Raced Profiles: Efficient Selection of Competing Compiler Optimizations

Raced Profiles: Efficient Selection of Competing Compiler Optimizations

Abstract

Many problems in embedded compilation require one set of optimizations to be selected over another based on run time performance. Self-tuned libraries, iterative compilation and machine learning techniques all compare multiple compiled program versions. In each, program versions are timed to determine which has the best performance.

The program needs to be run multiple times for each version because there is noise inherent in most performance measurements. The number of runs must be enough to compare different versions, despite the noise, but executing more than this will waste time and energy. The compiler writer must either risk taking too few runs, potentially getting incorrect results, or taking too many runs increasing the time for their experiments or reducing the number of program versions evaluated. Prior works choose constant size sampling plans where each compiled version is executed a fixed number of times without regard to the level of noise.

In this paper we develop a sequential sampling plan which can automatically adapt to the experiment so that the compiler writer can have both confidence in the results and also be sure that no more runs were taken than were needed. We show that our system is able to correctly determine the best optimization settings with between 76% and 87% fewer runs than needed by a brute force, constant sampling size approach. We also compare our approach to JavaSTATS(10); we needed 77% to 89% fewer runs than it needed.

1. Introduction

Measuring the execution time of a program is used in a number of ways to select the best compiler optimizations. In iterative compilation(2; 6; 12), different versions of a program are created with different optimization settings. Each version is profiled and the best is used for the final delivery of the program.

Although iterative compilation produces excellent results, the costs can be prohibitive for ordinary use. Machine learning techniques (1; 22; 17; 18) have been used to solve this problem. Heuristics are tuned 'at the factory' so that thereafter the optimization space does not need to be searched. Machine learning has successfully tuned heuristics for embedded applications that out-perform their expert derived counterparts. The training data for the machine learning tools, however, must be generated by large scale iterative compilation. The compute time to profile all the different versions of the training benchmarks can be on the order of weeks or months(9).

Each variation of a program must be run multiple times because of noise in performance measurements; everything from the other processes running on the machine or the state of the file system to the temperature of the computer can have an effect. This becomes more of a problem as the granularity of the measurements becomes finer. When individual functions and loops are measurement targets, the noise to signal ratio can be significant(17).

Different approaches are taken to circumvent the noise problem. In some instances, researchers have chosen a fixed sample size plan, running each program version a constant number of times without observing how the results are shaping up as they go(1; 6). The hope is that the constant number of runs is sufficiently large to yield good results but not too large to waste effort. Often there is no analysis presented as to whether this number of runs is truly sufficient or if it is too many; confidence intervals and standard error bars rarely feature on performance graphs.

It may be tempting to use simulation to overcome noise, but not only are simulators slow and incompletely accurate, they are also subject to measurement bias (19). To overcome that bias random variations in setup must be effected and simulations run multiple times; the result is noise in the measurements, just as there is noise for direct execution.

What is needed is a technique which provides statistically rigorous results in the presence of noisy data and simultaneously reduces the cost of searching a large space of optimization settings. To the best of our knowledge, there is little prior work in this area. The closest to our work are (10; 3) where the authors recommend statistical rigor. They examine each point in the compiler optimization space in isolation and propose performing executions until an estimate of the sample's inaccuracy is tolerably small or some maximum number of executions is reached. While this gives accurate measurements, it does not reduce the total number of executions needed for the whole optimization space. Indeed, we will show that their approach can require more executions than a perfectly selected constant sized sampling plan.

An algorithm for selecting the number of times to execute each program version is called a sampling plan(26). Sequential sampling plans(24), used in medical trials(27), adjust the sample size dynamically as data is gathered. These sequential systems adapt to ensure both that sample sizes are large enough for good results and that sampling stops when enough data have been collected.

This paper develops an algorithm which:

- Determines a subset of the optimization settings or program versions which are 'better' than all others, to some user supplied confidence level.
- Minimizes the number of runs required, dropping poorly performing versions early and finishing when sufficient data has been gathered for a decision.
- Provides statistically rigorous results.

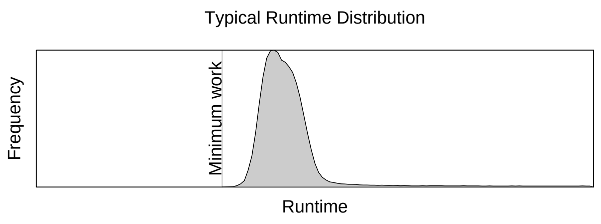


Figure 1. Typical distribution for the run time of a program. All programs have a minimum amount of work to achieve, giving a lower bound to the distribution. The shape of the distribution may vary but very often long tails can be observed. If small samples include observations from the tail then means can be thrown off. Without statistically rigorous techniques such situations will not be detected.

- Allows more points in the compiler optimization space to be examined.

Our algorithm 'races' different program versions, allowing those performing poorly to fall by the wayside with minimal sample size, whilst those fighting for the winning position are allowed more rein, increasing their sample size until either one wins or several draw.

We show that our adaptive, sequential sampling plan can dramatically reduce the number of runs needed to find the best program version. In this way, a greater part of the optimization space can be explored than would otherwise be possible.

The remainder of this paper is organized as follows. The next section presents examples showing the problem of noisy measurements. Section 3 gives an overview of our algorithm for the adaptively managed sampling plan and section 4 gives an in depth description of the technique. Then, section 5 shows the set up for our experiments, the results of which are given in section 6. Finally, sections 7 and 8 discuss related work and our conclusions respectively.

2. Motivation

Performance measurements on real systems are invariably noisy, an issue which becomes more problematic as the granularity of the measurement decreases. Figure 1 shows a typical distribution of cycle counts taken from function `run_length_encode_zeros` in the `MediaBench` `epic-encode` program. There is a minimum amount of work that the program must do, so there is a lower bound to the run time. On the other hand, there is no clear upper bound and the distribution features a long tail. The long tail of the distribution extends well beyond the median point and outliers from that tail, if included, can throw out the mean of small samples.

2.1 Confidence Intervals

Attempts to measure performance must be aware of the shape of its distribution. Certainly, a single observation will be insufficient to be confident that we have a good approximation for the performance; it might be nowhere near the mean. As the sample size grows, containing more and more observations of real measurements, we can be progressively more confident that the sample mean models the true mean of the distribution. As the sample size tends to infinity the difference between the sample mean and the true mean tends to zero.

Confidence intervals can be used to assess whether samples are sufficiently large. These statistical ranges show where the likely value of the true mean falls. A confidence interval always contains the sample mean and extends for some distance from it in each direction. As the number of observations in a sample increases the width of the confidence interval decreases; we become more confident that we can pin down the true mean to be closer to the sample mean.

Confidence intervals are also parametrized by a probability or confidence level. A confidence interval with a confidence level of 99% indicates that we are 99% sure that the true mean is inside the interval; only in 1% of trials should the true mean fall outside. Higher confidence levels require wider confidence intervals; conversely, if only low confidence is demanded the interval can be very small.

