Introducing the Pseudorandom Value Generator Selection in the Compilation Toolchain

Michael Leonard
Northwestern University
USA
michaelleonard2018@u.northwestern.edu

Simone Campanoni
Northwestern University
USA
simonec@eecs.northwestern.edu

Abstract

As interest in randomization has grown within the computing community, the number of pseudorandom value generators (PRVGs) at developers' disposal dramatically increased. Today, developers lack the tools necessary to obtain optimal behavior from their PRVGs. We provide the first deep study into the tradeoffs among the PRVGs in the C++ standard, finding no silver bullet for all programs and architectures. With this in mind, we have built PRV Jeeves, the first fully automatic PRVG selector. We demonstrate that when compiling widely-used, highly optimized programs with PRV Jeeves, we are able to cut execution time by 34% on average. This enhancement comes at no cost to developers.

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

Keywords Code generation, pseudorandom value generators, code selection

ACM Reference Format:
Michael Leonard and Simone Campanoni. 2020. Introducing the Pseudorandom Value Generator Selection in the Compilation Toolchain. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368826.3377906

1 Introduction

Interest in randomized algorithms has steadily grown within the computing community throughout the last several decades. Randomization has pervaded nearly every subfield of computer science, and is notably visible in encryption, machine learning, data analytics, robotics, and Internet of Things applications. Furthermore, there are no signs of reverting this trend in the near future.

Pseudorandom value generators (PRVGs) are at the core of most random programs. As their importance has grown, so too has their variety. This growth has notably been reflected in the widely adopted C++ programming language. Starting in C++11, the C++ steering committee has added one common interface to several varied, templated, and composable PRVGs as part of the language definition [2]. Today, C++ developers have dramatically more degrees of freedom to choose PRVGs that best suit their needs.

With so many options, the knowledge and effort required to construct the right PRVG goes beyond reasonable expectation for most programmers. In order to choose the best PRVG for an application, a developer needs to have a full understanding of the many tradeoffs in performance, memory consumption, and quality of randomness of all available PRVGs. This requires an understanding of the theoretical background of each PRVG's design, and while some select developers may have this knowledge, it is relatively uncommon. Furthermore, this paper demonstrates that these tradeoffs are architecture dependent. Hence, even if a developer does have the right background, to make an application optimal across platforms, he or she will have to write a PRVG that adapts to each target micro-architecture. The result would require hours of careful consideration into what is generally a tiny fraction of the total lines of code for any codebase. As such, the great majority of software relies on a single PRVG for all situations: `rand()`. This approach completely gives up on an immense opportunity to optimize program behavior across many potential dimensions.

Currently, developers are stuck with these two choices: use a simple, non-optimal PRVG, or take on the massive onus to make the right choice. Therefore, we propose migrating this decision to the compiler. Compilers have a long and successful history of abstracting away some of the most difficult decisions that developers originally had to make, such as mapping program variables to architectural registers and selecting the architectural instructions to perform higher-level program tasks. We propose adding one more decision to this compilation job list: the selection of PRVGs.

To explore the benefits possible from such a compiler extension, we have built the first fully automatic pseudorandom

CGO '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7047-9/20/02...$15.00
https://doi.org/10.1145/3368826.3377906

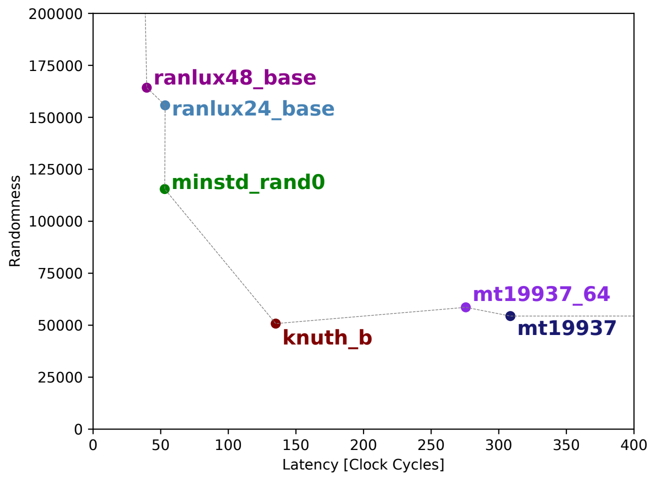

Figure 1. The tradeoff space between randomness and latency of PRVGs. The y-axis measures quality of random values generated by each PRVG by fitness to the birthday spacings theorem, and the x-axis measures average latency to obtain a single random value CPU cycles. These results were measured across all PRVG instantiations in C++11 on a 16-core Intel® Xeon® CPU with 32 GB DRAM. Visualizing the tradeoff reveals a pareto frontier, where the ideal PRVG would be at the origin. Here, we have zoomed in on the PRVGs near the inflection point of the pareto curve, which shows that there is no clear winner across both dimensions.

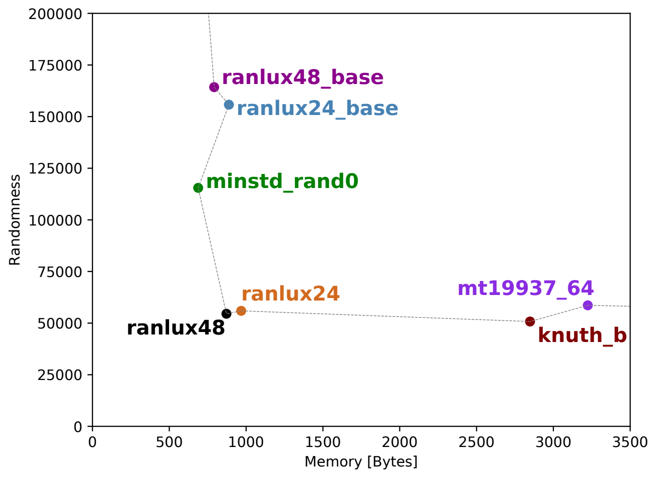

Figure 2. The tradeoff space between randomness and memory consumption of PRVGs. The setup is the same as Figure 1 except for the x-axis, which here measures the median working set size across several snapshots of program execution. Like Figure 1, there is no clear winner for all needs. However, note certain stark departures from Figure 1, especially the presence of the ranlux48 and ranlux24 generators on the inflection point of pareto curve. The optimal choices in the randomness-memory and randomness-latency tradeoffs, respectively, may be very different.

value generator selector. Our PRVG selector is built as an extension to the widely used Clang compiler, and as such, our system targets generic, unmodified C++ code. Without any added burden to the developers, our system analyzes the program given as input to identify the uses of PRVGs defined by standard libraries (e.g., rand()). Our compiler then iteratively tests substituting various replacements, drawn from a database of PRVGs, for each independent PRVG in the original code, until it converges on an optimal selection. Our autotuner drives this search, exploring the impact of each PRVG for the specific target architecture.

Our compiler decreases the overall execution time of the randomized PARSEC benchmarks by 34% on average, while preserving the original output quality. This speedup comes at no cost to developers, as it allows programmers to save time and effort in both development and maintenance. With our compiler, a developer may simply choose the first PRVG that comes to mind (e.g., rand()) and obtain better performance than even a carefully hand-tuned PRVG.

