Reducing the Overhead of Exact Profiling by Reusing Affine Variables

Leon Frenot
ENS Lyon
France
leon.frenot@ens-lyon.fr

Fernando Magno Quintão Pereira
Federal University of Minas Gerais
Brazil
fernando@dcc.ufmg.br

Abstract

An exact profiler inserts counters in a program to record how many times each edge of that program's control-flow graph has been traversed during an execution of it. It is common practice to instrument only edges in the complement of a minimum spanning tree of the program's control-flow graph, following the algorithm proposed by Knuth and Stevenson in 1973. Yet, even with this optimization, the overhead of exact profiling is high. As a consequence, mainstream profile-guided code optimizers resort to sampling, i.e., approximate, profiling, instead of exact frequency counts. This paper introduces a technique to reduce the overhead of exact profiling. We show that it is possible to use the values of variables incremented by constant steps within loops—henceforth called SESE counters—as a replacement for some profiling counters. Such affine variables are common, for they include the induction variable of typical loops. This technique, although simple, is effective. We have implemented it in the LLVM compilation infrastructure. Standard Knuth-Stevenson instrumentation increases the running time of the 135 programs in the LLVM test suite from 648 seconds to 817. The optimization suggested in this paper brings this time down to 738 seconds. In the 949 Jotai programs, standard instrumentation increases the number of processed x86 instructions from 2.96 billion to 3.34 billion, whereas the proposed technique causes 3.07 billion instructions to be fetched.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: Profiling, Overhead, Compiler

ACM Reference Format:
Leon Frenot and Fernando Magno Quintão Pereira. 2024. Reducing the Overhead of Exact Profiling by Reusing Affine Variables. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3640537.3641569

Permission to make digital or hard copies of all or part 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 components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CC '24, March 2–3, 2024, Edinburgh, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641569

1 Introduction

Profiling is one of the most important enablers of compiler optimizations. As an example of its importance, profile-guided optimizations have delivered speedups of 30% or more on top of the last optimization level of mainstream compilers such as `gcc` or `clang` [9, 20, 23]. The literature groups profiling techniques into two categories: *sampling-based* and *instrumentation-based* [7]. The latter transforms the program with code to initialize, increment and store counters. The former does not transform code; rather, the profiler periodically reads the value of hardware counters (often the program counter), to estimate the execution frequency of program points. Since the early 2000's, sampling-based profiling can be performed via hardware support with virtually no overhead [32]. As an example, Shye et al. [27] would sample the Itanium's branch trace buffer (BTB) to incur in less than 1% of profiling overhead while obtaining an accuracy level of more than 88%. Instrumentation-based profiling, in turn, still has overhead, even considering industrial-quality implementations, such as the one available in LLVM. When building profile data via instrumentation for the LLVM test suite, for instance, we observe an absolute slowdown of 13.1% on the LLVM test suite (the geomean of slowdowns is 1.3%).

The Importance of Exact Profiling. In spite of its overhead, instrumentation-based profiling is important. This importance comes from two reasons: the nature of applications and the capabilities of the hardware. Firstly, in contrast to sampling, instrumentation-based profiling is *exact*: it reports exactly how often each edge of a program's control-flow graph has been traversed during execution. Some applications require this precision. As an example, Soares et al. [28, 29] have used an exact profiler to show that some programs contained time-based side channels, although they were linearized by code-strengthening tools. This extra precision is also desirable when performance is critical. For instance, Chen et al. [9] reports that performance improvements due to sampling-based profile data was 15% inferior to optimizations guided by instrumentation. For this reason, Meta's `HHVM` is still optimized with instrumentation-based

profile data [18]. As a third example, Xu et al. [34] are interested in counting *copies*—pairs of loads and stores—instead of execution points. Xu et al.'s profiler is based on code instrumentation.

Instrumentation-based profiling is also necessary due to hardware capabilities, because not every architecture provides counters. Thus, instrumentation-based profiling is still common in non-Intel systems or in virtualized environments. To emphasize this last point, we quote Du et al.: "*For applications executing in a public cloud, running a [sampling]-based profiler directly in a guest does not result in useful output, because, as far as we know, none of the current VMMs [Virtual Machine Monitors] expose the PMU [Performance-Monitoring Unit] programming interfaces properly to a guest*" [12]. Therefore, techniques that reduce the overhead of exact profiling are still in demand.

The Contribution of This Work. This paper presents a technique to reduce the overhead of instrumentation-based profiling. This idea consists of identifying program variables whose values can be used to replace counters within loops. Henceforth, we shall call these variables *SESE Counters*, for they count how often Single-Entry, Single-Exit (SESE)[1] regions within loops are visited during program execution. Knuth and Stevenson [16] have demonstrated that fully precise profile data can be reconstructed from counters in the complement of the minimum spanning tree of the program's control-flow graph. Henceforth, we shall call KS counter the instrumentation inserted via Knuth and Stevenson's heuristics. The techniques presented in this paper can eliminate some KS counters in loops. Because these counters occur in loops, their elimination tends to be profitable. We summarize the contributions derived from these ideas as follows:

- **Design:** Section 3 introduces the notion of SESE counters, and explains how to use their values to replace some counters created by Knuth and Stevenson's profiler within loops.
- **Prevalence:** Section 4.2 shows that SESE counters are common in programs. Considering 5,535 loops, taken from 949 programs, we observe the occurrence of 5,944 SESE counters.
- **Performance:** Section 4.3 provides empirical evidence that the replacement of KS counters with SESE counters improves the performance of profiled programs. Considering the 135 programs in the LLVM test suite, our technique reduces the average slowdown of instrumentation from 26% (817sec/648sec) to 13.8% (738sec/648sec). Figure 13, on Page 9, provides a complete overview of these results.

[1]SESE regions is a concept present in the work of Johnson et al. [15]. We shall revisit this notion in Section 2.

2 Exact Profiling

We shall use the program in Figure 1 to introduce the key concepts discussed in this paper. Figure 1 shows a C function, `count_zeros`, that counts the occurrences of zero values within an array. Notice that the exact semantics of this program is not important to this presentation. Nevertheless, to draw the reader's attention, we ask them the following question: *What is the minimum number of counters that must be inserted in* `count_zeros` *so that we can know how many times each part of it was traversed during its execution*?

```
00 int count_zeros(int v[], unsigned N) {
01   int sum = 0;
02   for (int i = 0; i < N; i++) {
03     if (!v[i]) {
04       sum++;
05     }
06   }
07   return sum;
08 }
```

Figure 1. A function, implemented in C, that counts the number of zeros within an array.

The Beautiful Trick: Minimum Spanning Trees. In 1973, Knuth and Stevenson [16] showed how to find the minimum number of counters necessary to instrument a program. In that case, Knuth and Stevenson [16] were logging how often each edge of a program's control-flow graph (CFG) was traversed during program execution[2]. To this end, Knuth and Stevenson demonstrated that it suffices to instrument the complement of the minimum spanning tree of the program's CFG. Example 2.1 illustrates these ideas.

Example 2.1. Figure 2 shows the control-flow graph of the program in Figure 1. This CFG is in static single-assignment form; hence, it uses the phi-functions introduced by Rosen et al. [26] to select the right assignments at blocks where different program flows converge. Figure 2 uses dashed edges to mark a minimum spanning tree of the CFG. The complement of this tree appears as solid gray edges.