The difficulty in doing experiments with performance measurements is deciding how many observations are needed for each sample so that a confidence interval around the mean is sufficiently small. Typical, fixed size sampling plans require that this number of observations be fixed before any data is actually gathered and before any estimates of the noise are available.

The next section shows an example in which confidence intervals can prove or disprove the adequacy of different sample sizes.

2.2 Choosing a Sufficiently Large Sample Size

Figure 2 shows the number of cycles used by a loop in the function `run_length_encode_zeros` in the `MediaBench` `epic-encode` program. The loop was unrolled a different numbers of times, to see which unroll factor most improved performance. For each unroll factor, the program was run a certain number of times (2, 4, 8 or 32) and the number of cycles was recorded. We plot, in each of the four graphs, the mean of the samples together with their 95% confidence intervals (note that the axes change between figures).

When the sample size is only 2 (top left graph in figure 2) we cannot say which unroll factor is the best. We can already be sure that some unroll factors are doing badly (for example, factor 4 is bested by factor 9). However, the confidence intervals for some factors are so wide that we cannot be certain which has the lowest mean. With a constant sized sampling plan we have found that our sample size was too small.

As the sample sizes increase the confidence intervals become narrower. By the time we have sample sizes of 32 (bottom right graph in figure 2), we see that the complete interval for unroll factor 8 is lower than all the others and we thus find that factor to be the best. 32 executions of the program for each unroll factor are sufficient to tell which one to choose.

However, this simple, constant sized sampling plan does more work than necessary. Looking at the graph for sample size 4, we can see that the majority of the unroll factors were worse than factor 8; for all but factors 2, 9 and 10 the confidence intervals lay completely outside[1] the one for factor 8. If we had stopped executing those factors after sample size 4 and continued to 32 for the remaining 4 factors, we would only have executed each unroll factor an average of 7 times, a 78% reduction in the cost of sampling.

2.3 Choosing When to Stop Sampling

For some programs there will be no clear winner between two different versions. Alternatively, the difference might be so small compared to the noise that a huge sample size might be required to separate the program versions. In such a case we would like to stop early to avoid wasting effort. Consider the graph in figure 3; this graph shows the 95% confidence intervals of a different loop in `MediaBench` `epic-encode`. Again, the loop is unrolled different amounts, from 0 to 16, but this time the sample size is 1000. Even with this huge increase in sample size, only a few unroll factors (0,

[1] We do not advocate performing statistical tests visually in this fashion. Rather, bona fide Student's t-tests, ANOVA, etc. should be used.

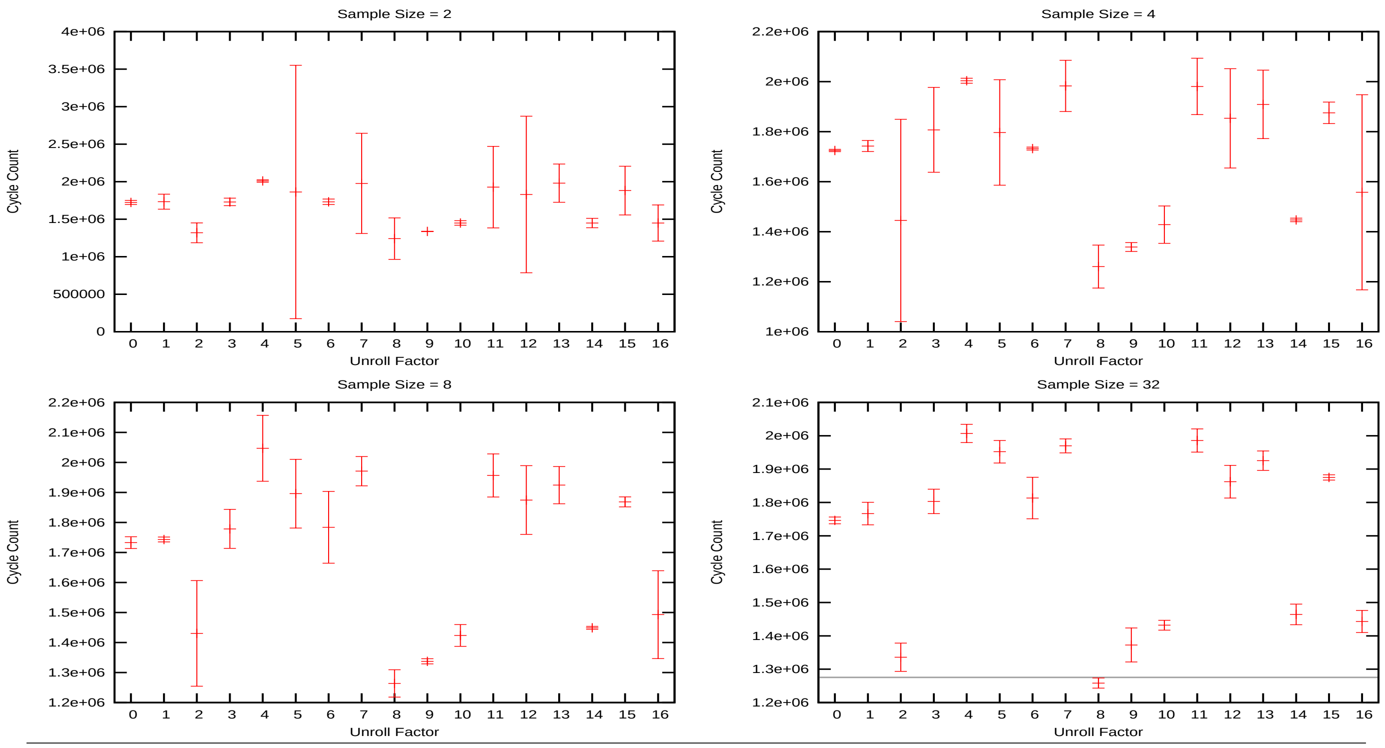


Figure 2. Confidence intervals at different sample sizes.
Data is from function `run_length_encode_zeros` in `MediaBench` `epic-encode`. Cycle counts are shown for different unroll factors of a particular loop. Only when the sample size is 32 per unroll factor does an unambiguous winner emerge.
The confidence interval is 95%. Note that the cycle count axis changes in each sub-figure.

1 and 11) can be excluded because their confidence intervals are completely disjoint to the one for unroll factor 4.

However, if we look at the worst case for unroll factor 4 and compare it to the best case for the other unroll factors we find a ratio of no more than 1.0065. In other words, if we chose factor 4 as the best factor, we could be fairly confident that if we are wrong it would be by not much more than 0.65%. The user, searching for the best program version might consider such a small error acceptable and agree that we need not execute the programs more times.

The situation is likely to be different for each program. In some cases, a small, constant sample size will suffice, in others much larger sample sizes must be taken. The user cannot, in general, know ahead of time how large the sample size should be.

This paper presents a mechanism by which the sample sizes are adaptively managed to ensure statistically valid results while at the same time drastically reducing the number of executions times needed to select the best program version.