The rest of this paper is organized as follows. First, to motivate our problem, we dive into the complex tradeoff space among the PRVGs available in C++11 (Section 2). To the best of our knowledge, this is the first study of this kind. Then, we describe the design and implementation of our compiler and explain how it selects PRVGs (Section 3). Finally, we detail how we evaluated our system and demonstrate the performance benefits we have obtained on a widely adopted benchmark suite through automatic PRVG selection (Section 4). We conclude with an analysis of related work within this space (Section 5).

2 Opportunity

As interest in randomness has grown, the sheer number of PRVGs available to developers has skyrocketed. Unfortunately, developers' understanding of these PRVGs has not kept up. To characterize the current state of the PRVG space, we provide the first analysis of the various PRVGs available in C++. In this section, we detail the criteria we used for analyzing each PRVG, and we present a series of tradeoffs that developers need to make when choosing a PRVG. Our conclusion is that there is no silver bullet for PRVGs - the right decision depends on the target program, the target platform, and the user-specific constraints, and it is too time consuming for most developers to make.

Modern C++ provides a plethora of PRVGs. Since C++11, the `<random>` header [1] has provided three PRVG templates, each with many degrees of freedom in instantiation. For example, `linear_congruential_engine` and `substract_with_carry_engine` each have 3 template parameters, while `mersenne_twister_engine` has 13. On top of these, the header provides two numerical adaptors that can be composed with any of the generators to provide further transformations to the random values produced. These adaptors are themselves also templated: `discard_block_engine` with 2 parameters, and `shuffle_order_engine` with 1. Should a

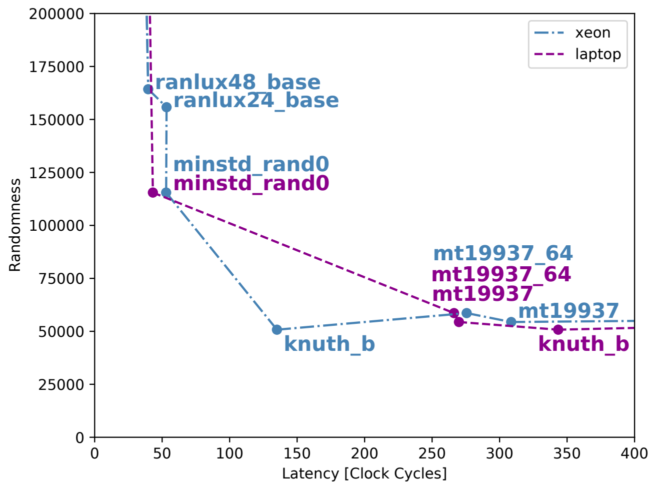

Figure 3. A comparison of the randomness-latency tradeoff space across two different micro-architectures. The blue curve with broader dashes is the same data shown in Figure 1, measured on server-grade hardware. The purple curve with smaller dashes measures randomness vs latency on a Lenovo Thinkpad with a dual-core Intel® Core® i5 CPU and 8 GB DRAM. Several differences are immediately visible between the two platforms, such as the vast latency disparity for the knuth_b generator, but note in particular that both the ranlux24_base and ranlux48_base generators lie on the pareto frontier for the server, whereas neither do for the laptop.

developer desire to compose both of these adaptors with a `mersenne_twister_engine`, on a 64-bit machine, he or she would have $2^{64^{13+2+1}} \approx 1.8e^{308}$ distinct ways of instantiating such a PRVG.

With so many options, programmers find themselves swimming in an ocean of potential tradeoffs, and they therefore currently tend to have a poor understanding of the behavior of these PRVGs. Instead of using the PRVGs supplied by the standard, programmers generally take one of two strategies when they need pseudorandom values. Most programmers still use the old, simple PRVGs provided by C, such as `rand()` and `drand48()`. A few programmers with deeper knowledge of pseudorandomness may opt to write their own custom PRVGs for their specific program. This approach is of course inaccessible to any developer without this knowledge, but even those with the ability to write a custom PRVG often end up writing one with behavior similar to one already provided by the C++ standard. For example, `bodytrack` from the PARSEC benchmark suite comes with a custom PRVG that exhibits behavior nearly identical to C++'s `knuth_b` engine. Regardless of approach, C++ developers seem to lack an understanding of the PRVGs at their disposal.

To gain a better understanding of the PRVG's available in C++, we created a series of simple micro-benchmarks to help evaluating each PRVG along several dimensions. We use these micro-benchmarks to characterize the latency, memory consumption, and quality of randomness of several C++ PRVGs. We perform this analysis across two very different platforms. The results yield a complex tradeoff space, with different behavior across different metrics and no clear, architecture-independent, program-independent winners.

2.1 Micro-benchmarks

To gain a better understanding of the PRVG's available in C++, we created a series of simple micro-benchmarks to help us evaluate each PRVG along several dimensions. Since it would be infeasible to analyze every possible instantiation of each PRVG template, we have decided to focus on the 9 default instantiations provided by the C++ standard. Each micro-benchmark draws 1.1 billion random values using one PRVG - a number large enough to guarantee significant latency values and obscure any noise in latency due to setup and cleanup code in the C++ runtime. We used these micro-benchmarks to measure the latency, memory consumption, and quality of random values produced by each PRVG.

Latency We measured the latency of each PRVG by using the Linux program *perf* to measure the total number of cycles for the program's execution. We divide this number by the total number of pseudorandom values drawn to obtain the mean number of cycles to draw a single value.

Randomness We computed the quality of randomness of the series of PRVGs our micro-benchmarks produce. We borrowed the well known birthday spacings test from Marsaglia's Diehard Battery of Tests of Randomness [36] to evaluate quality of randomness, which hypothesizes that for any bag of random numbers drawn from a large interval, the distances between each pair of points should be exponentially distributed. We calculated these distances and measured the chi-squared goodness of fit of each bag of distances to an exponential curve.

Memory Consumption Finally, to measure memory consumption, we used *massif* from the *valgrind* tool suite [42] to take several snapshots of the working set size of each micro-benchmark across its execution. Predictably, each PRVG had a fairly constant working set size throughout the entire program execution; however, we found that the C++ runtime introduces a brief, significant memory overhead at the startup of any C++ program. Therefore, we used median working set size as our metric for memory consumption, which we found more accurately characterizes the memory profile of each micro-benchmark.

2.2 Platforms

To understand how PRVG behavior varies across micro-architectures, we ran our micro-benchmarks on two different platforms. The first, which we designate as *server*, is a 16-core 16-core Intel® Xeon® CPU with 32 GB DRAM. The second, which we designate as *laptop*, is a Lenovo Thinkpad with a dual-core Intel® Core® i5 CPU and 8 GB DRAM. For further information on each platform, see section 4.1.