An exact profile of function `count_zeros` can be produced by instrumenting only the gray edges in Figure 2. Knuth and Stevenson have shown how to reconstruct a complete profile from frequency counts for these two edges. This result follows from Kirchhoff's First Law of Flow Conservation: "*the sum of flows arriving into a junction point equals the sum of flows leaving that point.*" Example 2.2 shows how this reconstruction process works.

[2]Knuth and Stevenson's formalization use "program flow charts". The notion of a control-flow graph was already in place at that time [1]; however, it had yet to acquire the popularity that it enjoys today.

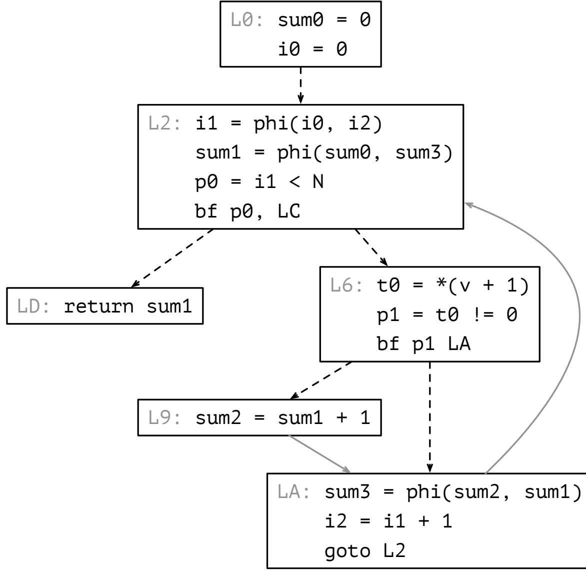

Figure 2. The control-flow graph of the program in Fig. 1. Dashed edges mark a minimum spanning tree. Gray edges are the complement of that tree.

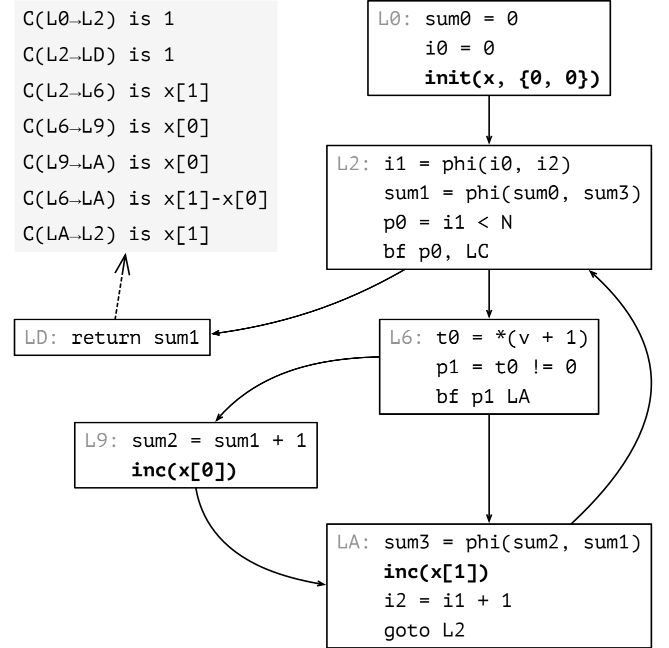

Figure 3. The instrumented version of the CFG in Figure 2. Instrumentation appears in boldface. The gray table shows how to compute full profile data at the return point.

Example 2.2. Figure 3 shows the instrumented control-flow graph of function `count_zeros` (Figure 1). To simplify the figure, we use two pseudo-instructions to represent instrumentation. Function `init` initializes a vector of counters, and function `inc` increments particular positions of that vector. The gray table in Figure 3 shows how a full program profile can be reconstructed from these two counters.

Knuth and Stevenson's result is optimal: full profile information cannot be reconstructed if any KS counter is omitted. However, it is possible to replace some of these counters with variables already present in the program. We call these variables *SESE Counters*—a name yet to be explained in Section 3. Example 2.3 concludes this section showing how program variables can replace Knuth and Stevenson's counters.

Example 2.3. In Figure 3, variable `sum1` holds, at execution's end, the same value as counter `x[0]`. Similarly, variable `i1` is equivalent to counter `x[1]`. Therefore, the five last entries in the table in Figure 3 could be reconstructed from these variables as follows: `C(L2 → L6) = i1`; `C(L6 → L9) = sum1`; `C(L9 → LA) = sum1`; `C(L6 → LA) = i1 − sum1`; and `C(LA → L2) = i1`. Indeed, a full profile of function `count_zeros` can be produced without any instrumentation.

3 Single-Entry Single-Exit Counters

The goal of this section is to define *SESE Counters*, program variables that can replace KS counters. To this end, this section revisits classic definitions due to Johnson et al. [15] and Havlak [13]. Thus, only Definition 3.5 in this section is original; the other concepts come from previous work. These definitions rely on two standard notions: control-flow graphs and dominance. A *control-flow graph* (CFG) is a directed graph G with a START node with no incoming edge and an END node with no outgoing edge. Vertices represent the program's *basic blocks*, and edges represent possible program flows. There is a path from START to any node $\ell_v \in G$, and there is a path from any node $\ell_v \in G$ to END. Given two nodes $\ell_u, \ell_v \in G$, ℓ_u *dominates* ℓ_v if, and only if, every path from START to ℓ_v contains ℓ_u. Analogously, ℓ_v *post-dominates* ℓ_u if, and only if, every path from ℓ_u to END contains ℓ_v. From these concepts, Definition 3.1 revisits *Reducible Loops*[3].

Definition 3.1 (Reducible Loop [13])**.** Given a CFG G, a reducible loop $L \subseteq G$ is a strongly-connected induced subgraph of G having a vertex ℓ_h (i.e., $\ell_h \in L$) such that ℓ_h dominates every node $\ell_v \in L$. Consequently, for any vertices $\ell_u \in G, \ell_u \notin L$ and $\ell_v \in L$, any path from ℓ_u to ℓ_v contains ℓ_h.

Reducible loops are *Single-Entry* regions of a control-flow graph. If, in addition to a single-entry node, every path leaving the loop passes through a common exit point, then the loop is also *Single-Exit*. The notion of Single-Entry, Single-Exit (SESE) regions applies to subsets of the CFG, even when they do not form loops, as Johnson et al. [15] have defined:

Definition 3.2 (SESE Region [15])**.** An ordered pair (ℓ_u, ℓ_v) formed by two blocks ℓ_v and ℓ_u of a CFG determines a *Single-Entry, Single-Exit (SESE) Region*, if, and only if:

[3]Reducible loops are called *natural loops* in some textbooks, such as Appel and Palsberg [3]'s.

1. ℓ_u dominates ℓ_v,
2. ℓ_v post-dominates ℓ_u,
3. If a cycle contains ℓ_u, it contains ℓ_v and vice versa.

The nodes ℓ_u and ℓ_v are said to be *Control Equivalent*.

Example 3.3. Blocks L6 and LA in Figure 3 form a SESE region. However, blocks L6 and L9 do not. To see why, notice that the cycle (L2, L6, LA, L2) does not contain block L9, although it contains L6.