3. Method

This section presents our sequential sampling method. The essential idea of our algorithm is that we:

1. Maintain a sample for each program version
2. Determine which versions are worse than any other - these are losers
3. Finish if there is only one non losing version or the non losers are close enough to each other
4. Increase the sample size by one in each non loser
5. Repeat from step 2

The algorithm ‘races’ program versions to find out which will win - i.e. have the best performance. Poorly performing versions are knocked out of the race while potential winners continue, increasing their sample size. The moment we find there is either one clear winner or that the front runners are all good enough, we stop.

Figure 4 shows a pictorial example of our algorithm. In the first panel, (a), the samples are initialized for each program version. Each version is, at this point a potential candidate to be in the winning set. We ensure that enough observations are in each sample for our statistical tests to work since they require a minimum sample size; little can be said statistically about a single observation.

In panel (b) the samples have been tested to see if any can already be identified as clear losers. A number of statistical tests are run (described in section 4) and any version shown to be worse than one of the other versions is taken out of the race. Since the remaining set of potential candidates contains more than one version we perform another set of tests to see if the versions are all approximately equal (detailed in section 4). At this point in the example there is not enough data to call the versions equal.

In panel (c) another observation is added to each sample and the process is repeated. As sample sizes grow, the amount of information available about each version increases; statistical tests become more able to make decisions about the relative merits of the different versions.

In the last panel, (d), the algorithm has run for several steps and discarded all the program versions but two. It has decided these are approximately equal so it returns them both.

3.1 Student’s T-Tests

Our algorithm makes heavy use of statistical testing to determine which means are unlikely to be in the winning set.

We use a student’s t-test(29) which is a statistical test that can determine if the means of two samples are significantly different. It may be that differences observed in the sample means are due to the sample sizes being too small, rather than because the true means themselves are different. A t-test can only be used to check that the

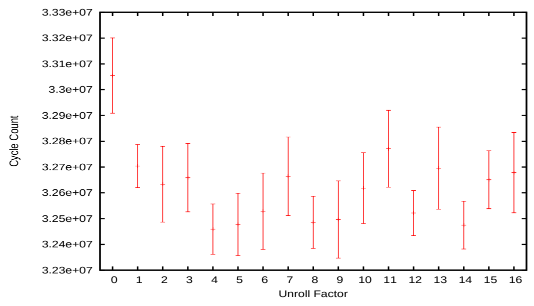


Figure 3. 95% Confidence intervals at sample size of 1000 for a loop in function `internal_filter` of benchmark `MediaBench epic-encode`.

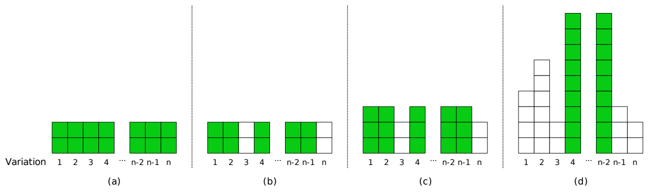


Figure 4. Several steps from our algorithm. In (a) initial samples are taken. In (b), versions that are clear losers are dropped. In (c), remaining versions are not deemed equivalent so the samples for them are grown by one. In (d), several steps have been run and the two remaining versions are found to be sufficiently good; the algorithm terminates.

means are different; if it does not declare the means to be different, that does not necessarily mean that the means are the same, it could also be that the sample sizes are too small to verify the difference.

The confidence level, α, of a t-test indicates the probability that the test will assert a difference in the means when none in fact exists (called a Type-I error). The lower this value the more confident we can be that a stated difference is real.

The t-test computes a 't-statistic' over the samples and compares this to a point on the cumulative distribution function (CDF) for the t-distribution (a probability distribution at the heart of the t-test). The point on the CDF parametrized by the confidence level, α, and an estimate of the number of degrees of freedom in the samples. The t-test normally indicates that the means are significantly different if the t-statistic is greater than the given point on CDF.

A t-test makes assumptions about the shape of the distribution and prefers it to look as normal as possible. There are also different variations on the t-test depending upon the exact use and what additional assumptions can be made. For example, if the sample sizes are equal or the variances are guaranteed to be equal then stronger tests can be used. There also alternatives to the t-test, for example the Mann-Whitney U test(14), which make different assumptions about the distributions under test. These alternatives, however, are generally not preferred if the t-test is applicable.

4. Algorithm Details

This section describes the algorithm and it's component parts in depth. Pseudo-code is presented in Algorithm 1.

Our algorithm begins with a set of all the program versions, C. For every version we maintain a sample which consists of the run time values we have taken so far for the corresponding version; these are S_c. In reality, we only need to record the sufficient descriptive statistics to determine the mean, confidence intervals and perform statistical tests, we never need to remember the complete list of run time observations and so the amount of space required by the algorithm is linear in the number of program versions.

The loop in lines 3 to 8 forms the bulk of the algorithm. First we remove any version that is provably worse than any other. Then we terminate if there is only one candidate left or all the remaining candidates are equal. Line 6 increases the sample size for remaining versions. Finally we terminate if some user defined limit is reached, allowing a hard boundary to be imposed on the total number of times each program version will ever be executed.

The loop does not remember which program versions were losers from iteration to iteration. This means that a version which is found to be a loser in one iteration has the opportunity to reenter the race later on. It can happen that a version deemed promising early on turns out to be less so once more information about it has been gathered. We found that reconsidering losers provided lower error rates.

A detailed description of the subroutines used by the algorithm follows.

Initialization At the beginning of the algorithm the sample sets are initialized to have two observations, the minimum necessary to make statistical inferences with a t-test. If other statistical tests are used, the initial number of observations may have to be different.

Sampling the Run time The cycle count for the current program version is measured by the function $sampleRuntime$. It assumes that there is some mechanism to profile the program to calculate the measurement.

Of special note here is that we take the natural logarithm of the run time. The reason for this is that run time distributions are both skewed and often suffer from outliers; applying a log transform is a common way to make the distribution look more 'normal' and to reduce the effects of outliers(4). Having distributions which are closer to a normal distribution frequently improves the accuracy of statistical tests. More general transformations, such as a Box-Cox transform(7) of which the logarithm is a special case, could be used instead, but we found adequate results from the simple logarithm.

Weeding Out Losers The algorithm needs to determine which versions are unlikely to be in the final winning set which is handled by the function, $losers$. The set of losers consists of any version, c, for which there is another version, d, that looks to be better performing. The 'better performing' test is a relation, $<_{\alpha_{LT}}$, over samples, (S_d, S_c).

To determine membership of the relation, $<_{\alpha_{LT}}$, we first check that the mean, μ_d, of S_d is less than the mean, μ_c, of S_c. If that is the case then we perform a student's t-test to discover if the difference in the means is significant to some user supplied confidence level, α_{LT}.

Since we cannot be certain that the variances are equal and since also the number of observations in each sample may be different, we use Welch's t-test(5) where the t statistic is:

$$t = \frac{\mu_d - \mu_c}{\sqrt{{s_d^2}/{n_d} + {s_c^2}/{n_c}}}$$

and μ_i, s_i^2 and n_i are the sample mean, variance and size of the i^{th} program version, respectively.