2.3 Results

Latency vs Randomness Comparing the latency and randomness metrics that we obtained for each PRVG, we found a rich tradeoff space. We first collected data on our server; plotting randomness vs latency, our PRVG's roughly fit to a pareto frontier. Figure 1 visualizes the inflection point of the pareto curve, where a hypothetical PRVG with 0 latency and perfect randomness would be at the origin. Generally, PRVGs with higher latency have better randomness, while those with lower latency have worse randomness. There is no PRVG that appears to be a clear best choice for both high performance and high quality of random values.

Memory vs Randomness Comparing the memory consumption of each PRVG to its quality of randomness, we found another, different tradeoff space. Figure 2 shows the tradeoff between randomness and working set size on our server. Like the tradeoff between randomness and latency, this tradeoff forms a pareto frontier, where gaining quality of randomness generally requires a sacrifice in memory consumption and vice versa. However, the pareto frontier is clearly different from the randomness-latency frontier. Note that, for example, both the *ranlux48* and the *ranlux24* PRVGs lie very close to the inflection point of this pareto frontier. These PRVGs have, by far, the longest latency of any in C++, and therefore lie nowhere near the inflection point of the randomness-latency frontier. In general, the tradeoff space between memory consumption and randomness differs significantly from the tradeoff space between latency and randomness.

Memory vs Latency Across Architectures Finally, we found that the tradeoff between latency and randomness varies significantly across our two platforms. Figure 3 shows overlays the randomness-latency pareto frontiers generated by both our server and our laptop, showing clear differences. For example, the *ranlux24_base* and *ranlux48_base* PRVGs both lie on the pareto frontier for our server, but neither of them appear on the pareto frontier for our laptop.

Conclusion To summarize, there is a rich tradeoff space between latency and randomness of PRVGs, which varies across different micro-architectures, and which differs greatly from the equally-rich tradeoff space between memory consumption and randomness. With so many potentially conflicting tradeoffs, the decision of the best PRVG to use even for a specific micro-architecture requires an expert in pseudorandomness who knows exactly the desired behavior for a specific program to make an optimal decision. Furthermore, even if a project has such an expert, there is no single PRVG that will be optimal across platforms.

3 The PRV Jeeves Solution

Based on the results of our micro-benchmarks, we argue that compilers should select the PRVGs that a program uses, not developers. To substantiate this claim, we have built the first fully automatic PRVG selector. Our compiler, implemented on top of the LLVM framework [31], takes a standard C++ program as input, analyzes it to find all locations and uses of PRVGs, and replaces each one with an optimal PRVG for its specific use in the program and its behavior on the target micro-architecture. We leverage state-of-the-art autotuning, profiling, and alias analysis to select the best option from an extensible database of PRVGs. Furthermore, we provide a well-founded, custom statistical analysis to verify that the binaries we generate maintain the original level of output quality.

3.1 PRV Jeeves in a Nutshell

PRV Jeeves is a fully automatic compilation flow that takes a generic C++ program and it generates a semantically-equivalent binary for the target platform. The binary is generated by replacing the originally-used PRVGs with the goal of minimizing a cost function (e.g., execution time) provided by a developer. This is performed while maintaining the same output quality of the original code.

To perform the described optimization, PRV Jeeves consists of a feedback loop between LLVM passes, an autotuner, and a profiler. This feedback loop is powered by a PRVG database, and a set of application-specific output distortion evaluators.

The front-end compiler. The input of PRV Jeeves is a set of C++ source files. The front-end compiler of PRV Jeeves translates the source files given as input into a single LLVM bitcode file (IR in Figure 4). Merging all code into a single bitcode file is important to obtain high accuracy from the alias analyses that are later used.

The middle-end compiler. The LLVM bitcode generated by the front-end compiler is consumed by the middle-end compiler. The middle-end compiler analyzes the bitcode for all locations and uses of PRVGs (e.g., a call to `rand()`) and computes the design space of valid PRVG options (e.g., using `drand48()` rather than `rand()`). To do so, alias analyses is used to track the states of the PRVGs used. The output of the middle-end compiler is a new IR file generated by modifying the input IR to make it amenable to selecting different PRVGs. While not fundamental, these IR changes simplified the design of the back-end compiler of PRV Jeeves. Finally, the middle-end compiler generates the design space composed by all PRVG options that can be chosen.

The autotuning loop and the back-end compiler. The rest of the system forms a closed feedback loop. The design space generated by the middle-end compiler becomes the input to our autotuner, which iteratively selects new configurations of PRVGs to test. The back-end compiler takes the

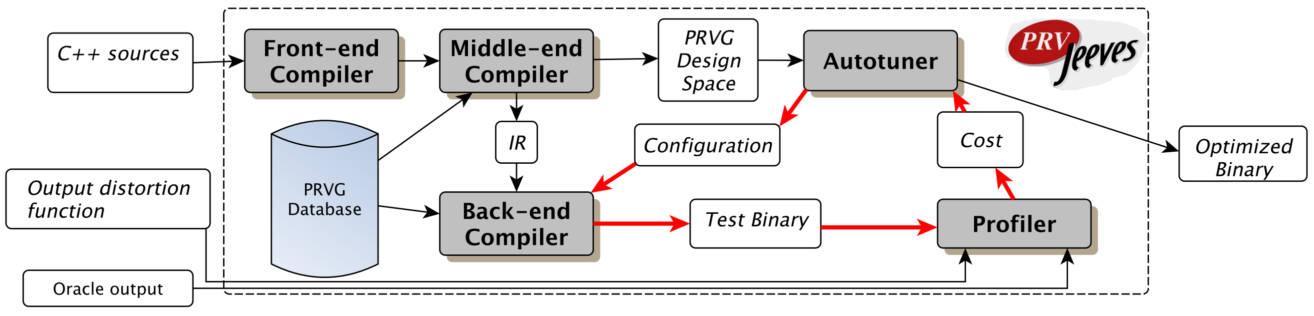

Figure 4. The overall design of the PRV Jeeves compiler. Developers supply a C++ program, along with an application-specific output distortion function and a ground-truth oracle output file. In addition to our middle-end, which analyzes the program for PRVGs, our core contribution is highlighted by the red arrows: a feedback loop between our autotuner, which searches for the best set of PRVGs; our back-end compiler, which instruments the code to use them; and our profiler, which measures the improvement for binary tested. The output is a semantically equivalent but more performant binary.

PRVG locations computed by the middle-end compiler, the IR of the program, and it transforms the code according to the configuration chosen by the autotuner. Then, the back-end compiler generates the binary of the target platform. Finally, PRV Jeeves runs the code several times using training inputs provided by the user. These runs profile the code according to a cost function chosen by developers (e.g., execution time), and we check to see how distorted its output is when compared to an oracle output (provided by developers). The end result is a binary semantically equivalent to the original program, but more performant.

3.2 PRVG Database

Crucial to our design is a finite but extensible database of PRVGs that our compiler can target. Because not all PRVGs can replace all the others, the database is subdivided by both the type of value that the PRVG produces and the underlying distribution that the PRVG draws from. Furthermore, within each of these sub-categories, the list of PRVGs is enumerated. Currently, our design only seeks to characterize the behavior of those PRVGs guaranteed by the standard in C++11 and later versions. Since C++ decouples distribution from engine, we support several distributions, including uniform, normal, and exponential. For each distribution, our database currently contains 14 PRVGs that produce ints and 11 PRVGs that produce doubles. For each of these PRVGs, the database stores an LLVM bitcode file preprepared for integration into an input bitcode program. Users of PRV Jeeves can easily extend our database to also consider other PRVGs.

