We randomly choose some internal performance benchmarks to perform iterative AutoFDO compilation: for the first iteration, we use profile collected from O2 binary; for the `N`th iteration, the profile is collected from the `N-1`th iteration. As can be seen in Figure 8, most application performance fluctuates every other iteration. After integrating indirect call promotion into the profile preparation step, the iterative AutoFDO performance is much smoother, as shown in Figure 9. Later iterations also tend to perform better than the first iteration because more context is collected in later iterations. For some applications, AutoFDO performance in later iteration even exceeds instrumentation based FDO.

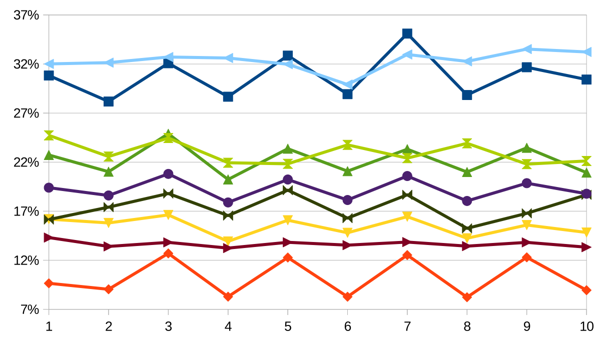


Figure 8. Iterative AutoFDO speedup (comparing with non-FDO binary) for different application (without integrating indirect call promotion into preparation step).

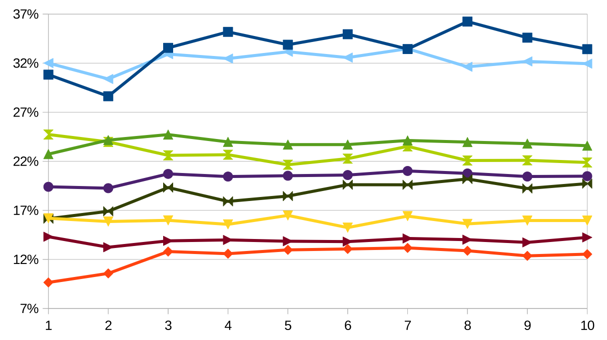


Figure 9. Iterative AutoFDO speedup (comparing with non-FDO binary) for different application (integrating indirect call promotion into preparation step).

Note that there are other feedback directed optimizations that could make iterative AutoFDO performance fluctuate. E.g. loop unrolling can distort loop trip counts in the collected profile. In the next iteration, the compiler makes less aggressive unrolling decision based on profile data, which makes unrolling optimization less effective. From our experience, for most of the C++ applications in Google, fixing indirect call promotion can solve most of the performance fluctuation issues in iterative AutoFDO.

Figure 10 shows the iterative AutoFDO performance for SPECCPU 2006 integer benchmarks. Most benchmarks show stable performance between iterations. Some benchmarks (e.g. 458.sjeng) shows variation between two iterations. This is because most of the performance benefits comes from loop based optimizations(e.g. loop unrolling), which does not work well in iterative AutoFDO.

5.3 Impact of Stale Profiles

In this experiment, we use old release binaries to collect the profile, and use the stale profile to optimize the latest source code. In Figure 11, we compared two ways to represent source location in profile: using absolute line number and using offset to the function start line. We choose an application that has a relatively stable code base, i.e. its source does not

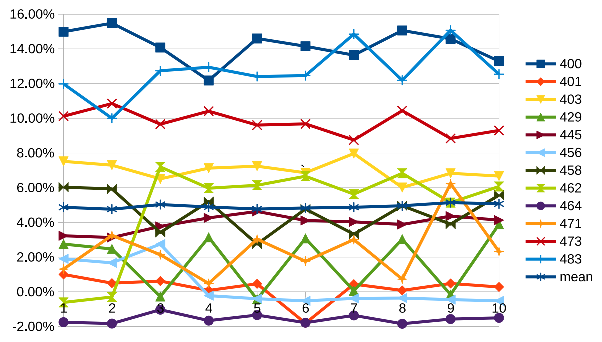


Figure 10. Iterative AutoFDO speedup (comparing with non-FDO binary) for SPECCPU 2006 integer benchmarks.