The degrees of freedom are estimated by the Welch-Satterthwaite equation(21):

$$d.f. = \frac{\left({s_d^2}/{n_d} + {s_c^2}/{n_c}\right)^2}{\frac{\left({s_d^2}/{n_d}\right)^2}{(n_d-1)} + \frac{\left({s_c^2}/{n_c}\right)^2}{(n_c-1)}}$$

As the testing proceeds, any version removed is not used to compare against subsequent versions in the tests for this iteration. In this

Algorithm 1 Pseudo code for our algorithm

```
1. $C \leftarrow \{c; c \text{ is a compilation strategy}\}$
2. $\forall c \in C,\ S_c \leftarrow \{sampleRuntime_c(),\ sampleRuntime_c()\}$
3. for ever do
4.     $C' \leftarrow C - losers(C,\ S)$
5.     if $|C'| = 1$ or $candidatesEqual(C',\ S)$ then return $C'$
6.     $\forall c \in C', S_c \leftarrow S_c \cup sampleRuntime_c()$
7.     if $sampleThresholdReached(C',\ S)$ then return $C'$
8. end for
9. $sampleRuntime_c()$
10.     $x \leftarrow$ execute strategy $c$ and record runtime
11.     return $\ln x$
12. $loosers(C,\ S)$
13.     return $\{c \in C;\ \exists d \in C,\ d \neq c,\ S_d <_{\alpha_{LT}} S_c\}$
14. $candidatesEqual(C',\ S)$
15.     $B \leftarrow \{b \in C';\ \mu_b \leq \mu_c, \forall d \in C'\}$
16.     return $\bigwedge_{b \in B, c \in C'; b \neq c} S_b =_{\alpha_{EQ},\epsilon} S_c$
17. $sampleThresholdReached(C,\ S)$
18.     return $\bigvee_{c \in C} |S_c| \geq MAX\_SAMPLE\_SIZE$
```

way the algorithm guarantees that at least one version survives the *losers* function.

Finding the Winners Stopping when enough data has been gathered is important since we may find that some program versions are either identical to each other or so nearly so that the compiler writer is content with several winners. Increasing the sample sizes once we know this is a waste of effort and, left unchecked, may continue indefinitely.

In the happy case that one version has beaten all others, we can return that single winner. If more survive then we check to see if they are all sufficiently close together for the compiler writer, decided by function *candidatesEqual*.

Detecting that two distributions are approximately equal is the domain of equivalence testing(25). The archetypal equivalence test is based on Westlake intervals(28) wherein a confidence interval is formed for the difference between two means and if that interval is completely contained within some 'indifference region' about zero then the distributions are considered equal. Indifference regions cannot be selected analytically, it is up to the compiler writer to express when distributions are equivalent.

In our case, the Westlake interval would require the indifference region to specified as a difference of an absolute number of cycles. It is more natural, instead, to describe the indifference region as an upper bound on the ratio of means. Our justification for this is that speed ups and slow downs are important in compiler fields, but rarely are absolute cycle counts. We might consider a speed up of 1.001 to be uninteresting, regardless of whether it represents one million or ten billion cycles; conversely a speed up of 1.5 is likely exciting with similar disregard for the number of cycles in the difference.

We expect compiler writers to be interested in the version returned with the lowest mean (the rest of the set we expect to be only of cursory interest, except perhaps to machine learning tools). We use this information to tune our equivalence test to the problem. Since the compiler writer will choose the one from the non losing set with the lowest mean, we wish to ensure, to some confidence, that none of the other non losers could have provided much of a speed up compared to that choice.

We can determine the most available speed up between two program versions by taking a confidence interval for each sample:

$$upper = \mu + t_{(\alpha_{EQ}, n-1)}\sqrt{s^2/n}$$

$$lower = \mu - t_{(\alpha_{EQ}, n-1)}\sqrt{s^2/n}$$

where t is the student's t statistic for a given user supplied confidence, α_{EQ}, and μ, s, and n are as before.

Now, if the program version with the lowest mean is b, and we have another version, c, we can calculate the worst case speed up of c over b by comparing the upper end of b's confidence interval against the lower end of c's interval. However, we must remember that we initially transformed our observations with a logarithm transform. If we simply compare the interval ends directly we will not be describing speedups; we must apply the inverse transform, first (note that it is this transformation which prevents us using Fieller's theorem(8; 13) for the confidence interval of the ratio of two means). The speedup is:

$$worstspeedup_{b,c} = \frac{e^{upper_b}}{e^{lower_c}} = e^{(upper_b - lower_c)}$$

This shows us the speed up in the worst case where the true value of the mean for version b is at the upper end of its confidence interval and the true mean for c is at the lower end of its interval.

The compiler writer, having already given a significance level, α_{EQ}, must also now specify an indifference region, ϵ. This gives the maximum worst case speed up for the equivalence test. Putting this together, we now have a relation, $=_{EQ,\epsilon}$, used in line 16, over pairs of samples such that:

$$(S_b, S_c) \in =_{EQ,\epsilon} \leftrightarrow 1 + \epsilon < worstspeedup_{b,c}$$

Other stopping conditions are possible, but we believe that this estimate of the nearness of the run times closely matches the requirements of iterative compilation. If the current experiment at hand needs a different stopping condition it should be easy to adjust our algorithm to it.

Limiting Total Sample Size Our algorithm also gives the compiler writer the opportunity to place hard limits on the total sample size through the constant, *MAX_SAMPLE_SIZE* in function *sampleThresholdReached*. This fixed limit makes ours a restricted sampling plan(26); other methods exist to ensure closed sample boundaries(20) and may be worth considering, although we have found that the restricted plan is quite adequate.

Having a hard sample size limit allows the compiler writer to choose how keen they are for correct results. A large limit permits the algorithm to expend more effort disambiguating difficult cases. Feedback is given when the limit is reached so that in those cases the compiler writer can either accept the best estimate so far from the algorithm or reject, deciding that further effort is not warranted. With a combination of this limit and the two confidence levels, the compiler writer can tune the breadth and accuracy of the space they wish to explore.

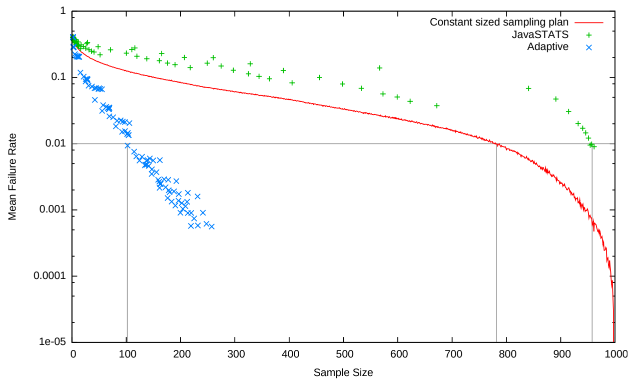
5. Experimental Setup

In this section we briefly describe the experimental set up. We performed two different experiments; the first was to find the best unroll factor for loops, while the second was to find the best compiler flags for whole benchmarks.

