Why Do Programs Have Heavy Tails?

Hiroshi Sasaki* Fang-Hsiang Su* Teruo Tanimoto† Simha Sethumadhavan*

*Department of Computer Science, Columbia University
†Graduate School of Information Science and Electrical Engineering, Kyushu University
*{sasaki,mikefhsu,simha}@cs.columbia.edu
†teruo.tanimoto@cpc.ait.kyushu-u.ac.jp

***Abstract*—Designing and optimizing computer systems require deep understanding of the underlying system. Historically many important observations that led to the development of essential hardware and software optimizations were driven by empirical studies of program behavior. In this paper we report an interesting property of dynamic program execution by viewing it as a changing (or social) network. In a program social network, two instructions are friends if there is a producer-consumer relationship between them. One prominent result is that the outdegree of instructions follow heavy tails or power law distributions, i.e., a few instructions produce values for many instructions while most instructions do so for very few instructions. In other words, the number of instruction dependencies is highly skewed.**

In this paper we investigate this curious phenomenon. By analyzing a large set of workloads under different compilers, compilation options, ISAs and inputs we find that the dependence skew is widespread, suggesting that it is fundamental. We also observe that the skew is fractal across time and space. Finally, we describe conditions under which skew emerges within programs and provide evidence that suggests that the heavy-tailed distributions are a unique program property.

I. Introduction

The discovery of common and widely employed program properties such as the existence of hot code regions, presence of spatial and temporal locality, repeated control flow and dependence patterns have had an indelible impact on computer systems. Each one of these fundamental broadly applicable observations has been leveraged to improve computer systems in numerous ways.

Recently, Sasaki et al. revealed another undiscovered program property, the presence of dependence skew in programs [13]. They observed that most instructions in a program produce values for a few instructions while a handful of instructions produce outputs for an inordinate number of instructions. This observation is borne out of their view of program execution as a social or information network.

In a program social network two nodes (instructions) are considered friends if they have a producer-consumer relationship. In other words the source node is an "influencer" and the destination node is an influencee or a "follower". With this network view they observe that the number of followers of an instruction obeys a heavy-tailed distribution, and in majority of cases a specific class of the distribution, the power law distribution.

What is the importance of this observation? Broadly speaking, the presence of skew is interesting for two reasons: first, power law and heavy-tailed distributions have been previously reported to occur in many natural and artificial, social and scientific phenomena, including human social networks, and it is surprising that they appear in programs. As such, many classical modeling results and analysis techniques in these fields may be directly used to improve computer systems modeling. Second, and perhaps more importantly, power laws and heavy tails in programs indicate the possibility of intervention on small part of a program (high outdegree instructions) to enable systematic large scale change. This asymmetry makes it a prime candidate for microarchitectural exploitation with low investment.

Before the phenomenon can be exploited, however, we need to understand it thoroughly. In this paper we provide a mechanistic reasoning for the presence of heavy-tailed distributions in program social networks. We conduct a series of controlled experiments to investigate what causes the emergence of heavy tails: is it for particular classes of programs? Is it due to specific compilers or compiler optimizations? Or is it due to specific instruction set features such as the use of two operands for most instructions? How does the behavior of the network change over time, or at the granularity of functions?

Our results show that heavy tails are broadly seen in a large class of programs (10 SPEC Int, 11 SPEC FP, 8 CortexSuite, 7 DaCapo, SQLite and V8), and are largely agnostic to the programming language (C, C++, Fortran and Java), the compiler (LLVM and GCC) and its optimization level (`-O0` to `-O3`), the instruction set architecture (variable operand or stack-based), and across input types and sizes. We also observe that heavy tails are fractal across time and space.

These results indicate that heavy tail pattern is a fundamental feature of how programs are written. We further demonstrate that heavy tails emerge when functions that compose the program have heavy tails by developing a synthetic model. We also conduct source code analysis to uncover the existence of high outdegree instructions. Finally we provide some evidence that dependence skew may lead to new architectures by clustering programs based on the outdegree distributions, and comparing those clusters to (micro)architectural metric based clustering results.