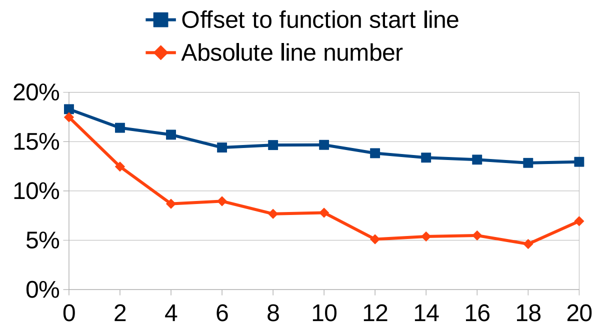


Figure 11. AutoFDO speedup (comparing with non-FDO binary) over staleness(weeks)

change very frequently. When using absolute line number as the key to represent source location, the speedup will quickly degrade as the profile becomes stale. When using offset to function start line as the key to represent source location, the speedup degrades gracefully over time as the profile grows stale. In Figure 12, we show speedup-over-staleness for 3 major applications. For applications 2 and 3, as they are already in mature state, using stale profile has nearly no impact on its performance. Application 1 is a rapid-changing project, with lots of source changes to its hottest code during the experiment's time range. For this application, using 3-weeks old profile only incurs a 2% to 3% of performance penalty; using 6-months old profile can still provide 50% of the speedup obtained from using fresh profile.

6. Experience

The process of deploying AutoFDO in Google's production data centers has exposed many lessons along the way.

6.1 Testing

Introducing a bug via compiler transformations can be an annoying burden in the best case or lead to subtle data corruption in the worst case, so testing is of paramount concern. Google has an internal benchmark suite to exercise the various compilation modes and ensure stable performance across releases. Further, when releasing a binary compiled

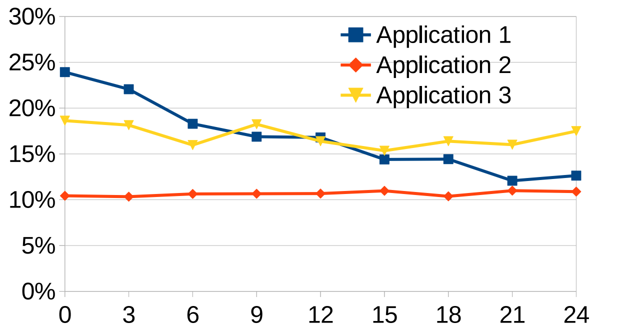


Figure 12. AutoFDO speedup (comparing with non-FDO binary) over staleness(weeks) for different applications

with FDO (in general, not just AutoFDO), teams also compile and run related tests using the generated profile, so hundreds of FDO-compiled tests may be run to qualify a binary for release. To date, only one production bug has been attributable to the compiler; a third party library exposed an optimizer bug that was not well-tested because most Google C++ applications do not use C++ exception handling.

It is somewhat common for FDO to expose latent bugs in applications. FDO can lead the compiler to make substantially different optimization decisions, especially with inlining, which can lead to failures to compile due to missing or duplicate symbols or runtime failures due to uninitialized uses and stack overflow. The risk exposed by FDO is similar to that of a normal compiler upgrade.

6.2 Stability

For latency-sensitive workloads, stable performance is especially important. Increasingly, latency-sensitive users are switching from traditional FDO to AutoFDO despite its slightly inferior benchmark performance. Their production monitoring has shown that the realized performance is equivalent between FDO and AutoFDO, indicating that imprecise profiles from real workloads are as good as precise profiles from training inputs. In one case, a user discovered that a rare error condition had poorly optimized code (because the profile suggested it was cold), which led to cascading overload of their service. They had to disable feedback-directed optimization altogether because stable performance outweighed average performance in this case. Note that this situation could arise with traditional FDO as well if the error path is not covered proportionally. To work around this issue, we are investigating disabling FDO on a per-file basis.

6.3 Context Sensitivity

Generated profiles instruction counts are proportional to real-world behavior. By default, we will generate a single profile for a given binary. However, some binaries are used by multiple teams with very different workloads, and the merged profile may not be optimal for the largest user and may actually penalize smaller users. To resolve this, we allow multiple profiles to be generated for each binary, and samples can be partitioned by any metadata relating to the binary (usually the Borg job where the samples were observed).

6.4 Utility

FDO generally is most effective on code that has many function calls and biased branches that the compiler cannot statically predict. This type of code accounts for a large fraction of Google's entire software footprint. However, there is a considerable amount of time spent in routines that are already very well tuned, like compression, encryption, and math libraries. FDO provides little benefit for those types of applications.