3.3 Middle-End Compiler

Our middle-end compiler is responsible for the required program analyses. It takes an LLVM bitcode program as input, logs the location of every PRVG allocation and use, and computes the PRVG design space for the autotuner. It consists of two passes: a first pass to compute a conservative version of this design space, and a second pass to shrink it and thereby shorten the search for an optimal configuration.

Pass 1: PRVG Identification This pass of the middle-end compiler takes a program in LLVM bitcode and our PRVG database, identifies the PRVGs used by the code, and it computes the initial design space for the autotuner.

PRVGs need to be identified based on their states. A PRVG is defined by a state (a state is used to generate a new random value and then it is updated), its algorithm (e.g., `knuth_b`), and its value distribution (e.g., uniform distribution). The first task of this pass is to describe each PRVG found in the code in these three dimensions.

This compilation pass is conservative: only PRVGs that can be unambiguously and completely described are logged. Hence, only these PRVGs will be considered in the optimization performed by the autotuner. This pass of the middle-end compiler could miss a PRVG if, for example, its state cannot be detected precisely. This can happen if a pointer of a state is read from memory and our alias analyses cannot define a unique instruction that must have allocated it. This conservativeness is necessary because when a PRVG is replaced, its state allocation needs to be replaced accordingly (different PRVGs have different memory allocators). Hence, this is a safe code transformation only if the information about which instruction have allocated the state is correct.

PRVGs can be divided into reentrant and non-reentrant. Reentrant PRVGs allocate some state and modify this state directly on each function call; in these cases, we log a tuple containing the single allocation of each PRVG object and each use of that object. Non-reentrant PRVGs are stand-alone function calls which modify some global state that is hidden to the programmer (e.g., `rand()`); in these cases, we simply log the line number of each function call in the bitcode. The output of this first pass of the middle-end compiler is this list of unique PRVG locations.

Additionally, this compilation pass generates the design space for our autotuner. We describe the design space as a tuple of dimensions, with one dimension per unique PRVG. To determine the cardinality of each dimension, we determine the type of value that each PRVG produces (e.g. `int`, `float`), and we refer to our database to find the total number of candidate PRVGs to which we can transform it. For example, if a program contains an invocation of `rand()` and no other PRVGs, our first pass identifies such invocation, find that it returns an `int` drawn from a uniform distribution, note that our database currently stores 14 different PRVGs that produce uniform ints, and output a design space of (14).

Pass 2: Invocation Counting The second compilation pass in our middle-end compiler aims to shrink the design space described by the previous pass. It does so by eliminating points of this design space corresponding to PRVGs that are never (or rarely) invoked at run time. Eliminating such points have the only effect of reducing the set of possible options that the autotuner has.

Our front-end compiler generates the IR by invoking `clang` using its code optimizations (`O3`). Doing so, `clang` performs aggressive optimizations such as function inlining and loop peeling. Sometimes these code transformations lead to dead code (a function that is not invoked anymore) or code that is rarely executed.

The autotuner considers each point in the design space when selecting configurations. So, if any PRVGs fall entirely in dead regions of code or in one that is rarely executed, the autotuner's search for the optimal configuration will still waste time trying to find the best PRVG that will never or rarely be used. This observation led us designing the second compilation pass of the middle-end compiler: delete PRVGs that are never or rarely used. Our second compilation pass removes such PRVGs from our design space by instrumenting our input program with counters for each PRVG. It adds one global variable per each PRVG identified by the first pass, increments this variable right after each invocation of each PRVG, and invokes a function right before program exit to dump all these invocation counts to an output file. We compile this modified program to binary and run it to obtain the invocation counts, and if any PRVG is never invoked (or invoked less than 10 times), we remove its corresponding point from the design space and its locations from our log.

The middle-end compiler runs only once the program compiled. Once it has produced the design space to input to the autotuner and the PRVG location log to input to the back-end compiler, the compiler enters the phase at which it spends most of its execution: a feedback loop between the autotuner, the back-end compiler, and the profiler, each of which is described next.

3.4 Autotuner

The autotuner of PRV Jeeves iteratively tries to select the best configuration of the PRVGs identified by the middle-end compiler. This autotuner is built on top of the OpenTuner framework [6]. It takes two inputs, a description of the PRVG design space and a cost function, and it tests various configurations within that design space to try to optimize the cost function. The default cost function is the end-to-end execution time of the compiled program. However, other cost functions are available (e.g., energy consumption, peak memory).

Each time the autotuner is invoked, the design space defines the legal set of configurations it can choose. As previously mentioned, we represent the design space as a tuple of dimensions; each dimension is interpreted by the autotuner as a set of independent switches, such that the selection of one switch does not impact the selection of any other switch. Each time the autotuner is invoked, for each dimension in the design space, it selects a point less than the cardinality of that dimension - in the previous example, if the autotuner sees a design space of (14), it returns a tuple where each point is some value between 0 and 13, such as (4) or (2).

Over iterative invocations of the autotuner, the autotuner uses its own selection history and the cost function to attempt to make better choices. In its first invocation, the autotuner selects a random configuration. Later in the feedback loop, the profiler determines a cost of that execution. The autotuner stores this cost in a database, and, in each subsequent execution, it uses a suite of machine learning to try to predict a configuration that will produce a cost more optimal with respect to the cost function. For example, if a developer chooses minimizing total memory footprint as a cost function and objective, and the autotuner will analyze trends across past runs to predict a configuration with a lower memory footprint than any it has seen before.

The purpose of the autotuner is to accelerate the average time to find the ideal or near-ideal configuration for the design space. The developer specifies the length of time that the compiler executes, and if let run indefinitely, the autotuner will eventually evaluate every point any design space to find the best configuration. This is the only guarantee with respect to optimality; however, our experience has been that it generally takes a small fraction of the total possible configurations for the autotuner to find the optimal selection.

3.5 Back-End Compiler

Once the autotuner has chosen a configuration to test, our back-end compiler transforms each PRVG according to the autotuner's decision and compiles the bitcode program to binary. This compiler interprets each point in the configuration provided by the autotuner as the index of the PRVG in our database within the target data type and distribution. In

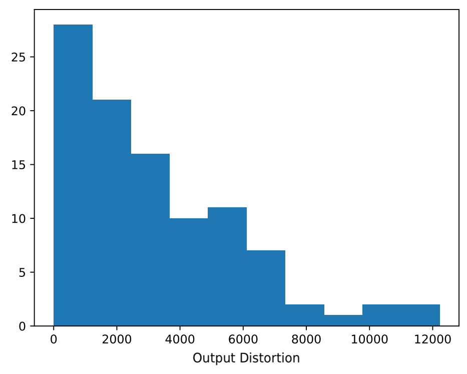

Figure 5. A histogram of output distortion levels of 100 executions of bodytrack. The distribution appears to be exponential in nature, making it hard to draw conclusions about the population of all output distortions from a random sample.