II. Methodology

Network Construction: in a program social network, each node (or vertex) is a *static* instruction and each directed

edge represents *dynamic* dependency, either control or data dependency. Multiple dynamic dependencies between a pair of instructions are represented by a single edge. We define a control dependence as a connection between a source instruction and its immediate successive destination instruction. A data dependence is defined as a producer-consumer relationship. Specifically, when a source instruction writes a value and the following destination instruction reads the value before it gets overwritten. For the sake of brevity, we generate a network from the main thread of program execution where most of the programs analyzed are single-threaded benchmarks. We run our analysis on SPEC CPU2006 benchmarks, CortexSuite, DaCapo suite, and two commercial programs: SQLite and V8. While many analyses can be performed on program social networks, in this paper we focus on outdegrees (number of outgoing edges) of instructions to understand how instructions influence the whole program execution. The dependence skew is the distribution of the outdegrees.

Background on Distributions: formally speaking, heavy-tailed distributions are probability distributions whose tails are not exponentially bounded. Roughly speaking, this means that the distribution is not random and more importantly there are some heavy skews in the distribution. Power law distribution is a specific and popular subclass of heavy-tailed distributions. A distribution obeys a power law if it is drawn from a probability distribution $p(x) \propto x^{-\alpha}$. A power law distribution has two parameters: the scaling factor α and the minimum value $x_{\min}$. The α controls how sharply the probability decreases (the smaller the α the heavier the tail), and the $x_{\min}$ decides where the heavy *tail* of the distribution begins. Readers who wish to skim may skip rest of this section without loss of continuity.

Testing Method: we apply a standard statistical testing procedure to verify if the distributions align with power laws and heavy tails [3]. This procedure has three steps: (1) estimate the parameters (α and $x_{\min}$) of the power law model, (2) perform a goodness-of-fit test to obtain a p-value, where a value greater than 0.1 indicates that the power law is a plausible hypothesis for the distribution, otherwise it is rejected, and (3) compare the power law against other distributions via a likelihood ratio test to see if there exist better alternatives than the power law. Also we test whether the distribution is a heavy-tailed distribution or not in this step.

In the first step the α is estimated by the maximum likelihood method. To find an estimate for the standard error on $\hat{\alpha}$, we make a quadratic approximation to the log-likelihood at its maximum and take the standard deviation of the resulting Gaussian form for the likelihood as our error estimate. In order to estimate the correct $x_{\min}$, we choose the $\hat{x}_{\min}$ value that makes the probability distributions of the empirical data and the best-fit power law model as similar as possible above $\hat{x}_{\min}$. We use the Kolmogorov-Smirnov or KS statistic which is commonly used for non-normal dataset to quantify the distance between two probability distributions. KS statistic is the maximum distance between the CDFs (cumulative distribution functions) of the empirical data and the fitted model. Readers who are interested in the mathematical details of the procedure are referred to relevant papers [1, 3].

Second step is a Monte Carlo procedure which synthesizes testing datasets by the estimated α and $x_{\min}$. We fit each synthetic dataset to its *own* power law model and count what fraction of the time the model is a poorer fit. This fraction becomes our p-value and hence higher values indicate that the empirical dataset is more likely to fit power law. If the p-value ≥ 0.1, we consider the power law is a plausible fit. In other words, our null hypothesis is: the empirical data and the synthetic data are both drawn from the same distribution. Higher p-value (≥ 0.1) means that there is statistically insufficient evidence to distinguish the empirical data from the synthetic data drawn from a power law distribution. We compare our empirical dataset with 5,000 synthetic datasets.

Finally, in the third step we see whether alternative distributions, lognormal and exponential distributions, are better fits via a statistical likelihood ratio test. If either power law or lognormal distribution is favored over the exponential distribution, we conclude that the distribution is a heavy-tailed distribution since lognormal distribution is another heavy-tailed distribution while exponential distribution is not.