The concept of a SESE counter emerges from the intersection of the notions of SESE regions, already seen in Definition 3.2, and the classic notion of a *Data-Dependence Graph*, which is standard in compiler textbooks. For self-consistency, this paper restates the definition from Cooper and Torczon [10], considering programs in static single-assignment form:

Definition 3.4 (Data-Dependence Graph)**.** A Data-Dependence Graph $D = (V_d, E_d)$ is a graph formed by the values defined and used in an SSA-form program P, such that, for each instruction $v_0 = op(v_1, \ldots, v_n) \in P$:

- $\{v_0, v_1, \ldots, v_n\} \subseteq V_d$;
- $v_i \longrightarrow v_0 \in E_d$, for every $v_i \in \{v_1, \ldots, v_n\}$.

Let $G = (V_g, E_g)$ be the control-flow graph of P, and let $G' = (V'_g, E'_g)$ be an induced subgraph of G. The dependence graph formed only by instructions in V'_g is called the dependence graph *induced* by G'.

Given a directed graph G, a *circuit* in G is a path in which source and destination are the same vertex. SESE counters are circuits in the dependence graph involving phi-functions and increments, such that any pair of increments forms a SESE region—a fact that implies that all these SESE regions are equivalent, in case the circuit contains more than two increments. Circuits in data-dependence graphs of SSA form programs necessarily involve phi-functions at loop headers[4]. Thus, SESE counters can be represented by these phi-functions, as Definition 3.5 formalizes:

Definition 3.5 (SESE Counter)**.** Let L be a reducible loop with header h within a CFG G. A phi-function $v_0 = \phi(\ldots)$ in h is a SESE counter if, and only if, any circuit $(v_0, v_1, \ldots, v_n, v_0)$ in the dependence graph D induced by L is such that:

1. For any v_i, $1 \leq i \leq n$, either v_i is defined by an affine expression $v_i = u + c$ or by a phi-function $v_i = \phi(\ldots)$.
2. If $v_i = u_i + c_i$ and $v_j = u_j + c_j$ are distinct instructions at $\ell_i \in L$ and $\ell_j \in L$, then (ℓ_i, ℓ_j) forms a SESE region.

Example 3.6. The program in Figure 1 contains two SESE Counters. The CFG in Figure 2 shows the two corresponding phi-functions: $\texttt{i1} = \phi(\texttt{i0}, \texttt{i2})$ and $\texttt{sum1} = \phi(\texttt{sum0}, \texttt{sum3})$. Figure 4 shows the induced dependence graphs that contain these two phi-functions. These graphs are induced by blocks L2, L6, L9 and LA in Figure 2

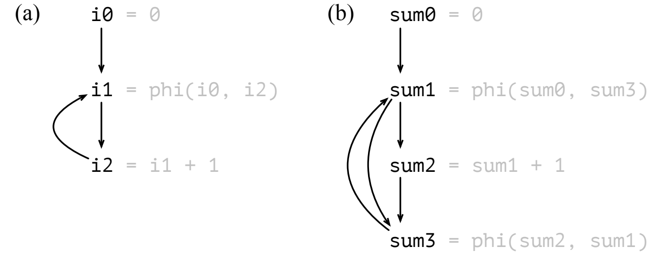

Figure 4. (a) The SESE Counter associated with instruction $\texttt{i1} = \phi(\texttt{i0}, \texttt{i2})$ in Figure 2. (b) The SESE Counter associated with instruction $\texttt{sum1} = \phi(\texttt{sum0}, \texttt{sum3})$.

SESE counters must be associated with at least one increment (see Theorem 3.9). The two SESE Counters in Example 3.6 are, each, associated with exactly one increment. However, SESE counters can be associated with multiple increments, as long as they are all control equivalent (as per Definition 3.2). Therefore, if a SESE counter is associated with multiple increments, they must belong into the same circuit. From this fact, we arrive at the essential property of SESE counters: once the loop that contains a SESE counter terminates, the value stored in this counter holds enough information to infer how often any edge in the circuit was traversed. Example 3.7 clarifies this fact showing a circuit that is not a SESE counter.

Example 3.7. Function `nonSESE`, in Figure 5 returns the same value as Function `count_zeros` in Figure 1. However, whereas variable `sum` is a SESE counter in Figure 1, this is not the case in Figure 5. To see why, Figure 6(a) shows the CFG of function `nonSESE`. Figure 6(a) shows the dependence graph induced by the CFG's loop that contains $\texttt{sum1} = \phi(\texttt{sum0}, \texttt{sum4})$. This dependence graph contains two increments. These increments define variables `sum2` and `sum3`, at blocks L6 and LA. These blocks do not form a SESE region: the cycle (L2, L6, LB, L2) does not contain LA. Consequently, `sum1` can be incremented along two different circuits, which Figure 6(b) shows. Hence, once the loop terminates, the value stored in `sum1` cannot be used to recover the frequency count of CFG edges along any of these circuits.

3.1 Replacing KS Counters with SESE Counters

A SESE counter c_s replaces one KS counter c_k at every exit point of the loop L that contains c_s. This observation follows from Theorem 3.11. If multiple SESE counters share the same circuit, then only one of them is necessary to replace the KS counter. If $v_s = \phi(v_0, \ldots, v_n)$ is a SESE counter at header h in loop L, then we build a frequency counter for L using the two steps below, which Example 3.8 illustrates:

[4]This result follows from Cytron et al. [11]'s iterated dominance criterion, which causes a phi-function to exist at any point reached by two definitions of a variable name. The header is reached by the definition outside the loop, and by at least one definition inside the loop (the increment).

```
00 int nonSESE(int v[], unsigned N) {
01   int sum = 0, i = 0;
02   for (i = 0; i < N; i++) {
03     sum++;
04     if (!v[i]) { sum++; }
05   }
06   return sum - i;
07 }
```

Figure 5. A new—rather artificial—implementation of the program in Figure 1.

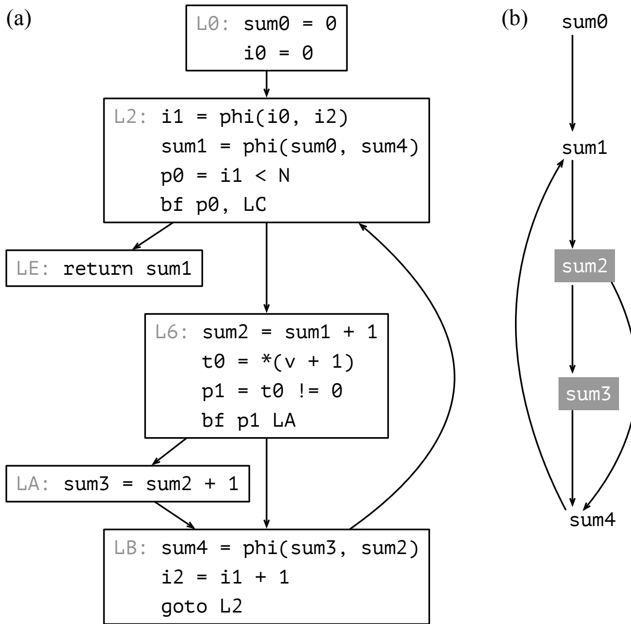

Figure 6. (a) The control-flow graph of the program in Figure 5. (b) The dependence graph that contains sum1 = ϕ(sum0, sum4). This phi-function is not a SESE counter.