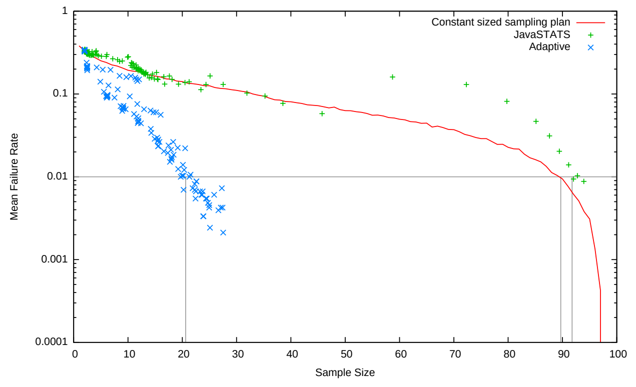
5.1 Experiments

Loop Unrolling Experiment Loop unrolling has been targeted in a number of previous machine learning and iterative compilation works(17; 22). Being a fine grained optimization, there is a wide range of variation in noise to signal ratios in different loops.

Compiler Flags Experiment Finding the best compiler flags has also been widely explored in both iterative compilation and machine learning(9). The very long data gathering phases, often equating to months of compute time(9), make efficiency of paramount importance.



(a) Loop Unrolling. At the 1% failure rate, an adaptive plan with $\alpha_{LT} = 0.02$ and $\alpha_{EQ} = 0.02$ needed only 102 samples compared to an optimal fixed sample plan of 780 samples, a reduction of 87%. JavaSTATS required 956 samples to dip below the 1% failure rate, our reduced that by 89%.



(b) Compilation Flags. At the 1% failure rate, an adaptive plan with $\alpha_{LT} = 0.002$ and $\alpha_{EQ} = 0.01$ needed only 21 samples compared to an optimal fixed sample plan of 90 samples, a reduction of 76%. JavaSTATS required 92 samples to dip below the 1% failure rate, our reduced that by 77%.

Figure 5. Comparison between our method, constant sized sampling and JavaSTATS(10). Mean failure rates are shown against average sample sizes. A failure is when the given number of samples fails to find a program version that is within 0.5% of the cycle count of the best version. Intersection with a failure rate of 1% is shown. The many points of our adaptive sampling plan show results with different values of α_{LT}, the confidence interval for the $losers$ function, and α_{EQ}, the confidence interval for the $candidatesEqual$ function, for these, θ, the equivalence threshold, is always 0.5%. Different points from the JavaSTATS algorithm are also shown with varying α and θ.

5.2 Compiler Setup

For both experiments we used GCC 4.3.1. In the first we extended the compiler to allow unroll factors to be explicitly specified for each loop in a program. In the second we altered GCC to accept command line arguments externally, regardless of the benchmarks' makefile. This allowed us to force different compilation flags to be used.

5.3 Benchmarks

Loop Unrolling Experiment For the loop unrolling experiment we took 22 embedded benchmarks from the MediaBench and UTDSP benchmark suites. Those benchmarks which did not compile immediately, without any modification except updating path variables, were excluded.

Compiler Flags Experiment For the compiler flags experiment we added the MiBench suite, extending the number of benchmarks to 57. Again, we excluded those which did not immediately compile.

5.4 Platform

These experiments were run on an unloaded, headless machine; an Intel dual core Pentium 6 running at 2.8 GHz with 2Gb of RAM. We used a fast machine so that we could gather sufficient data for the naïve, constant sized sampling plans.

5.5 Data Generation

Loop Unrolling Experiment For loop unrolling, we selected each of the 230 loops in the benchmarks and unrolled them different number of times. Each loop was unrolled between 0 and 16 times inclusive, giving a total of 17 different program versions per loop. Only one loop was modified at any one time, meaning that a program with ten loops would be compiled 170 times. This allowed each loop to be considered in isolation.

The current loop being changed was instrumented to record the cycle count before and after the loop. Each different version of a program was run 1000 times.

Compiler Flags Experiment For the compilation flags experiment, we took 86 of GCC's flags and generated random collections of them. Each flag had a 5% chance of being set in each collection. Each benchmark was compiled using the flags from each new collection. If that produced a different binary than had already been seen for the benchmark, then the binary was run at least 100 times with the cycle count recorded.

On some benchmarks, particularly small ones, the compiler would generate identical binaries for many different flag collections. Thus, the number of different points in the compiler optimization space varied across the benchmarks, from 34 for `mibench.telecomm.adpcm` to 288 for `mediabench.pegwit`.

5.6 Failure Rate

We need a method to compare the different sampling plans. We want to ensure that the result of a sampling run chooses a program version which, if it is not the best, is slower than the best by no more than a given amount. If the version chosen by a sampling run is further from the best, then we call the run a failure, else we call it a success. This allows, by repeated running of the sampling plan, to generate mean failure rates and hence compare the quality of different plans; a better plan will have a lower mean failure rate.

Specifically, we desire that the true mean of the performance of whatever program version is finally selected by a plan should be sufficiently near to the true mean of the best possible version. The true mean is estimated by taking the mean of the full data set. We distinguish the estimated true mean as $\overset{*}{\mu}_i$ for the i^{th} program version, as opposed to the sample mean, μ_i.

If some sampling plan selects a version, c, and the best possible version according to the complete data set is b, then we compare ratio of the estimated true means, $\overset{*}{\mu}_b/\overset{*}{\mu}_c$, which gives the slowdown caused by choosing version c over version b. If the ratio is greater than $1-\theta$, for some positive θ, then we call the trial a success, otherwise it is a failure. In our experiments we fix θ to 0.5%, which means that, to be successful, a sampling run must choose a program version whose true mean is no more than 0.5% slower than the true mean of the best possible version.

5.7 Techniques Evaluated

Profiled Races To evaluate our approach, we explored different values of the parameters which define our algorithm. Both the confidence level for the less than test in the $losers$ function, α_{LT}, and α_{EQ}, the confidence level for the equality test in $candidatesEqual$, were allowed to range over the set {0.0001, 0.005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5}. The set of confidence levels gives a broad spectrum of confidences from extremely high confidence at 0.0001 to very low confidence at the other end. For example, when $\alpha_{EQ} = 0.5$, the confidence intervals for two samples, even small ones, will likely be very narrow, so the algorithm will be very generous considering if two versions are equivalent. Conversely, if $\alpha_{LT} = 0.0001$, then the confidence interval for two samples will be wide unless either the samples are large or there is a very small standard deviation; the algorithm will be very sure before deciding that one program version outperforms another.

In all cases, the threshold, θ, which determines how far from the best is acceptable in function $candidateEqual$, is set to 0.5%, matching the boundary for failure. The *MAX_SAMPLE_SIZE* constant is set to the maximum amount of data available in each experiment; in day-to-day use this constant may not be so large. Each parameter setting was run 100 times to determine the mean failure rate and average sample size for those values.