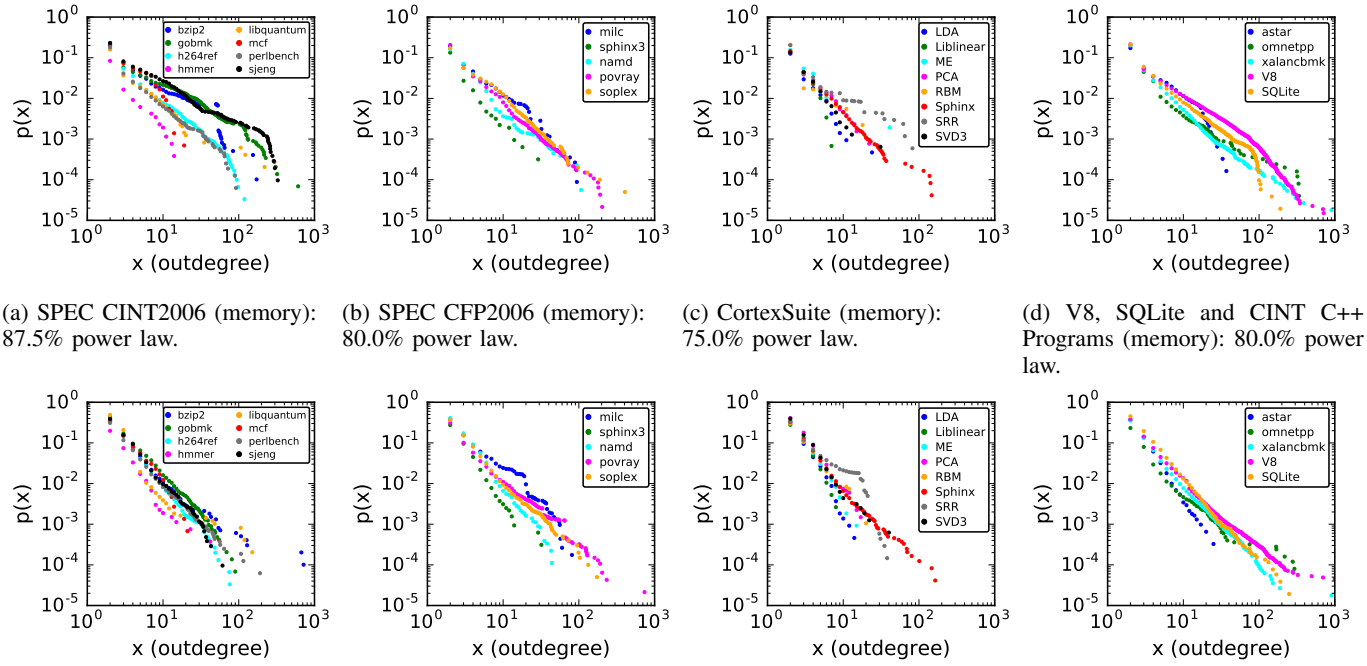
III. Heavy Tails in Program Structure

We analyze the outdegree distribution of programs from the SPEC CPU2006 benchmark suite [7], CortexSuite [14] — a collection of more simple computation kernels — and also Chrome V8 JavaScript engine [6] and SQLite [11] as examples from real-world programs. With CortexSuite, SPEC and the large scale programs we aim to cover the spectrum of programs from kernels to in-the-wild programs.

We construct program social networks at the x86–64 ISA level. For this purpose, we dynamically generate the network using Pin [9]. Since the machine instructions communicate with each other by means of register files and memory, we generate and analyze two sets of networks per benchmark: one having only register dependencies as data flow edges and the other having only memory dependencies as data flow edges. Both networks contain all control edges. Also, we construct the networks using only the instructions in the main program binary since we are interested in characterizing the pure communication behavior of the programs. In other words, all the nodes and edges that account for other binary images (e.g., shared library) are not recorded, unless otherwise specified.

All the programs are compiled using LLVM 3.3 with `-O3` optimization except V8 and SQLite which are compiled using GCC 4.8 with `-O2` optimization. We use the `test` and `small` inputs for SPEC CPU2006 and CortexSuite, respectively, and for V8, we use the `crypto` input which comes with the V8 source code under the "benchmarks" directory. For SQLite, "speedtest1.c", which is a program to estimate the performance of SQLite under a typical workload, is used.

Fig. 1 demonstrates the results of all the benchmarks. Each figure represents the complementary cumulative distribution



(a) SPEC CINT2006 (memory): 87.5% power law.

(b) SPEC CFP2006 (memory): 80.0% power law.

(c) CortexSuite (memory): 75.0% power law.