1. Save the incoming value $v_{\texttt{init}}$ of v at every block that reaches h.
2. At L's exit blocks, store the value $(v - v_{\texttt{init}})/\texttt{step}$, where `step` is the sum of constant increments along the circuit that contains v.

Example 3.8. Figure 7(a) shows a program that finds the first position, within an array `v` that contains a variable `q`, up to position `N`. Figure 7(b) shows the same program, this time instrumented with code to store the value of the SESE counter associated with variable `p`. Notice that `p` is a SESE counter, in spite of having a pointer type. This pointer contains enough information to infer the number of iterations of the loop, once this loop terminates.

```
00 long* first_occurrence(long v[], long q, int N) {
01   long* p = v;
02   v[N] = q;
03   while (*p != q) { p++; }
04   return p;
05 }
```
(a)

```
00 long* first_occurrence(long v[], long q, int N) {
01   long* p = v;
02   v[N] = q;
03   long* init = p;
04   while (*p != q) { p++; }
05   long c = ((size_t)p-(size_t)init)/sizeof(long*);
06   log("first_occurrence, Line 03", c);
07   return p;
08 }
```
(b)

Figure 7. (a) Program that finds address of first occurrence of value within array. (b) Version of `first_occurrence` instrumented with a SESE counter.

3.2 Properties of SESE Counters

This section discusses some properties of SESE counters, concluding with Theorem 3.11, which states that a SESE counter contains enough information to replace one KS counter in an instrumented program.

Theorem 3.9. *A SESE counter is associated with at least one affine increment.*

Item 1 of Definition 3.5 states that the circuit forming a SESE counter contains either phi-functions or increments. Increments create new definitions of variables; phi-functions emerge to ensure that these definitions meet the SSA-form property. Thus, without increments, there would be no need for phi-functions, and, consequently, there would exist no phi-function at all in the loop header. □

Theorem 3.10. *If $v_s = \phi(\ldots)$ is a SESE counter in a loop L, then the frequency f_s of the SESE regions that contain the increments associated with v_s is a function of v_s.*

Let `init` be the value of v_s before entering the loop; `last` be the value of v_s after exiting the loop; and `step` be the sum of constant increments in the SESE circuit. At loop exit, the frequency $f_s = (\texttt{last} - \texttt{init})/\texttt{step}$. If the circuit runs 0 times, then $f_s = 0$. But `init` = `last`, and the SESE counter is also zero. Assume that the theorem holds up to $n - 1$ iterations of the circuit. Let $\texttt{last}^{n-1}$ be the value of v_s at that iteration. We have that $f_s = (\texttt{last}^{n-1} - \texttt{init})/\texttt{step}$ by the induction hypothesis. If the circuit iterates once more, then $f_s + 1 = ((\texttt{last}^{n-1} + \texttt{step}) - \texttt{init})/\texttt{step}$. Thus $f_s + 1 = (\texttt{last}^{n} - \texttt{init})/\texttt{step}$. □

A SESE counter replaces one KS counter in an optimal instrumentation of a program. Theorem 3.11 explains why it is possible to establish a function from SESE counters to KS counters. However, the opposite is not true: it is possible that a KS counter finds no SESE counter to replace it.

Theorem 3.11. *A SESE counter c_s replaces exactly one KS counter c_k within the loop L that contains c_s.*

In what follows, we write $c_s \equiv c_k$ if the value of c_k is the frequency f_s of Theorem 3.10. We let L' be the induced subgraph of L that contains the increments that characterize c_s. There must be at least one increment, as per Theorem 3.9. Let $(\ell_u, \ell_v) \in L'$ be the SESE region that contains these increments. L' must contain at least one KS counter, for a cycle contains at least one edge in the complement of any minimum spanning tree. Let this counter be c_k. There are two cases to consider. Firstly (Fig. 8-a), if there exists c_k, such that $c_k \equiv c_s$, then c_k and the increments of c_s are control-equivalent. In this case, c_k can be replaced with f_s (see Theorem 3.10). Otherwise (Fig. 8-b/c), f_s is computed as a function of multiple KS counters, e.g., $f_s = c_1 \pm c_2 \pm \ldots \pm c_n$. Exactly one of these counters belong to L' (more counters would contradict the optimality of Knuth and Stevenson's result; no counter would make it impossible to know the frequency of (ℓ_u, ℓ_v) within L'). Let $c_k = c_i, 1 \leq i \leq n$. We replace c_k with f_s in $c_1 \pm c_2 \pm \ldots \pm c_n$. □

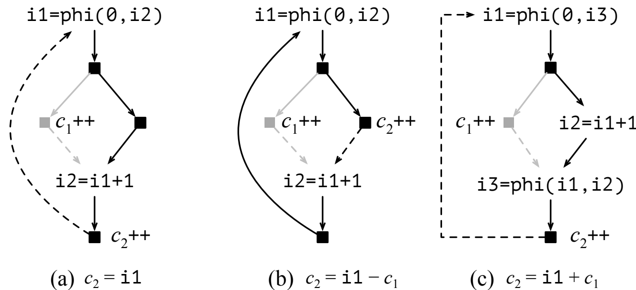

Figure 8. Cases mentioned in the proof of Theorem 3.11; c_1 and c_2, on the dashed edges, are KS counters; `i1` is a SESE counter. L' is made of dark edges.

4 Experimental Evaluation

This section evaluates the following research questions:

RQ1: What is the overhead, in terms of time and space, of exact profiling done via Knuth and Stevenson [16]'s instrumentation?

RQ2: How prevalent are SESE counters in C/C++ programs present in either typical benchmark suites or mined from open-source repositories?

RQ3: How much overhead can be saved on top of the base cost of Knuth and Stevenson [16]'s instrumentation, via the techniques introduced in this paper?

Hardware. Experiments evaluated in this section were performed on an AMD Ryzen 7 4800HS with a Radeon Graphics CPU, featuring 7.5GB of RAM and two 256 KiB L1 caches.

Software. The experimental setup runs on Linux Ubuntu 22.04.2 LTS. Instrumentation is implemented in LLVM 17.0.0 (stable release). To count instructions, we use `CFGGrind` [25][5], a plugin for `Valgrind` 3.21.0 (commit from June 2nd, 2023). Instrumentation is not thread safe (LLVM's default)[6], for it uses non-atomic increments; thus, counters might contain inaccurate data under thread contention.

Benchmarks. This paper uses two collections of benchmarks, which we describe below. Numbers in this section come from observations of the runtime behavior of these programs compiled with `clang` at the `-O1` optimization level. We chose this optimization level because it is still able to hoist some KS counters outside loops, without changing too much the control-flow graph of programs. Higher optimization levels make it very hard to map frequency counts from the LLVM intermediate representation back to source code, due to optimizations like inlining, loop unrolling, and vectorization with replication of the innermost loop. The benchmarks evaluated in this paper come from the following collections:

Jotai: a collection formed by 950 programs from the `Jotai`[7] collection. Each program contains a single function, which does not call other functions. Each function contains at least one loop. This codebase has been mined from open-source repositories. Inputs for the benchmarks were produced via fuzzing. In spite of fuzzing, none of these programs runs into undefined behavior, as far as `Frama-C` [6] is able to detect.