Constant Sized Sampling Plan The first technique we compared against is a straight forward constant sized sampling plan. Here a fixed number of observations is taken of each program version's runtime or cycle count. Again, we ran plans for each sample size 100 times to generate mean failure rates.

JavaSTATS The second method for comparison is the statistically rigorous approach, JavaSTATS(10; 3). JavaSTATS runs each program version until an estimate of the sample's inaccuracy is sufficiently small. The inaccuracy metric is a confidence interval divided by the sample mean. This metric provides a unit-less indicator of the accuracy of the current sample; a value near to zero is an accurate sample. As the sample size grows to infinity the metric generally approaches zero.

Several parameters are required: α, the confidence level for the confidence intervals; θ, the threshold at which the metric indicates an accurate sample and minimum and maximum sample sizes. We allowed the confidence level and threshold to both range over the set {0.0001, 0.005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5}. The minimum and maximum sample sizes were set at 2 and the maximum size of the data respectively, just as for our own algorithm.

6. Results

For each of our two experiments we compared average sample sizes and mean failure rates of the different techniques with their different parameter values.

6.1 Loop Unrolling Experiment

Figure 5(a) shows the performance of the different techniques for the loop unrolling experiment. A horizontal line indicates a 1% mean failure rate, an arbitrarily chosen point of comparison.

Constant Size Sampling Plan On average, a sample size of 780 or more is needed per unroll factor to achieve a failure rate less than 1%. In practice the size will have to be greater since the compiler writer cannot have prior, perfect knowledge of how many observations are needed in the samples.

JavaSTATS Sampling Plan To manage the 1% failure rate, JavaSTATS needed tight settings of $\alpha = 0.0005$, $\theta = 0.0001$. At these settings, the average sample size was 957, nearly all the data and worse than the constant plan. We attribute this to the fact that it will sometimes fail early, damaging the mean failure rate; punitive settings are needed to compensate, which while stopping the early failures also force large sample sizes when no early failure has occurred. On the other hand, with such aggressive settings, the compiler writer gets good results without perfect knowledge of the right sample size as is required for the constant sampling plans.

Since each program version in the compiler optimization space is considered independently of the others, JavaSTATS will spend as much effort producing accurate estimates for the poor ones as for the best. JavaSTATS brings only statistical rigor, not efficient iterative compilation search.

Profile Races Our algorithm performed very well compared to both of the other plans. At $\alpha_{LT} = 0.02$ and $\alpha_{EQ} = 0.02$ our adaptive algorithm first dips below an average failure rate of 1%, getting a failure rate of only 0.94% for an average sample size of only 102. This is 87% less than for the fixed size plan and 89% less than JavaSTATS.

Even applying very strict confidence levels of $\alpha_{LT} = 0.0001$ and $\alpha_{EQ} = 0.0001$, our algorithm attained a tiny mean failure rate of just 0.056% for a small increase in sample size to only 257.

6.2 Compiler Flags Experiment

Figure 5(b) shows the performance of the different techniques for the compiler flags experiment. The graph looks very similar to that of the loop unrolling experiment.

Constant Size Sampling Plan To achieve a failure rate less than 1%, an average sample size of 90 was needed per program version.

JavaSTATS Sampling Plan Once more, JavaSTATS did slightly worse than the constant plans. At settings of $\alpha = 0.001$, $\theta = 0.0001$ the failure rate dipped below 1% with an average sample size of 92. Again, it is still better to use JavaSTATS than the constant plan since it does not require knowing the perfect sample size ahead of time.

Profile Races At $\alpha_{LT} = 0.002$ and $\alpha_{EQ} = 0.01$ our algorithm gets a failure rate of less than 1%, needing only 21 observations on average in each sample. This is 76% less than the constant plan and 77% less than JavaSTATS.

Again, cautious use of very strict confidence levels, $\alpha_{LT} = 0.0001$ and $\alpha_{EQ} = 0.0001$, is not costly. With these values, our algorithm needs a sample size of just 27 and gets a mean failure rate of 0.021%.

6.3 Parameter Sensitivity

Figure 6 shows a few contours for different fixed values of the α_{LT} and α_{EQ} confidence levels, demonstrating the role those parameters play in controlling the sample size and failure rates. The contours in the figure are for the loop unrolling experiment, but are very similar to those in the compilation flags experiment.

The points with the highest mean failure rate are those with high values of α_{LT} and α_{EQ}. When these are 0.5, for example, the algorithm never increases the samples more than the minimum since at that level confidence intervals are very small. The points with the lowest failure rate are those where α_{LT} and α_{EQ} are also the smallest, as is expected since they demand more confidence before either discarding versions or determining equality.

6.4 Individual Cases

In the previous section we showed that our adaptive algorithm significantly out-performs a simple, constant sized sampling plan. In this section we show how our sequential sampling plan performs on a number of individual cases, giving a flavor of what to expect.

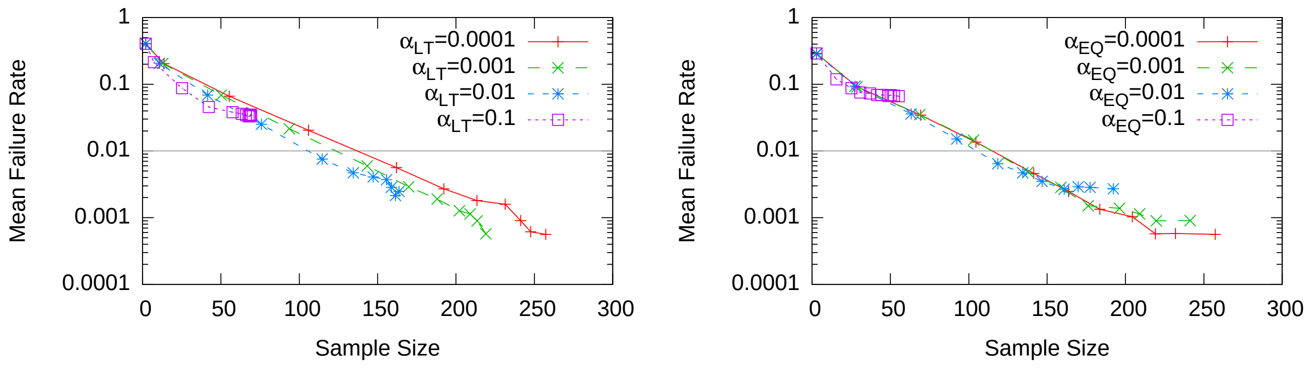


Figure 6. Parameter sensitivity. Contours of the two confidence levels, α_{LT} and α_{EQ}, are shown for the loop unrolling experiment in figure 5. α_{LT} is the the confidence level for the less than test in the $losers$ function; α_{EQ} is the confidence level for the equality test in $candidatesEqual$

Figures, 7 to 10, show the behavior of our technique over particular examples of the loop unrolling data set. The settings of α_{LT} and α_{EQ} are both 0.02, the point at which the failure rate is 1%.

In each figure, the left hand graph shows the mean cycle count after all data (1000 executions) are considered; the variability of the data is shown with error bars at one standard deviation.