(d) V8, SQLite and CINT C++ Programs (memory): 80.0% power law.

(e) SPEC CINT2006 (register): 75.0% power law.

(f) SPEC CFP2006 (register): 40.0% power law.

(g) CortexSuite (register): 87.5% power law.

(h) V8, SQLite and CINT C++ Programs (register): 60.0% power law.

Fig. 1: Outdegree-based log-log plot of the CCDF for the communication networks for various benchmark suites compiled with LLVM `-O3`. V8 and SQLite are compiled with GCC `-O2`. Both memory and register networks are shown. CINT2006 programs written in C++ are shown with V8 and SQLite for better visibility. Percentage of programs that follow power laws are also presented. All the results follow heavy-tailed distributions.

function (CCDF) on doubly logarithmic axes, where x represents the outdegree and $p(x)$ represents the complementary cumulative probability. Figures 1(a) to 1(d) show the CCDF of memory-based networks for SPEC CINT2006, SPEC CFP2006, CortexSuite, and V8 and SQLite, respectively. Figures 1(e) to 1(h) show the CCDF of register-based networks for the same set of programs. CINT2006 programs written in C++ (`astar`, `omnetpp` and `xalancbmk`) are shown together with V8 and SQLite to make the figures legible.

Interestingly, we can see similar trends across programs. By performing the analysis described in Section II, we find that all the benchmarks have heavy tails for both memory- and register-based networks. Also, a majority of them follow power laws. Program social networks follow heavy-tailed distributions, and it is likely that the phenomenon is not unique to any type of programs.

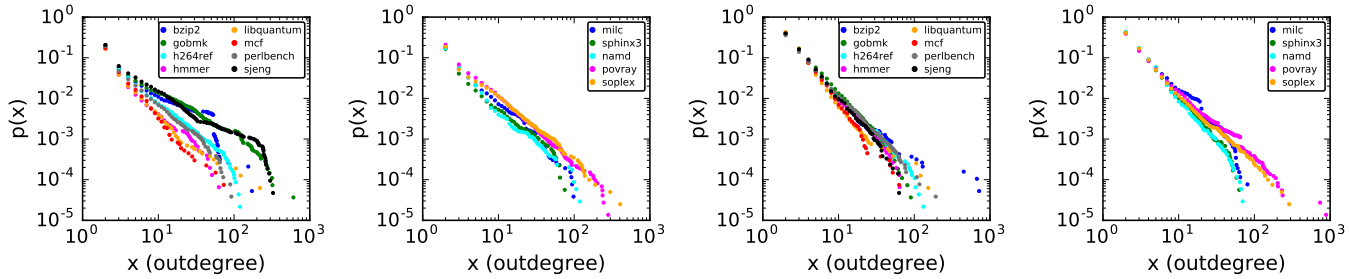
At this point, however, we have exercised only a single pair of compiler and its optimization level per benchmark, and also the underlying ISA (x86–64) is fixed. Since the compiler and the ISA act as a filter for the programs in generating their communication networks, we would like to ask if either or both of them are root causes of heavy tails. We explore them in the following sections, but before delving into these details, we first explore whether ignoring the nodes and edges from other binary images affect our results or not in the next section.

IV. Are Heavy Tails Due to Excluding Shared Libraries?

Although our purpose is to analyze the communication behavior of instructions in the main program, we would like to know whether including instructions in other binary images such as shared libraries break the heavy-tailed distributions or not. Fig. 2 presents the results of the SPEC CPU2006 benchmarks including instructions in shared libraries. The compiler and its optimization level are the same with that of the previous section (LLVM with `-O3` optimization). We can clearly see that the distributions with and without shared libraries are highly similar (e.g., Figures 2(a) vs. 1(a)), and we observed that the distributions with shared libraries also have heavy tails. In the rest of the paper we only present the results of instructions in main programs, although we believe that the observations and findings do apply even when we include the instructions in other binary images.