LLVM-TS: The LLVM test suite: a collection of 134 programs available as part of the LLVM compilation infrastructure.

4.1 RQ1 – The Baseline Overhead

The core instrumentation algorithm proposed by Knuth and Stevenson [16], here implemented following Ball and Larus [4]'s specification, imposes an overhead onto profiled programs: the cost of loading, incrementing and storing counters. This section gauges this overhead along two dimensions: the extra *time* necessary to run instrumented programs, and the extra *space* that counters occupy in the binary code.

Methodology–Time. We report time overhead in two ways. First, we use CFGGrind to count the total number of

[5]Available at https://github.com/rimsa/CFGgrind

[6]See https://clang.llvm.org/docs/UsersManual.html#cmdoption-fprofile-update (visited on January 3rd, 2024.)

[7]Taken from https://github.com/lac-dcc/jotai-benchmarks on June 2023.

instructions fetched by the processor during the execution of non-instrumented and instrumented programs. Second, we measure the running time of original and instrumented programs. For each of these two experiments we use different benchmarks. When counting fetched instructions, we use `Jotai`. We restrict this analysis to `Jotai` programs because it is easy to filter instructions per function via `CFGGrind`: each benchmark consists of a single function that does not call library functions. In other words, every instruction from call to return of the benchmark function is "visible" [2] to `Valgrind+CFGGrind`. When timing programs, we use only `LLVM-TS` because these programs are larger, and run for a longer time; hence, time variance is smaller than in `Jotai`.

Methodology–Space. We measure "static" and "dynamic" space. Static space is measured as the size, in bytes, that `clang` produced for each program, at the `-O1` optimization level. Dynamic space is measured via `CFGGrind`, as number of different x86 instructions visited during the execution of programs. In the former case, we consider the two benchmark collections: `Jotai` and `LLVM-TS`. In the latter, only `Jotai`, for the reasons explained in the previous paragraph.

Discussion–Time. The right boxes in Figures 9 and 10 compare the running time of original and instrumented programs. Instrumentation happens either via Knuth and Stevenson (KS) counters, or via SESE counters. However, this section focuses on the impact of standard (KS-based) instrumentation. Section 4.3 will discuss the overhead of SESE counters. On average (Median in Figure 10–bottom right box), KS counters increase the number of instructions fetched during the execution of `Jotai` benchmarks by 18.9%. This number has no variance: we are counting the discrete number of instructions fetched during the execution of programs. In the worst case across all the 949 `Jotai` benchmarks, the number of fetched instructions increased by 119%. Considering running time, the `LLVM-TS` programs instrumented with KS counters run for approximately 817 seconds, compared to 648 seconds without any instrumentation. The median (Median of running time in Figure 9–bottom right box) slowdown caused by instrumentation was 1.3%. However, although this number seems low in principle, pathological cases are possible. For instance, one of the benchmarks in the LLVM test suite, `Ptrdist/ks`, runs for 1.13 seconds without instrumentation, whereas its instrumented version (using KS counters only) runs for 6.5 seconds (average of three samples).

Discussion–Space. Figures 9 and 10 compare original and instrumented programs in terms of binary size. In regards to the LLVM test suite, KS counters increase the size of programs by 15.3% (Median in Figure 9–bottom left box). The growth observed in the `Jotai` program is larger: the median number of x86 instructions fetched for the original benchmark functions is 37, versus 57 once KS counters are in place: a median growth of 43.9%. Nevertheless, we remind

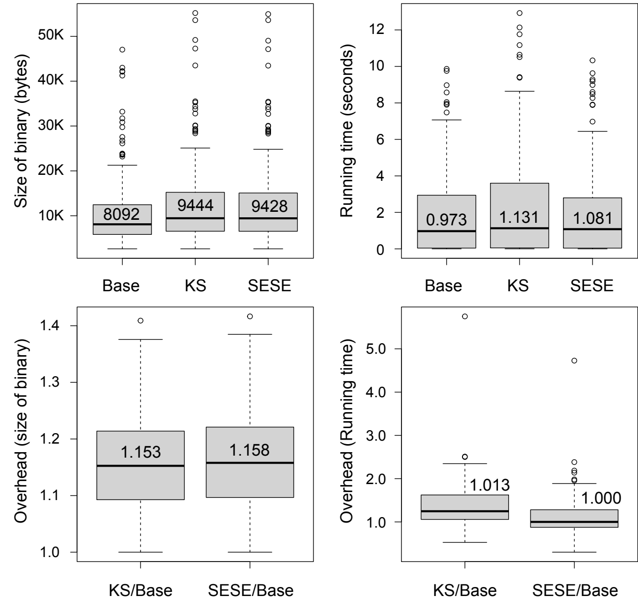

Figure 9. The overhead of exact profiling on 135 programs in the LLVM test suite. The two boxes on the left refer to the size of the programs, in number of LLVM instructions. The two boxes on the right refer to the running time of programs, in seconds. The upper boxes show absolute numbers; the lower boxes show ratios. Numbers within boxes are medians.

the reader that `Jotai` functions are much smaller than the benchmarks in the LLVM test suite: in this case, each benchmark consists of a single function, that does not call other functions. Each function does come with a driver that generates inputs for it; however, Figure 10 does not consider the counters inserted in the driver.

4.2 RQ2 – The Prevalence of SESE Counters

The replacement of KS Counters with SESE counters is meaningful inasmuch as SESE counters are common structures in code. Intuitively, such should be the case, for SESE counters include the basic induction variables that control the trip count of loops. Indeed, basic induction variables are the main source of SESE counters, as this section reports.

Methodology. We report the number of SESE counters in the `Jotai` collection. We restrict this analysis to `Jotai` for convenience: in contrast to the programs in `LLVM-TS`, each `Jotai` benchmark is formed by a single C file; thus, it is simple to report cumulative data per benchmark. To give the reader some perspective on the number of SESE counters, this section also reports the number of reducible loops per benchmark (see Definition 3.1). However, notice that these two quantities—number of reducible loops and number of SESE counters—are not directly comparable. A SESE counter is a program variable, whose semantics (initialization and updates) must be implemented via several

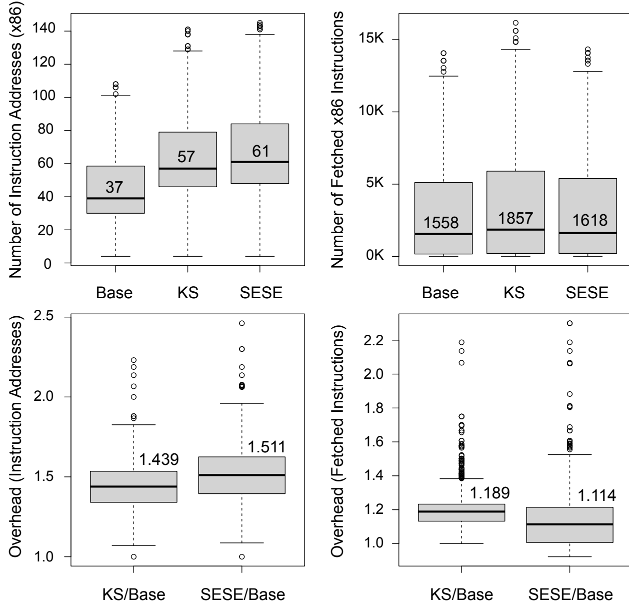

Figure 10. Analysis of x86 instructions visited and fetched during the execution of 949 `Jotai` programs, as reported by `CFGGrind`. The two boxes on the left refer to the number of *instruction addresses* visited during execution. The more instruction addresses are visited, the larger the path of executed code. The two boxes on the right refer to the total number of instructions fetched during the execution of programs. The more instructions are fetched, the longer the program runs. Numbers in boxplots are medians.