The right hand graph shows how our sequential sampling plan performs, averaged over 100 simulations. The average sample size for each unroll factor is shown together with a one standard deviation error bar to indicate variability.

6.4.1 Low Variability, Single Winner

The first example, figure 7, shows a scenario where the variability in the cycle count (left hand graph) is very small and there is a single program version which significantly out performs all of the others. The algorithm excludes the poor versions, often without needing to execute them beyond the minimum number; it only needs to execute from the best three perhaps once more. The average sample size was 2.15, compared to a minimum sample size of 2. This example shows how the well the algorithm handles easy cases.

6.4.2 Low Variability, Multiple Winners

The next example, figure 8, is only a little harder. Unroll factors 1 and 3 are very close together but the others are poor by comparison. The poorly performing versions are removed very quickly and focus is left on the two remaining candidates. These two are, on average, found to be equivalent after 35 executions each. The average sample size was 6.05.

6.4.3 High Variability, Multiple Winners

In figure 9, the different versions are more difficult to distinguish. The algorithm needs larger samples to come to a conclusion, but still is able to reduce efforts on poorer versions. For this problem, average sample size was 161.8 with no failures in 100 simulations. On average, the algorithm returned 8 of the 17 unroll factors in the winning set.

6.4.4 Sample Exhaustion

Finally, figure 10, shows a much harder case. Here some of the versions could not be culled and, at the same time, could not be proved to be equal. In 97% of the sampling runs the algorithm terminated because the maximum sample size limit was reached. The average number of samples was 454.9 and again there were no failures.

7. Related Work

Iterative compilation has been explored for some time(2; 6; 12) and efforts have been made to reduce the cost of it(1), albeit by reducing the number of versions that need to be searched to find the best. Machine learning approaches to compilers attempt to automatically tune heuristics and require large sets of data, captured through iterative compilation(18; 17; 23). These works, to the best of our knowledge, all use only fixed sized sampling plans.

Efforts to promote statistical rigor in execution time measurements have been made(10; 3). In these, a program version is run multiple times until either an estimate of inaccuracy is sufficiently small or some maximum number is reached. Each point in the optimization space is executed until we have a good estimate of its mean so the data is statistically valid. However, this effort does not take into account the relative merits of each point. A point that is clearly bad will be refined just as much as the most promising point in the space. Since their technique considers each point in isolation it can perform worse than an optimally chosen constant sized approach.

In (19), the difficulties of avoiding measurement bias are described. The authors demonstrate that, even with a simulator, apparently innocuous modifications (such as sizes of irrelevant environment variables) can affect the performance of a program. They suggest that random changes must be made to the set up state so that multiple measurements are required. Even in simulators, previously a haven of noise free data, must correct measurements handle noise.

Sequential analysis, however, has been used reduce the cost of sampling in contexts from industrial processes(24) to medical trials(27). Our work is most similar to (15; 16) wherein machine learning models are raced to find the best. Their work, however, relying on Hoeffding's inequality(11), requires that the random variables under consideration are all bounded - which is not the case for run times. Moreover, their work only concentrates on removing poor performers, it does not consider the situation where some of the random variables are equivalent for practical purposes. To the best of our knowledge, our paper is the first to bring sequential sampling to iterative compilation.

8. Conclusion and Further Work

In this paper we have shown that using fixed sized sampling plans can have unintended consequences for performance measurement and iterative compilation. Too small a sample can generate incorrect results. Noisy data can make a small sample appear to have

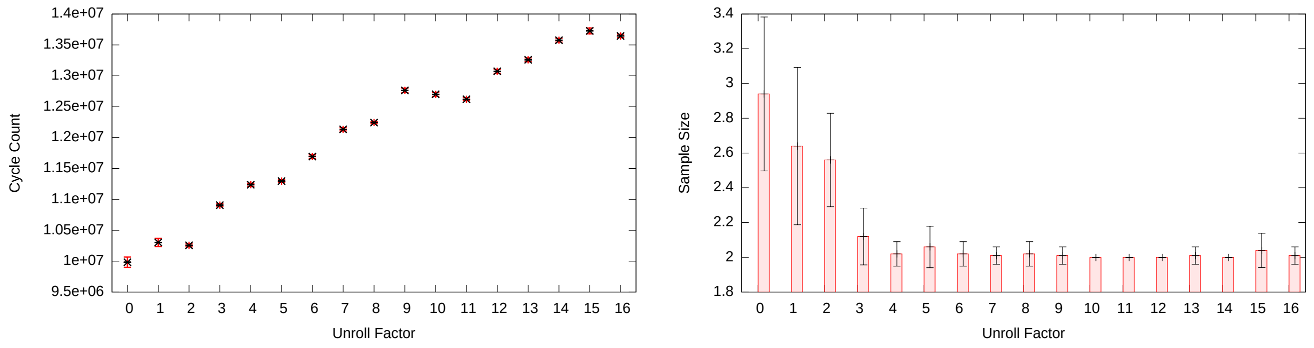


Figure 7. A loop from `MediaBench gsm-encode`, function `Gsm_preprocess`. If the winner is clear very early on, then very small sample sizes will result. Error bars are, in both graphs, one standard deviation.

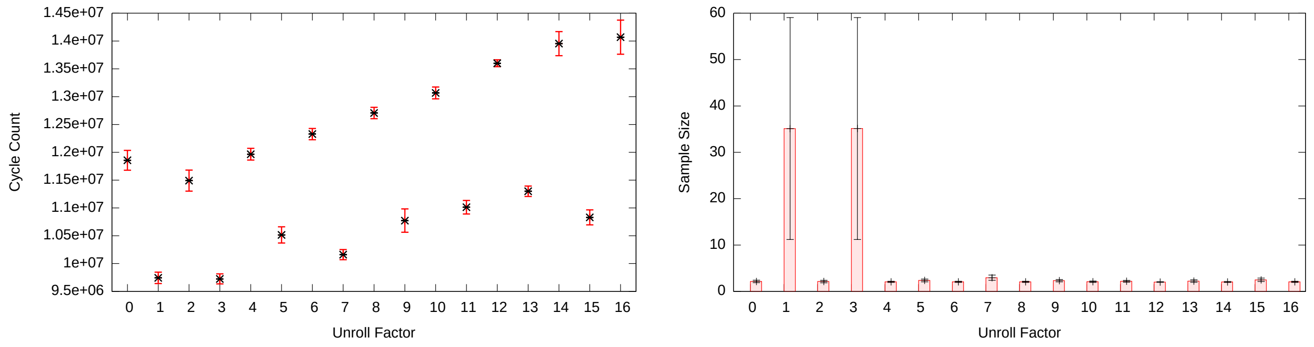


Figure 8. A loop from `MediaBench adpcm-decode`, function `adpcm_decoder`. Poorly candidates are quickly discarded and effort focused on the remaining set.
Error bars are, in both graphs, one standard deviation.