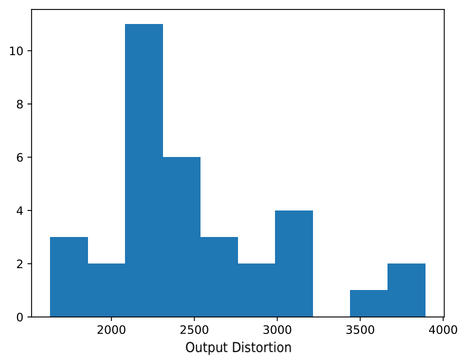

Figure 6. A histogram of the mean output distortions across several samples of 30 executions of bodytrack. Conforming to the central limit theorem, despite the highly non-normal nature of each sample, the means of the samples are roughly normally distributed around the true mean output distortion, unblocking much more insightful statistical analysis.

lock step, the pass iterates over both the unique PRVGs in the location log file produced by the middle end, as well as the points in the configuration. If the PRVG is reentrant, then the pass transforms the program to allocate the PRVG object that the autotuner selected, and it transforms each associated invocation to use that object. If the PRVG is non-reentrant, unless the exact same PRVG has already been selected in the same configuration, the pass allocates the corresponding PRVG object as a global variable, and it transforms that function call to use that object. Should any subsequent points in the configuration correspond to the same PRVG, they all use the same object as the first one. Once all PRVGs have been transformed, the back-end compiler uses an LLVM backend to compile the resulting bitcode to binary.

3.6 Profiler

The profiler runs the binary and computes the cost of the program according to the cost function specified to the autotuner. The profiler is extensible, and the choice of what profiler to use depends on what the programmer wants to optimize. This paper focuses on studying performance, so we use the Linux program *perf* to compute the total number of clock cycles needed to execute the binary. We run each binary several times and try to minimize the total number of cycles across all executions as our cost function. Future users may use any profiler available to them, or potentially several in combination, as long as they can produce some number as a cost that the profiler can report to the autotuner.

3.7 Output Distortion Evaluator

One final, but equally critical component of our system is the output distortion evaluator. This evaluator compares the quality of the output produced by each binary we generate to the quality of the output produced by the baseline program. We reject any binary whose output is more distorted than the baseline program.

For each configuration, the output distortion evaluator determines whether the corresponding program produced acceptable output. Along with each benchmark we target, we provide a file containing a ground-truth set of output values. We also provide an application-specific module that can compare any output from that program to ground-truth and compute a metric representing output quality. Before targeting a new benchmark, we run it many times, unmodified, and we utilize this module to compute the output quality for each of these program executions. We use the mean of these output quality values as our threshold for our compiler.

In order to understand how we compare the output quality of a binary that we produce to output quality of the unmodified program, one must understand how output quality varies. Next we use `bodytrack` as example, but similar results are obtained with all the randomized benchmarks of PARSEC. Figure 5 shows a histogram of the output quality values computed from 100 executions of `bodytrack`, where lower values correspond to higher quality. Output quality tends to fit to an exponential distribution; most of the time, output quality is very high, but fairly frequently, a binary will produce much lower quality output.

Exponential distributions are typically very difficult to characterize and compare. Conceptually, we seek to estimate the output quality we should expect in the next program execution; in an exponential distribution, this is the mean value. However, exponential distributions require extremely large sample sizes to generalize information about the underlying population.

To circumvent this difficulty, we utilize the Central Limit Theorem to gain more insight from our data and decide whether to accept a configuration. The Central Limit Theorem states that for any distribution, there is some minimum number of samples such that, with a large enough sample size, the distribution of the means of all samples is normally distributed. Therefore, in our system, we iteratively collect samples of several executions of our generated binary, and measure the mean output distortion of each of these samples. The sample size is left to the developer, although we have found 30 to be suitable. The system iteratively draws more samples until the means of these samples pass D'Agostino and Pearson's test for normality [18]. The programmer selects the alpha value for the normality test; here, we have used 0.1. Figure 6 shows the histogram of the means from one invocation of our compiler. The resulting near-normal distribution is now amenable to a rich set of statistical inferencing techniques. We utilize a two-sample, one-tailed t-test to evaluate whether the test binary generates a distribution of overall greater output distortion than the baseline. If it is greater, we reject this configuration.

In general, the output distortion evaluator is necessary to ensure that we do not transform any program to use worse-quality PRVGs than it originally uses. However, the real power of the output distortion evaluator stems from the fact that we can often greatly improve output quality by improving the quality of the PRVGs used by a program. In these cases, we can then tune other parameters to bring output quality closer to baseline. These parameters are program specific (e.g., annealing layers of `canneal`). We relied on the same parameters used by the STATS compiler [19]. Sometimes such parameters are more influential to program performance than just the PRVGs, so this process is crucial to the performance improvements we obtain.

To summarize, our compiler takes a C++ source as input, analyzes it for all PRVG uses and definitions, and iteratively tests several possible substitutions for each PRVG until it converges on an optimal solution. The resulting binary we generate produces output with at least as high quality as the original program, but does so more performantly. Next section describes the empirical evaluation of PRV Jeeves.

4 Evaluation

To measure the impact that the choice of PRVG has on real applications, we optimized all the PARSEC benchmarks that are randomized [11]. The original version of one of these benchmarks, `bodytrack`, uses a hand-written, domain specific PRVG, and yet we still observe that by selecting a higher quality PRVG, we decrease their overall execution time *without sacrificing any output quality.* This suggests that even expert knowledge of both PRVGs and a specific application may not be enough to select an optimal PRVG. Furthermore, we observe that different architectures require different PRVGs to produce optimal behavior, further solidifying the need for an automatic, platform-aware PRVG selector. Finally, we observe that the optimal decision differs when trying to optimize for performance, memory consumption, and output quality. Our system easily adapts to whatever criteria the programmer chooses to optimize, saving the programmer from writing a custom PRVG for each situation.

4.1 Experimental Setup

Next we describe the testbed we use for our experiments.

Platforms Our primary platform for evaluation, which we label *server*, consists of 16 Intel® Xeon® X5560 cores spread across 2 sockets, each running at 2.79 GHz. Each core has 32 KB of both private L1d and L1i cache and 256 KB of private L2 cache, each socket shares 8 MB of L3 cache, and the entire machine has 32 GB of DRAM.

Benchmarks Due to their expert implementation, broad community acceptance, and significant use of PRVG's, we benchmark our system on the randomized applications in the PARSEC benchmark suite. Each benchmark is necessarily coupled with an application-specific module to evaluate its output quality. These benchmarks are `bodytrack`, `ferret`, `swaptions`, `canneal`, and `streamcluster` and we used the same output quality functions previously proposed for them [19, 39]. The only benchmark that needs further discussion is `bodytrack` because of its custom PRVG.

Bodytrack takes a video in the form of a sequence of frames as input and identifies where the human body is located within each frame. To do this performantly, it subdivides each frame into several particles. Each particle randomly samples the pixels assigned to it and classifies each of those pixels. At termination, bodytrack outputs a set of vectors, one per frame, each containing the locations of the body in its frame. Bodytrack depends on a single hand-tuned PRVG, which is almost identical to the *knuth_b* generator from C++. To make a valid comparison to the other C++ PRVGs that we generate, we modify bodytrack to use *knuth_b* as its baseline PRVG.