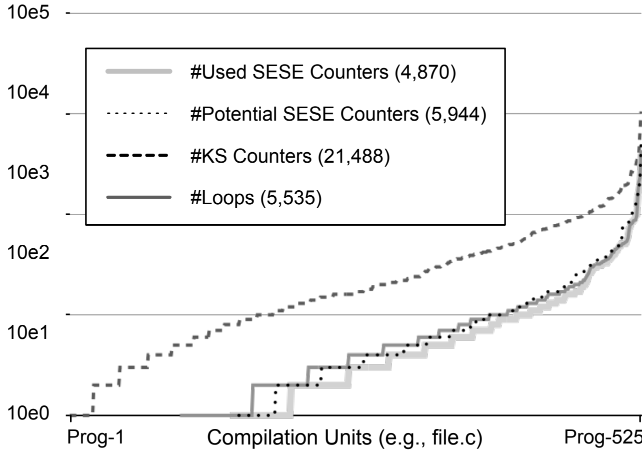

Figure 11. Cumulative distribution of number of loops and counters found across 525 source files in the LLVM test suite.

LLVM instructions; a loop is a control-flow construct, who implementation requires also multiple LLVM instructions.

Discussion. Figure 11 shows four cumulative distributions: number of loops; number of KS counters; number of possible SESE counters; and number of SESE counters used for the instrumentation. We remind the reader that the number of KS counters is the number of edges in the complement of the minimum spanning tree of a program’s CFG. The relation between number of loops and number of SESE counters is clear. In general, we find one SESE counter per program loop. The Pearson Coefficient relating these two quantities is 0.98, indicating strong correlation. Nevertheless, a program might contain slightly more SESE counters than loops. For instance, the program in Figure 1 contains one loop and two SESE counters. However, not all these counters can be used to replace KS counters, for some of them might cover the same circuit within a control-flow graph. There are also loops that do not feature SESE counters, simply because they do not contain affine variables. Example 4.1 shows two instances of such loops taken from the same benchmark.

Example 4.1. Figure 12 shows code snippets of two functions taken from a C file in the LLVM test suite (`Ptrdist/ks/KS-1.c`). Each function contains an outermost loop whose induction variable forms up a SESE counter, and an innermost loop that has no SESE counters. Incidentally, this benchmark, `Ptrdist/ks` accounts for one of the heaviest slowdowns that we have observed. The original program runs for 1.13 sec; the program with KS counters takes 6.49 secs; and the program with SESE counters take 5.34 secs.

```
 35 void ReadNetList(char *fname) {
 ..   ...
 53   for (net = 0; net < numNets; net++) {
 ..   ...
 66     while ((tok = strtok(NULL, " \t\n")) != NULL)
 ..     {...}
 77   }
 78 }
 ..
 82 void NetsToModules(void) {
 ..   ...
 91   for (net=0; net<numNets; net++) {
 92     for (modNode = nets[net]; modNode != NULL;
 ..          modNode = (*modNode).next) {...}
101   }
102 }
```

Figure 12. Examples of loops with SESE counters (outermost loops) and without them (innermost loops).

4.3 RQ3 – Overhead Reduction

SESE counters reduce the overhead of exact profiling, for they allow removing some instrumentation from programs. Therefore, once SESE counters are used to collect profile data, the overhead that remains after instrumentation is due to the counters left in the target program, plus the overhead

to store SESE counters at the exit point of loops. This section measures this overhead, contrasting it with the numbers that Section 4.1 reports for the cost of the standard instrumentation of Knuth and Stevenson. In this regards, this section adopts the same methodology seen in Section 4.1.

Discussion–Time. Figure 10 shows that SESE counters raise the number of instructions fetched during the execution of `Jotai` program by 11.4% (Median across 950 benchmarks–lower right box in Figure 10). Contrasting these numbers with those presented in Section 4.1, we see that SESE counters reduce the overhead of exact instrumentation from 18.9% to 11.4%. These numbers find similar counterparts once absolute running time is considered. As Figure 9 illustrates, SESE counters increase the median execution time of programs in the LLVM test suite from 0.973 seconds to 1.081 seconds, whereas KS counters alone increase it to 1.131 seconds. If we consider medians of ratios, then SESE counters bring virtually no overhead (Figure 9–lower right box). However, running time presents considerable variance. Thus, we believe that the experiments with `Jotai` benchmarks bring a better idea of the overhead of instrumentation, be it using KS counters only, or also using SESE counters.

Discussion–Space. Figure 10 compares the size of programs instrumented without and with SESE counters. When running the `Jotai` programs, 48,217 different x86 instructions are visited for the original (non-instrumented) codes; 66,960 instructions are visited for codes instrumented with standard KS counters, and 70,331 instructions are visited once SESE counters are considered. Overall, SESE counters increase the number of instructions visited per program by 51.1% (Median across 950 programs–lower left box in Figure 10), whereas standard instrumentation causes 43.9% more instructions to be visited. Therefore, programs instrumented with SESE counters are larger, due to the code to store frequencies. However, this code is moved outside loops. If we consider the size of binary executables, in bytes, as Figure 9 shows, then we observe that while programs instrumented with KS counters are 15.3% larger than the original codes (lower left box in Figure 9), programs instrumented with SESE counters are 15.8% larger.

4.4 Summary of Results

Figure 13 summarizes the data discussed in this section. A few conclusions can be drawn from this table. First, SESE counters reduce the overhead of Knuth and Stevenson's optimal instrumentation. Reduction is observed regardless of the metric: number of x86 instructions fetched or running time. Reduction is also observed regardless of the summarization methodology: absolute number or median.

On the other hand, SESE counters do not reduce code size. Rather, we observe a slight increase in size, when SESE counters are used to replace KS counters. This growth is very small, and it is due to the extra instructions necessary to compute the value of a KS counter from a SESE counter. In this case, some arithmetic operations are necessary to rematerialize the value of a KS counter, as explained in Section 3.1 (see Example 3.8). Notice that this growth is very small, and not always present. In particular, when experimenting with the LLVM test suite, we have observed that although the median size of executables grew (from 10,500 bytes with KS counters to 10,536 bytes with SESE counters), the sum of all the binaries did not (5,736 $\times 10^3$ bytes with KS counters vs 5,713 $\times 10^3$ bytes with SESE counters).