6.5 Release Integration

Google's release package manager has integration with the build system that makes it very easy to build a binary and simultaneously package it for deployment and archival. When building a system that requires multiple binaries (as is often the case with multi-stage MapReduce[12] jobs), it is common to put all of the binaries into the same archive. Sometimes these binaries are even sub-packaged within Java, Python, or Shell archives. This is problematic for FDO because the binaries must all be built in the same build invocation, with the same profile. Unfortunately, there is no convenient solution; each binary that is to use FDO must be compiled and archived separately.

7. Future Work

Many opportunities remain to enhance performance for existing AutoFDO users. As mentioned in Section 4.1.4, we can further improve profile quality by enhancing the discriminator to record cloned instructions. Traditionally, top cycle consumers have also enabled cross-module optimization (LIPO)[20], which can often yield an extra 5-10% performance over FDO. Compiling with LIPO is more complex than traditional FDO, and has taken more work in compiler internals and build system integration to operate with AutoFDO. We are currently in the final steps of making AutoFDO+LIPO (AutoLIPO) available and as easy to use as AutoFDO. Traditional FDO supports targeted value profiling, e.g., of the distribution of size and alignment of arguments to memcpy. We are investigating ways to obtain this information via sampling.

We are also working on ThinLTO[17], a new approach to cross-module optimization which aims to scale better than existing techniques. ThinLTO makes module grouping decisions at compile time, but in a lazy mode that facilitates large scale parallelism.

8. Related work

In a recent paper[9], sampling based profiling is used to drive feedback directed compiler optimization. The author observed that profile accuracy is very important to SampleFDO performance, and proposed to improve profile accuracy by sampling multiple events. Follow-up research [10] refines this work by sampling the LBR. It provides theoretical proof that LBR sampling can provide a near fully accurate instruction frequency profile. Our work builds on top of this paper, i.e. using LBR for binary-level profile collection. We differ in how we represent and use the source profile:

- We introduce a separate step to prepare for annotation for function inline instances. This is important for performance of C++ applications where function inlining is common.
- We use a flow based algorithm to propagate edge frequency; [10] uses Minimum Cost Circulation algorithm [19]. In theory, this approach is equivalent to those in[10], but our flow based algorithm is conceptually simple, and much more efficient than the minimal-cost circulation.

Aside from the differences in compiler implementation, this paper also discusses issues discovered while deploying AutoFDO to production (e.g. iterative compilation and stale profile) that are not covered by [10].

Shen et. al proposed a framework [26] to migrate an old profile to a new version of source code. They use different metrics to check if some part of the old profile is reusable for the new source, and selectively instrument the new source to only collect profiles that needs to be recollected. Their approach mainly focuses on offline profiling, while we focus on online profiling. Instead of recollecting profiles, we reuse the entire stale profile and tolerate discrepancies, which turns out to be effective for stable performance.

Wang et. al proposed a binary matching algorithm to map stale profile to the new binary[29]. The input of their algorithm is stale profile and both the old and new binary. With careful binary matching, their algorithm can accurately map stale profile to the new binary. In our experience, binary matching of basic blocks is hard for rapidly changing C++ binaries because inline decisions would be very different between two versions of the binary. Instead of taking the old binary as input, we simplified the process by just looking at the stale profile and new source.

Different methods have been proposed for online profiling. Some methods use PMU-based sampling to collect system-wide profiles. DCPI[6] is designed to profile continuously in production and provide insights on how to fine-tune performance problems. The Google-wide Profiler[25] extends the continuous sampling profiler to warehouse scale. This paper builds on top of the Google-wide Profiler to automatically guide compiler optimizations for the entire data center.

Some methods do not rely on the hardware PMU. For example, the Morph system[30] collects profiles via statistical sampling of the program counter on clock interrupts. Alternatively, Conte et al. proposed sampling the contents of the branch-prediction hardware using kernel-mode instructions to infer an edge profile[11]. In particular, the tags and target addresses stored in the branch target buffer (BTB) serve to identify an arc in an application, and the branch history stored by the branch predictor can be used to estimate each edge's weight. These methods do not provide accurate instruction frequency profiles, thus are less effective in driving compiler optimization.