Software Our entire system is built on top of LLVM 5.0.1 and all experiments were run on Linux version 3.10.0. Moreover, we used OpenTuner 0.8 to build our autotuner. Finally, we forced our autotuner to end its search in two hours.

Parameters and Inputs Our compiler presents several parameters that the developer can tune.

For training, we use the simlarge input for every benchmark. When evaluating a configuration, we collect samples of 30 iterations, and we continue to collect samples until our test for the normality of the sample output quality means returns a p-value of at least 0.1, indicating 90% confidence that the underlying distribution is normal. For more precise distributions, the developer should increase the sample size and the critical p-value. After a group of sample means is determined to be normally distributed, we accept a configuration if the t-test comparing its output quality distribution to the baseline returns a p-value of 0.05. Again, increasing the critical p-value will result in greater certainty in the test decision.

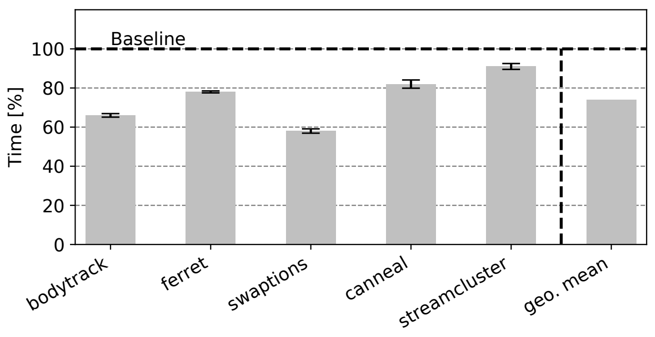

Figure 7. Time saved by PRV Jeeves with respect to the time spent by the original binaries (i.e., compiled with `clang -O3 -march=native`). When supplied with native input, PRV Jeeves binaries use on average only 76% of the baseline time to generate an output of the same quality. This time reduction is obtained automatically.

Once the training phase has selected an optimal configuration, we test this configuration by running it 100 times under the native input. We do the same with the baseline configuration, and we compare the execution times and output qualities of iteration.

4.2 Performance Obtained by PRV Jeeves

Our compiler is able to automatically decrease the execution time of randomized programs without sacrificing their output quality. This is done by choosing PRVGs that better fit the specific needs of a compiled program. On average, we decrease the overall execution time down to 76% of the baseline (`clang -O3 -march=native`) (shown in Figure 7). The source of this speedup is described next.

Randomized programs are often designed with an iterative algorithm in it. This allows them to reach the minimum output quality robustly because such output gets further improved iteration after iteration (e.g., by trying different centroids in a K-cluster benchmark while keeping the best solution in memory: `streamcluster`). In randomized programs, like the PARSEC ones we target, there is a relation between the number of iterations needed to reach the target output quality and the quality of the PRVGs. The better PRVGs, the less iterations are needed. However, a PRVG with important randomness generates highly random values, but it costs in terms of latency and memory consumption. On the other hand, low quality PRVGs generate less quality random values, but they leave a small memory footprint and they are fast. PRV Jeeves finds the best sweet spot between quality of PRVGs, and therefore algorithm iterations, and their costs. This is what generated the time savings shown in Figure 7.

To further measure this relation between algorithm iterations and time saved by PRV Jeeves, Figure 8 shows the reduction in algorithm iterations for the target benchmarks. It is interesting to notice `swaptions`. This benchmark sees the biggest reduction of algorithm iterations because of the choice of a PRVG that generates high quality random values (`knuth_b`). However, because the latency of this PRVG is much higher than the one of the baseline, the time saving reduction shown in Figure 7, while significant, is less than the algorithm iteration reduction.

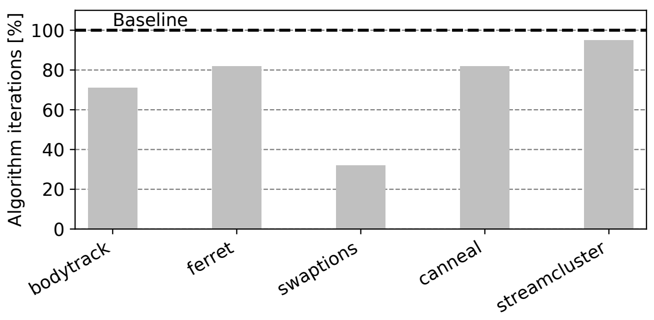

Figure 8. Reduction obtained by PRV Jeeves to the number of algorithm iterations needed to converge to the same output quality of the baseline.

Finally, it is important to mention that PRV Jeeves chose different PRVGs for different benchmarks. So a simple static solution that, for example, picks always the PRVG with the highest amount of randomness is sub-optimal. For example, PRV Jeeves selected the *ranlux48_base* PRVG for `bodytrack`. Recalling Figure 1, this was the fastest PRVG on the randomness-latency pareto frontier and near the inflection point. On the other hand, PRV Jeeves selected the `knuth_b` PRVG for `swaptions`, which is a more balanced tradeoff between randomness and latency.

4.3 Output Quality

To preserve the original output quality, PRV Jeeves implements the tests described in Section 3.7. These tests are used during the autotuning loop to discard solutions that lead to less quality outputs.

To evaluate this aspect of PRV Jeeves, we measure the output quality of each benchmark over 100 runs. We do this for two binaries for each benchmark: the one generated by the baseline (i.e., `clang -O3 -march=native`) and the one generated by PRV Jeeves. Notice that each run of a binary generates a different output because of the randomness of the compiled program. It is important to understand the distribution of these outputs and whether PRV Jeeves changed such distribution.

Figure 9 shows the box plots of the output qualities of 100 iterations of both the baseline and our optimized version of each benchmark. Each output is compared against the oracle output, which defines the 100% output quality. The differences between the distributions of output qualities between the baseline and PRV Jeeves are too small. In other words, a t-test was unable to determine any significant difference in the overall distributions of the output distortions.

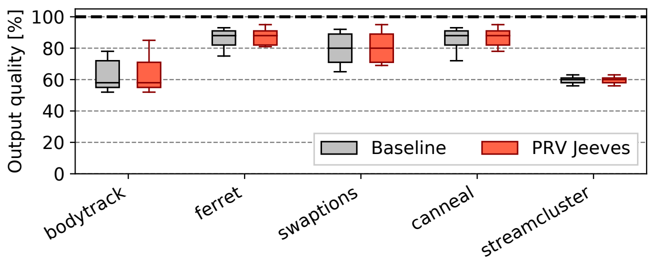

Figure 9. A comparison of the qualities of output produced by the original, baseline binaries and by the ones generated by PRV Jeeves. The two distributions are not significantly different.