		`Jotai` (949 Programs)		LLVM Test Suite (135 Programs)	
		Size Different x86 instrs. fetched	Time x86 instrs. fetched	Size Bytes	Time seconds
Base	Total	48,217	2,965M	4,595K	648.11
	Med.	40	2,326	8,982	1.14
KS	Total	66,960	3,345M	5,736K	817.00
	Med.	59	2,784	10,500	1.43
SESE	Total	70,331	3,075M	5,713K	738.51
	Med.	62	2,449	10,536	1.33

Figure 13. Summary of data discussed in this section.

5 Related Work

This section explains how this paper improves on previous profiling techniques. As mentioned in Section 1, profiling techniques can be grouped into exact and approximate approaches. Although this paper concerns the former, this section also goes over the latter for completeness.

Profile-Guided Code Optimizations. The main usage of profile data is to support compilers to perform code optimizations. Several classic compiler optimizations, such as register allocation and inlining benefit from such data. Section 2 of Chang et al. [8]'s work describe several examples of these optimizations, and how they can be enhanced with a profiler's feedback. Nowadays, profile data is commonly used to enable inlining decisions in data-center applications [9]; to support more accurate branch prediction strategies [31]; to realign code, so to improve locality in the instruction cache [23]; and to support more informed prefetching decisions [19].

Approximate Profiling. Approximate profiling can be based on sampling or instrumentation. Sampling is the preferred choice in industrial-quality tools, such as AMD's `uProf`, Intel's `vTune`, GNU's `gprof`, Microsoft's `WPT`, or `HWPMC` (in the `FreeBSD 6.0` and above). Approximate profiling via instrumentation is typically done in just-in-time compilers. For instance, Sol et al. [30] explain how profiling is done in

Mozilla's `SpiderMonkey`: "*Each conditional branch is associated with a counter initially set to zero. If the interpreter finds a conditional branch during program interpretation, then it increments the counter. The process of checking and incrementing counters is called, in* `TraceMonkey`*'s jargon, the monitoring phase*". Notice that counters, in this case, are associated with branches—not with control-flow edges. Thus, an exact program profile cannot be reconstructed from these counters only. Nevertheless, there exists heuristics to estimate missing frequency counts in approximate profile data. Examples include techniques from Wu and Larus [33], Levin et al. [17] and He et al. [14]. These heuristics are not guaranteed to reconstruct an exact profile from approximate data.

Exact Profiling. The baseline instrumentation approach used in this paper is due to Knuth and Stevenson [16]. This algorithm has been firstly discovered by Nahapetian [21]; however, Nahapetian's work did not deal directly with programs. His formulation was more abstract; namely, finding the minimum number of measurements necessary to determine the flow in a graph governed by Kirchhoff's Conservation Law. Most of the modern formulation of exact profiling is due to Ball and Larus [4]. Ball and Larus extends Knuth and Stevenson [16]'s optimal algorithm to profile not only edges, but also paths within programs [5]. Still, optimality, be it in path, vertex or edge profiling, is bounded by the complement of the minimum spanning tree of the program's CFG, as originally proved by Knuth and Stevenson, following Nahapetian's work. As a consequence, Knuth and Stevenson's algorithm is nowadays part of mainstream compilation infrastructures, such as LLVM[8] or BOLT[9].

There exist approaches to reduce the amount of counters necessary to instrument programs in particular contexts. For instance, if users are interested in only parts of a program, then Ohmann et al. [22] has proposed heuristics to eliminate counters. Achieving minimality in this scenario, nevertheless, seems unlikely: Ohmann et al. have shown that such problem is NP-complete. Notice that Ohmann et al. worked on a related, but not equal, version of the profiling problem: they were interested in determining binary coverage, i.e.: either a CFG edge is traversed or not. In contrast to code profiling, binary coverage does not abide by Kirchhoff's Conservation Law. Thus, while Knuth and Stevenson's algorithm solves Ohmann et al.'s problem, the opposite is not true. Indeed, Probert [24] has demonstrated that for structured programs[10], it is possible to use less counters than Knuth and Stevenson's algorithm, if only coverage, but not a complete frequency count is desired.

[8]See PGOInstrumentation_8cpp_source.html (Aug 2nd 2023)
[9]See Line 333 of https://github.com/llvm/llvm-project/blob/main/bolt/lib/Passes/Instrumentation.cpp (Aug 2nd 2023)
[10]Probert's work precedes Johnson et al.'s by more than one decade; hence, it uses a definition of "Structured Program" based on a grammar of acceptable coding constructs, instead of relying on the notion of Single-Entry, Single-Exit regions, which is standard today.

Counters and Classic Compiler Optimizations. Except for that previous work related to code coverage [22, 24], we are not aware of other attempts to reduce the number of counters inserted by Knuth and Stevenson's algorithm. However, the combination of scalarization and classic compiler optimizations can deliver results similar to our technique. For instance, Ball and Larus [5] mentions that counters should be kept in registers throughout the execution of loops. Loads and stores are only necessary at entry and exit points of loops. In the LLVM instrumentation framework, this optimization is called *counter promotion*[11]. If the loop's trip count is symbolically known (for instance, the loop has a single affine induction variable with invariant bounds), then LLVM's scalar evolution is able to remove the increments from the promoted counter, storing its final value only once. As an example, LLVM, at the `-O1` optimization level (the setup evaluated in Section 4), is able to replace the counter in block `LA` in Figure 3 with a single store of value `N` in block `LD`. This optimization cannot remove the increments in block `L9`, nor the counter increments within the loop in Figure 7, as SESE counters do. Nevertheless, notice that LLVM is not replacing a counter with a program variable: the counter is inserted, and then classic compiler optimizations mitigate its overhead. More examples of such interactions, albeit concerning dynamic safety checks, not instrumentation, can be found in Section 3.6 of Yarahmadi and Rohou [35]'s work.

6 Conclusion

This paper has introduced a technique to lower the overhead of the optimal instrumentation proposed by Knuth and Stevenson in 1973. Said technique consists in replacing some of the counters that Knuth and Stevenson's approach requires with variables whose values are an affine function of such counters. Therefore, the proposed approach does not contradict Knuth and Stevenson's optimality result: an exact program profile still requires probing the complement of the minimum spanning tree of a control-flow graph. However, some of these probes can be replaced with program variables. Because these variables exist in loops, reusing them to avoid inserting counters in programs tends to be profitable. Experimental results performed on many different programs show that often it is possible to reduce to virtually zero the overhead of obtaining exact profiles.
Software The implementation of SESE counters used in this paper is available at https://github.com/lac-dcc/Nisse.

Acknowledgement

We thank Xinliang David Li (Google), Maksim Panchenco (Meta) and the CC reviewers for suggestions. Users `u/cxzuk` and `u/moon-chilled`, from `reddit/compilers`, helped revising a draft of this paper. This project is sponsored by FAPEMIG (APQ-00440-23) and CNPq (304441/2021-0).

[11]See InstrProfiling_8cpp_source.html (Aug 4th 2023)

References

[1] Frances E. Allen. 1970. Control Flow Analysis. *SIGPLAN Not.* 5, 7 (jul 1970), 1–19. https://doi.org/10.1145/390013.808479

[2] Andrei Rimsa Álvares, José Nelson Amaral, and Fernando Magno Quintão Pereira. 2021. Instruction visibility in SPEC CPU2017. *J. Comput. Lang.* 66 (2021), 101062. https://doi.org/10.1016/j.cola.2021.101062

[3] Andrew W. Appel and Jens Palsberg. 2003. *Modern Compiler Implementation in Java* (2nd ed.). Cambridge University Press, USA.