Some methods build on ideas from both program instrumentation and statistical sampling. For example, Traub, Schechter, and Smith propose periodically inserting instrumentation code to capture a small and fixed number of the branch's executions[27]. A post-processing step is used to derive traditional edge profiles from the sampled branch biases collected. Their experiments show that the derived profiles show competitive performance gains when compared with using complete edge profiles to drive a superblock scheduler. Rather than dynamically modifying the binary, others have proposed a similar framework that performs code duplication and uses compiler-inserted counter-based sampling to switch between instrumented and non-instrumented code in a controlled, fine-grained manner[15]. Ammons, Ball, and Larus proposed instrumenting programs to read hardware performance counters[5]. By selecting where to reset and sample the counters, the authors are able to extract flow and context sensitive profiles. These profiles are not limited to simple frequency profiles. The authors show, for example, how to collect flow sensitive cache miss profiles from an application. All these methods involve instrumentation, which is too intrusive and not acceptable in a production runtime environment.

Some methods propose specialized hardware to facilitate PMU-based profiling. ProfileMe was proposed hardware support to allow accurate instruction-level sampling[13] for Alpha processors. Merten et al. also propose specialized hardware support for identifying program hot spots[23]. Unfortunately the hardware they propose is not available in today's commercial processors.

9. Conclusion

Combined with production automation and scale, AutoFDO has simplified the deployment of FDO in our datacenters to require only adding some compiler flags. These advances have led to an 8X increase in customer adoption and doubled the number of cycles covered by FDO.

Acknowledgments

We'd like to express our sincere thanks to the reviewers for taking the time to read the paper and provide feedback.

Thanks to Josh Kihm, Shari Brooks, Robert Hundt, Teresa Johnson, Than McIntosh, Darryl Gove, Diego Novillo and Luiz Andre Barroso, without whom AutoFDO cannot be as useful as of today.

References

[1] `http://wiki.dwarfstd.org/`.

[2] `https://investor.google.com/earnings/2014/`.

[3] `https://perf.wiki.kernel.org/`.

[4] Accepted for publication but not yet published.

[5] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[6] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, Nov. 1997.

[7] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[9] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming hardware event samples for fdo compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 42–52, New York, NY, USA, 2010. ACM.

[10] D. Chen, N. Vachharajani, R. Hundt, X. Li, S. Eranian, W. Chen, and W. Zheng. Taming hardware event samples for precise and versatile feedback directed optimizations. *Computers, IEEE Transactions on*, 62(2):376–389, Feb 2013.

[11] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox. Hardware-based profiling: An effective technique for profile-driven optimization. In *Int'l Journal of Parallel Programming*. Citeseer, 1996.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[13] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302. IEEE Computer Society, 1997.

[14] M. J. Eager. Introduction to the dwarf debugging format. *Group*, 2007.

[15] N. Gloy, Z. Wang, C. Zhang, B. Chen, and M. Smith. Profile-based optimization with statistical profiles. *Harvard University, Cambridge, Massachusetts*, 1997.

[16] D. Gove and L. Spracklen. In *SPEC 2006 Workshop*, 2006.

[17] T. Johnson and X. D. Li. Thinlto: A fine-grained demand-driven infrastructure. `http://llvm.org/devmtg/2015-04/slides/ThinLTO_EuroLLVM2015.pdf`.

[18] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[19] R. Levin, I. Newman, and G. Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'08, pages 291–304, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] X. D. Li, R. Ashok, and R. Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2010.

[21] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pages 330–339, 2010.

[22] A. Mericas, N. Peleg, L. Pesantez, S. Purushotham, P. Oehler, C. Anderson, B. King-Smith, M. Anand, J. Arnold, B. Rogers, L. Maurice, and K. Vu. Ibm power8 performance features and evaluation. *IBM Journal of Research and Development*, 59(1): 6:1–6:10, Jan 2015.

[23] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W.-m. W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ACM SIGARCH Computer Architecture News*, volume 27, pages 136–147. IEEE Computer Society, 1999.

[24] R. Rajwar, P. Lachner, L. Knauth, and K. Lai. Processor with last branch record register storing transaction indicator, July 2 2013. URL `https://www.google.com/patents/US8479053`. US Patent 8,479,053.

[25] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.

[26] X. Shen, Y. Ding, B. Wu, and M. Zhou. Profmig: A framework for flexible migration of program profiles across software versions. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society.

[27] O. Traub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling. *Unpublished technical report, Department of Electrical Engineering and Computer Science, Hardward University, Cambridge, Massachusetts*, 2000.

[28] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[29] Z. Wang, K. Pierce, and S. McFarling. Bmat - a binary matching tool for stale profile propagation. *J. Instruction-Level Parallelism*, 2, 2000.

[30] X. Zhang, Z. Wang, N. C. Gloy, J. B. Chen, and M. D. Smith. System support for automated profiling and optimization. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 15–26, 1997.