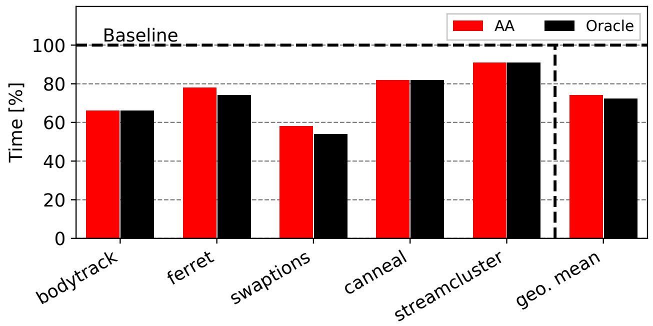

Figure 10. Impact of improving alias analysis in PRV Jeeves. Current alias analyses are good enough to reach most opportunities to select better PRVGs.

4.4 Impact of Alias Analyses

PRV Jeeves relies on the alias analyses included in the latest HELIX [14, 41] compiler to identify the states of the PRVGs used by the input program. Our results were 2-15% worst when the alias analysis of the earlier HELIX compiler [15, 16] was used. Low accuracy in alias analyses translates into less PRVGs identified and, therefore, less PRVGs targeted by PRV Jeeves. Less PRVG targeted means less opportunities to reduce the execution time of a program. To evaluate this impact, we run PRV Jeeves substituting the alias analyses with the oracle information. We generated such oracle with a tool built in house, which is similar to [29].

The accuracy of the alias analyses used is good enough for the benchmarks we targeted. This is shown by Figure 10. Even a potentially overestimate of the best possible alias analysis accuracy (i.e., oracle) does not increase significantly the time saved by PRV Jeeves.

5 Related Work

Although this work provides the first deep study into PRVG tradeoffs and the first automatic PRVG selector, PRV Jeeves is indebted to various prior work in pseudorandomness, autotuning, and instruction selection. We detail each as follows.

Pseudorandomness. PRV Jeeves can target several existing random value generators. C++ specifically draws on influential research to use three basic classes of PRVGs: linear congruential engines [35], mersenne twister engines [37], and subtract with carry engines, which are a subset of a more general class of PRVGs known as generalized Fibonacci generators [25]. The same work that presented generalized Fibonacci PRVGs also motivated the use for one of C++'s adaptors, the discard block engine. The other adaptor, the shuffle order engine, is drawn from an extensive study into numerical techniques [53]

For evaluating PRVGs, the computing community is deeply indebted to the seminal work of Geroge Marsaglia [36]. The tests developed by Marsaglia are still the standard barometers for PRVG quality, and this work specifically borrows the birthday spacings test from the Diehard suite.

At a more meta-level, randomness is completely essential to countless sub-domains of computing, including but not limited to information theory [50], machine learning and data analytics [26, 27, 40, 43, 46, 47, 49], computer vision [20, 21, 51], and quantum computing [30, 33, 34]. Furthermore, although much of the work in cryptography has focused on true randomness, PRVGs are still prevalent in cryptographic applications [12, 13, 32, 48]. Randomness in both security and consensus algorithms have significant implications to large-scale distributed systems, such as the Internet of Things [8, 9, 28, 38, 44].

Autotuning. PRV Jeeves relies on an ad-hoc autotuner to accelerate the search for good configurations of PRVGs. Our autotuner is built upon the OpenTuner framework [6]. This is similarly done by other projects [5, 7, 19, 45, 52].

Instruction Selection. One of the most important tasks for any compiler is the optimal selection of instructions to generate in the target language, subject to the semantic of the original code. PRV Jeeves can be conceptualized as a specific type of instruction selector - we select the best PRVG instructions. However, more general instruction selection has been an active and influential topic of research within the compiler community since its inception [3, 4, 10, 17, 22–24].

6 Conclusion

Existing programs rarely get the most out of their pseudorandom value generators. We have presented the first in-depth study of PRVG's in C++ and the first compiler with a fully automatic PRVG selector. Through PRV Jeeves, we have demonstrated that migrating the choice of which PRVGs to use to a compiler comes with no cost; developers can stop thinking about PRVGs entirely and get significant performance improvements.

Acknowledgments

This project is made possible by support from the United States National Science Foundation via the grant CNS1753743.

References

[1] [n.d.]. <random> - C++ Reference. http://www.cplusplus.com/reference/random/

[2] [n.d.]. Random Number Generation in C++11. https://isocpp.org/files/papers/n3551.pdf

[3] A V Ah and S C Johnson. [n.d.]. Optimal Code Generation for Expression Trees. ([n. d.]), 14.

[4] Alfred V. Aho, Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. 1989. Code Generation Using Tree Matching and Dynamic Programming. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989), 491–516. https://doi.org/10.1145/69558.75700

[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. [n.d.]. PetaBricks: A Language and Compiler for Algorithmic Choice. ([n. d.]), 12.

[6] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 303–315. https://doi.org/10.1145/2628071.2628092

[7] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*. 85–96. https://doi.org/10.1109/CGO.2011.5764677

[8] James Aspnes. 2003. Randomized protocols for asynchronous consensus. *Distributed Computing* 16, 2 (Sept. 2003), 165–175. https://doi.org/10.1007/s00446-002-0081-5

[9] James Aspnes and Maurice Herlihy. 1990. Fast randomized consensus using shared memory. *Journal of Algorithms* 11, 3 (Sept. 1990), 441–461. https://doi.org/10.1016/0196-6774(90)90021-6

[10] A. Balachandran, D. M. Dhamdhere, and S. Biswas. 1990. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages* 15, 3 (Jan. 1990), 127–140. https://doi.org/10.1016/0096-0551(90)90006-B

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*. ACM Press, Toronto, Ontario, Canada, 72. https://doi.org/10.1145/1454115.1454128

[12] L. Blum, M. Blum, and M. Shub. 1986. A Simple Unpredictable Pseudo-Random Number Generator. *SIAM J. Comput.* 15, 2 (May 1986), 364–383. https://doi.org/10.1137/0215025

[13] M. Blum and S. Micali. 1984. How to Generate Cryptographically Strong Sequences of Pseudorandom Bits. *SIAM J. Comput.* 13, 4 (Nov. 1984), 850–864. https://doi.org/10.1137/0213053

[14] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. 2014. HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 217–228. http://dl.acm.org/citation.cfm?id=2665671.2665705

[15] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. 2012. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 84–93. https://doi.org/10.1145/2259016.2259028

[16] Simone Campanoni, Timothy Jones, Glenn Holloway, Gu. Y. Wei, and David Brooks. 2012. The HELIX project: Overview and directions. In *DAC Design Automation Conference 2012*. 277–282. https://doi.org/10.1145/2228360.2228412

[17] D. R. Chase. 1987. An Improvement to Bottom-up Tree Pattern Matching. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 168–177. https://doi.org/10.1145/41625.41640 event-place: Munich, West Germany.