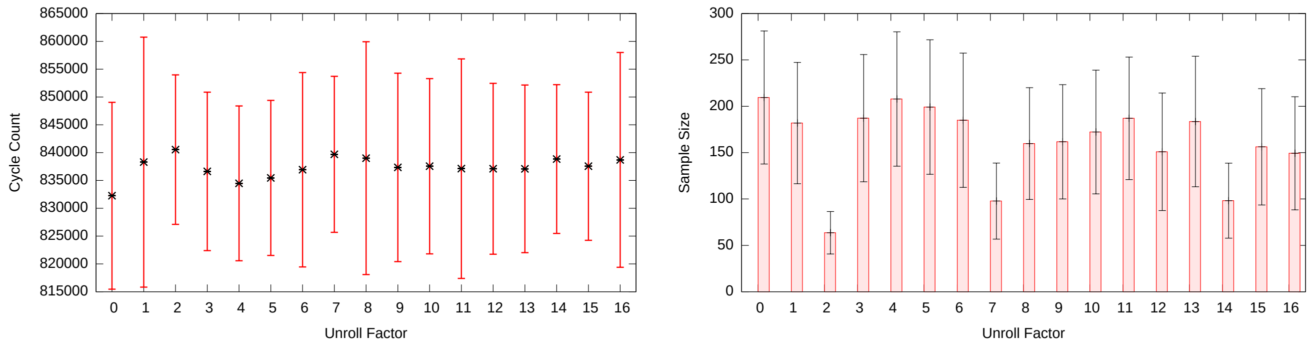


Figure 9. A loop from `MediaBench jpeg-encode`, function `emit_eobrun`. When the relative noise is large more samples must be taken.
Error bars are, in both graphs, one standard deviation.

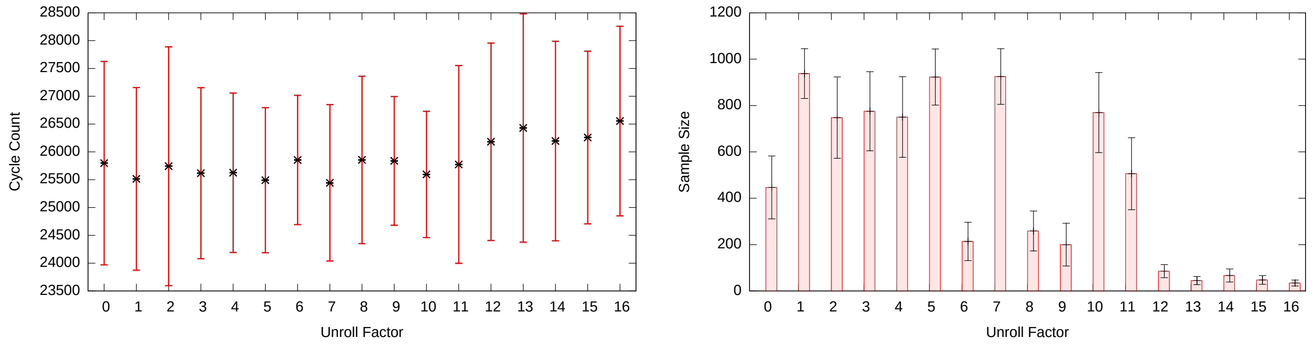


Figure 10. A loop from `UTDSP fft_1024`, function `main`. Sometimes the noise will case the sample limit restriction to be reached.
Error bars are, in both graphs, one standard deviation.

a promising mean where a larger sample would give a very different answer. It is not enough to only look at the means of performance measurements, the statistical significance of the results must be taken into account.

Too large a sample wastes excessive work since program versions are executed an unnecessary number of times. Often, some program versions could be discarded early but a fixed sampling plan is oblivious to these opportunities. With fixed sampling plans, the user must choose, ahead of time, how many runs each version needs to get good data. This is very difficult to do without already having data to examine.

We have provided an algorithm which automatically adapts to the requirements of the problem at hand. In cases where there is little noise and a clear winner is visible early the algorithm will take very few samples. When particular versions require larger sample sizes to disambiguate them the algorithm does just that. Finally, the algorithm terminates when versions are equivalent so that no further work is done trying to tell identical versions apart.

We applied our technique to finding the best loop unrolling factor for a number of loops from different benchmarks and also to finding the best compiler flags for whole programs. Some loops and programs generated noisy data while others had relatively clean data. Our method was able to adapt to these differences, choosing different sample sizes in each case. We reduced the cost of iterative compilation by between 76% and 87% compared to a fixed sized sampling plan. Compared to JavaSTATS(10), we reduced the cost by between 77% and 89%.

There are other possible formulations of the algorithm, using different criteria to remove versions and decide when to stop. We will explore these in the future.

References

[1] F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. pages 295–305, 03 2006.

[2] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 231–239, New York, NY, USA, 2004. ACM.

[3] S. M. Blackburn, K.S. McKinley, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA 2006, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2006.

[4] J Martin Bland and Douglas G Altman. Transforming data. *BMJ (Clinical research ed.)*, 312, mar 1996.

[5] B.L.Welch. The generalization of 'student's' problem when several different population varlances are involved. *Biometrika*, 34:28–35, 1947.

[6] F. Bodin, T. Kisuk, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Prole 14 and Feedback-Directed Compilation, in Conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 10 1998.

[7] G. E. P. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26(2):211–252, 1964.

[8] E.C.Fieller. Some problems in interval estimation. *Journal of the Royal Statistical Society*, 16:175–185, 1954.

[9] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, Francois Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael OB́oyle. Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC DevelopersŚummit*, June 2008.

[10] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[11] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, March 1963.

[12] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. *SIGPLAN Not.*, 38(7):12–23, 2003.

[13] M.A.Creasy. Confidence limits for the gradient in the linear functional relationship. *Journal of the Royal Statistical Society*, 18:64–69, 1956.

[14] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, 18:50–60, 1947.

[15] Oded Maron and Andrew W. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In *Advances in neural information processing systems 6*, pages 59–66. Morgan Kaufmann, 1994.

[16] Oded Maron and Andrew W. Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11:193–225, 1997.

[17] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics, 2002.

[18] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanović, Carla Brodley, and David Scheeff". Learning to schedule straight-line code. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[19] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[20] Armitage P. and Healy M.J.R. Interpretation of χ^2 tests. *Biometrics*, 13:113–115, 1957.

[21] F. E. Satterthwaite. An approximate distribution of estimates of variance components. *Biometrics Bulletin*, 2:110–114, 1946.

[22] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. 06 2003.

[24] Abraham Wald. *Sequential Analysis*. 1947.

[25] Stefan Wellek. *Testing Statistical Hypotheses of Equivalence*. CRC Press, 2003.

[26] G.Barrie Wetherill. *Sequential methods in statistics*. London, Methuen; New York, Wiley, 1986.

[27] John Whitehead. *The Design and Analysis of Sequential Trials*. Ellis Horwood, 1992.

[28] W.J.Westlake. Use of confidence intervals in analysis of comparative bioavailability trials. *Journal of Pharmaceutical Science*, 61:1340–1341, 1972.

[29] W.S.Gosset. The probable error of a mean. *Biometrika*, 6:1–25, March 1908.