[4] Thomas Ball and James R. Larus. 1994. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.* 16, 4 (jul 1994), 1319–1360. https://doi.org/10.1145/183432.183527

[5] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *MICRO* (Paris, France). IEEE Computer Society, USA, 46–57.

[6] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. 2021. The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. *Commun. ACM* 64, 8 (jul 2021), 56–68. https://doi.org/10.1145/3470569

[7] Michael D. Bond and Kathryn S. McKinley. 2005. Continuous Path and Edge Profiling. In *MICRO* (Barcelona, Spain) *(MICRO 38)*. IEEE Computer Society, USA, 130–140. https://doi.org/10.1109/MICRO.2005.16

[8] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. 1991. Using Profile Information to Assist Classic Code Optimizations. *Softw. Pract. Exper.* 21, 12 (dec 1991), 1301–1321. https://doi.org/10.1002/spe.4380211204

[9] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *CGO* (Barcelona, Spain). Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2854038.2854044

[10] Keith D. Cooper and Linda Torczon. 2023. Chapter 4 - Intermediate Representations. In *Engineering a Compiler (Third Edition)* (third edition ed.), Keith D. Cooper and Linda Torczon (Eds.). Morgan Kaufmann, Philadelphia, 159–207. https://doi.org/10.1016/B978-0-12-815412-0.00010-3

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[12] Jiaqing Du, Nipun Sehrawat, and Willy Zwaenepoel. 2010. Performance Profiling in a Virtualized Environment. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 2.

[13] Paul Havlak. 1997. Nesting of Reducible and Irreducible Loops. *ACM Trans. Program. Lang. Syst.* 19, 4 (jul 1997), 557–567. https://doi.org/10.1145/262004.262005

[14] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. 2022. Profile Inference Revisited. *Proc. ACM Program. Lang.* 6, POPL, Article 52 (jan 2022), 24 pages. https://doi.org/10.1145/3498714

[15] Richard Johnson, David Pearson, and Keshav Pingali. 1994. The Program Structure Tree: Computing Control Regions in Linear Time. In *PLDI* (Orlando, Florida, USA). Association for Computing Machinery, New York, NY, USA, 171–185. https://doi.org/10.1145/178243.178258

[16] Donald E. Knuth and Francis R. Stevenson. 1973. Optimal Measurement Points for Program Frequency Counts. *BIT* 13, 3 (sep 1973), 313–322. https://doi.org/10.1007/BF01951942

[17] Roy Levin, Ilan Newman, and Gadi Haber. 2008. Complementing Missing and Inaccurate Profiling Using a Minimum Cost Circulation Algorithm. In *HiPEAC* (Göteborg, Sweden). Springer-Verlag, Berlin, Heidelberg, 291–304.

[18] Yuhao Li, Abhishek Gupta, Alex Yang, Peinan Chen, Joey Pinto, Brian Karrer, Mayank Pundir, Maximilian Balandat, Arun Kejariwal, and Benjamin Lee. 2023. HHVM Performance Optimization for Large Scale Web Services. In *ICPE* (Coimbra, Portugal). Association for Computing Machinery, New York, NY, USA, 137–148. https://doi.org/10.1145/3578244.3583720

[19] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. 2022. CRISP: Critical Slice Prefetching. In *ASPLOS* (Lausanne, Switzerland). Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/3503222.3507745

[20] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. 2021. VESPA: Static Profiling for Binary Optimization. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 144 (oct 2021), 28 pages. https://doi.org/10.1145/3485521

[21] Armen Nahapetian. 1973. Node Flows in Graphs with Conservative Flow. *Acta Inf.* 3, 1 (mar 1973), 37–41. https://doi.org/10.1007/BF00288650

[22] Peter Ohmann, David Bingham Brown, Naveen Neelakandan, Jeff Linderoth, and Ben Liblit. 2016. Optimizing Customized Program Coverage. In *ASE* (Singapore, Singapore). Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2970276.2970351

[23] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, New York, US, 2–14.

[24] R. L. Probert. 1982. Optimal Insertion of Software Probes in Well-Delimited Programs. *IEEE Trans. Softw. Eng.* 8, 1 (jan 1982), 34–42. https://doi.org/10.1109/TSE.1982.234772

[25] Andrei Rimsa, José Nelson Amaral, and Fernando M. Q. Pereira. 2021. Practical dynamic reconstruction of control flow graphs. *Softw. Pract. Exp.* 51, 2 (2021), 353–384. https://doi.org/10.1002/spe.2907

[26] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *POPL* (San Diego, California, USA). Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[27] Alex Shye, Matthew Iyer, Tipp Moseley, David Hodgdon, Dan Fay, Vijay Janapa Reddi, and Daniel A. Connors. 2005. Analyis of Path Profiling Information Generated with Performance Monitoring Hardware. In *INTERACT*. IEEE Computer Society, USA, 34–43. https://doi.org/10.1109/INTERACT.2005.3

[28] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. 2023. Side-Channel Elimination via Partial Control-Flow Linearization. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 13 (jun 2023), 43 pages. https://doi.org/10.1145/3594736

[29] Luigi Soares, Fernando Magno, and Quintão Pereira. 2021. Memory-Safe Elimination of Side Channels. In *CGO* (Virtual Event, Republic of Korea). IEEE Press, New York, US, 200–210. https://doi.org/10.1109/CGO51591.2021.9370305

[30] Rodrigo Sol, Christophe Guillon, Fernando Magno Quintão Pereira, and Mariza A. S. Bigonha. 2011. Dynamic Elimination of Overflow Tests in a Trace Compiler. In *Compiler Construction* (Saarbrücken, Germany). Springer-Verlag, Berlin, Heidelberg, 2–21.

[31] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: Profile-Guided Btb Replacement for Data Center Applications. In *ISCA* (New York, New York). Association for Computing Machinery, New York, NY, USA, 742–756. https://doi.org/10.1145/3470496.3527430

[32] Kapil Vaswani, Matthew J. Thazhuthaveetil, and Y. N. Srikant. 2005. A Programmable Hardware Path Profiler. In *CGO*. IEEE Computer Society, USA, 217–228. https://doi.org/10.1109/CGO.2005.3

[33] Youfeng Wu and James R. Larus. 1994. Static Branch Frequency and Program Profile Analysis. In *MICRO* (San Jose, California, USA). Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/192724.192725

[34] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the Flow: Profiling Copies to Find Runtime Bloat. In *PLDI* (Dublin, Ireland). Association for Computing Machinery, New York, NY, USA, 419–430. https://doi.org/10.1145/1542476.1542523

[35] Bahram Yarahmadi and Erven Rohou. 2020. Compiler Optimizations for Safe Insertion of Checkpoints in Intermittently Powered Systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings* (Samos, Greece). Springer-Verlag, Berlin, Heidelberg, 169–185. https://doi.org/10.1007/978-3-030-60939-9_12

Received 09-NOV-2023; accepted 2023-12-23