[18] Ralph B. D'Agostino. 1971. An Omnibus Test of Normality for Moderate and Large Size Samples. *Biometrika* 58, 2 (1971), 341–348. https://doi.org/10.2307/2334522

[19] Enrico A. Deiana, Vincent St-Amour, Peter A. Dinda, Nikos Hardavellas, and Simone Campanoni. 2018. Unconventional Parallelization of Nondeterministic Applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 432–447. https://doi.org/10.1145/3173162.3173181

[20] J. Deutscher, A. Blake, and I. Reid. 2000. Articulated body motion capture by annealed particle filtering. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No.PR00662)*, Vol. 2. 126–133 vol.2. https://doi.org/10.1109/CVPR.2000.854758

[21] J. Deutscher, B. North, B. Bascle, and A. Blake. 1999. Tracking through singularities and discontinuities by random sampling. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, Vol. 2. 1144–1149 vol.2. https://doi.org/10.1109/ICCV.1999.790409

[22] Yuanbo Fan, Simone Campanoni, and Russ Joseph. 2019. Time squeezing for tiny devices. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. 657–670. https://doi.org/10.1145/3307650.3322268

[23] Yuanbo Fan, Tianyu Jia, Jie Gu, Simone Campanoni, and Russ Joseph. 2018. Compiler-guided Instruction-level Clock Scheduling for Timing Speculative Processors. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. ACM, New York, NY, USA, Article 40, 6 pages. https://doi.org/10.1145/3195970.3196013

[24] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. 1992. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems* 1, 3 (Sept. 1992), 213–226. https://doi.org/10.1145/151640.151642

[25] Bert F. Green, J. E. Keith Smith, and Laura Klem. 1959. Empirical Tests of an Additive Random Number Generator. *J. ACM* 6, 4 (Oct. 1959), 527–537. https://doi.org/10.1145/320998.321006

[26] Thomas L Griffiths and Joshua B Tenenbaum. [n.d.]. From Algorithmic to Subjective Randomness. ([n. d.]), 8.

[27] Thomas L Griffths and Joshua B Tenenbaum. [n.d.]. Probability, algorithmic complexity, and subjective randomness. ([n. d.]), 6.

[28] Haowen Chan, A. Perrig, and D. Song. 2003. Random key predistribution schemes for sensor networks. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. IEEE Comput. Soc, Berkeley, CA, USA, 197–213. https://doi.org/10.1109/SECPRI.2003.1199337

[29] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. 2010. SD3: A scalable approach to dynamic data-dependence profiling. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 535–546.

[30] E. Knill, D. Leibfried, R. Reichle, J. Britton, R. B. Blakestad, J. D. Jost, C. Langer, R. Ozeri, S. Seidelin, and D. J. Wineland. 2008. Randomized benchmarking of quantum gates. *Physical Review A* 77, 1 (Jan. 2008), 012307. https://doi.org/10.1103/PhysRevA.77.012307

[31] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, San Jose, CA, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[32] M. Luby and C. Rackoff. 1988. How to Construct Pseudorandom Permutations from Pseudorandom Functions. *SIAM J. Comput.* 17, 2 (April 1988), 373–386. https://doi.org/10.1137/0217022

[33] Easwar Magesan, J. M. Gambetta, and Joseph Emerson. 2011. Scalable and Robust Randomized Benchmarking of Quantum Processes. *Physical Review Letters* 106, 18 (May 2011), 180504. https://doi.org/10.1103/PhysRevLett.106.180504

[34] Easwar Magesan, Jay M. Gambetta, B. R. Johnson, Colm A. Ryan, Jerry M. Chow, Seth T. Merkel, Marcus P. da Silva, George A. Keefe,

Mary B. Rothwell, Thomas A. Ohki, Mark B. Ketchen, and M. Steffen. 2012. Efficient Measurement of Quantum Gate Error by Interleaved Randomized Benchmarking. *Physical Review Letters* 109, 8 (Aug. 2012), 080505. https://doi.org/10.1103/PhysRevLett.109.080505

[35] GEORGE Marsaglia. 1972. The Structure of Linear Congruential Sequences. In *Applications of Number Theory to Numerical Analysis*, S. K. Zaremba (Ed.). Academic Press, 249–285. https://doi.org/10.1016/B978-0-12-775950-0.50013-3

[36] G. MARSAGLIA. 2008. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. *http://www.stat.fsu.edu/pub/diehard/* (2008). https://ci.nii.ac.jp/naid/10025030014/

[37] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation* 8, 1 (Jan. 1998), 3–30. https://doi.org/10.1145/272991.272995

[38] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399 event-place: Vienna, Austria.

[39] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of Service Profiling. In *International Conference on Software Engineering (ICSE)*.

[40] Frank Moosmann, B Triggs, and Frederic Jurie. 2006. Randomized Clustering Forests for Building Fast and Discriminative Visual Vocabularies. *Neural Information Processing Systems* (Jan. 2006).

[41] Niall Murphy, Timothy Jones, Robert Mullins, and Simone Campanoni. 2016. Performance Implications of Transient Loop-carried Data Dependences in Automatically Parallelized Loops. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 23–33. https://doi.org/10.1145/2892208.2892214

[42] Nicholas Nethercote and Julian Seward. [n.d.]. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. ([n. d.]), 12.

[43] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. 2002. Streaming-data algorithms for high-quality clustering. In *Proceedings 18th International Conference on Data Engineering*. IEEE Comput. Soc, San Jose, CA, USA, 685–694. https://doi.org/10.1109/ICDE.2002.994785

[44] Miyako Ohkubo, Koutarou Suzuki, and Shingo Kinoshita. [n.d.]. Cryptographic Approach to "Privacy-Friendly" Tags. ([n. d.]), 17.

[45] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. [n.d.]. Portable Performance on Heterogeneous Architectures. ([n. d.]), 13.

[46] Ali Rahimi and Ben Recht. [n.d.]. Random Features for Large-Scale Kernel Machines. ([n. d.]), 8.

[47] Ali Rahimi and Benjamin Recht. [n.d.]. Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning. ([n. d.]), 8.

[48] Andrew Rukhin, Juan Soto, James Nechvatal, Elaine Barker, Stefan Leigh, Mark Levenson, David Banks, Alan Heckert, and James Dray. [n.d.]. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. ([n. d.]), 131.

[49] Andrew M Saxe, Pang Wei Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, and Andrew Y Ng. [n.d.]. On Random Weights and Unsupervised Feature Learning. ([n. d.]), 9.

[50] Claude E. Shannon. 1957. Certain results in coding theory for noisy channels. *Information and Control* 1, 1 (Sept. 1957), 6–25. https://doi.org/10.1016/S0019-9958(57)90039-6

[51] Hedvig Sidenbladh, Michael J. Black, and David J. Fleet. 2000. Stochastic Tracking of 3D Human Figures Using 2D Image Motion. In *Proceedings of the 6th European Conference on Computer Vision-Part II (ECCV '00)*. Springer-Verlag, London, UK, UK, 702–718. http://dl.acm.org.turing.library.northwestern.edu/citation.cfm?id=645314.649449

[52] C. Tapus, I-Hsin Chung, and J. K. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 44–44. https://doi.org/10.1109/SC.2002.10062

[53] Joab R Winkler. 1993. Numerical recipes in C: The art of scientific computing, second edition. *Endeavour* 17, 4 (Jan. 1993), 201. https://doi.org/10.1016/0160-9327(93)90069-F