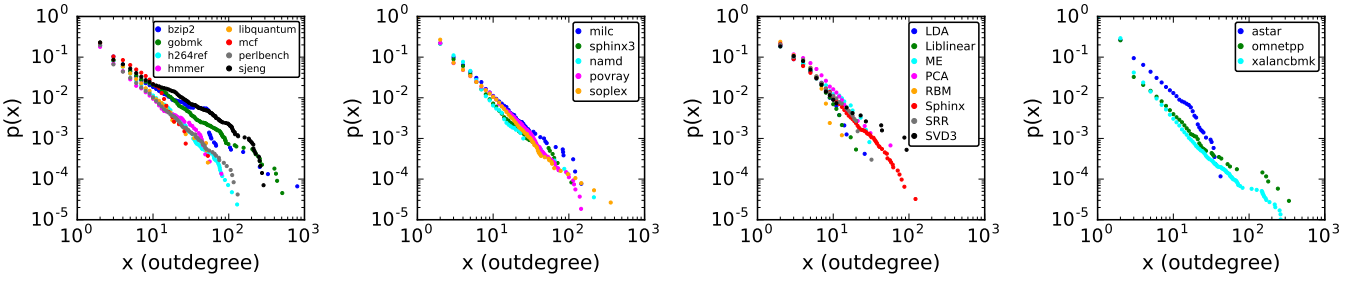
V. Are Heavy Tails Due to Compilation?

The choice of a compiler and its optimizations have a potential to greatly affect the outdegree distributions of program networks. For instance, a compiler can reduce (e.g., eliminating redundancy) or increase (e.g., unrolling) the number of nodes (static instructions) in the network. Similarly, different register allocation algorithms can affect the register communication



(a) SPEC CINT2006 (memory). (b) SPEC CFP2006 (memory). (c) SPEC CINT2006 (register). (d) SPEC CFP2006 (register).

Fig. 2: Outdegree-based log-log plot of the CCDF for the communication networks *including the shard libraries* for SPEC benchmark suites compiled with LLVM `-O3`.

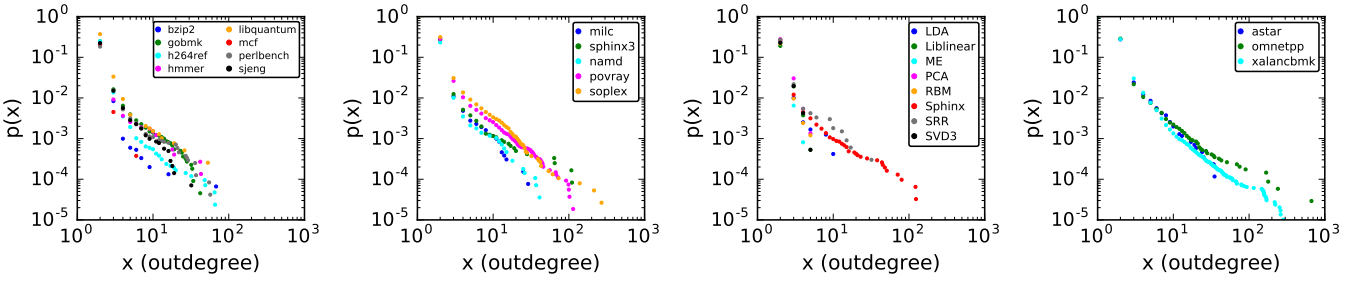


(a) SPEC CINT2006 (memory): 40.0% power law.

(b) SPEC CFP2006 (memory): 80.0% power law.

(c) CortexSuite (memory): 100% power law.

(d) V8, SQLite and CINT C++ Programs (memory): 100% power law.



(e) SPEC CINT2006 (register): 20.0% power law.

(f) SPEC CFP2006 (register): 100% power law.

(g) CortexSuite (register): 100% power law.

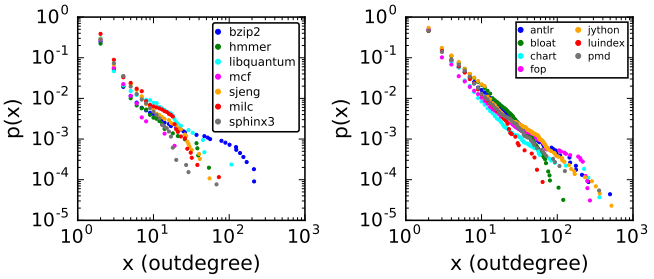
(h) V8, SQLite and CINT C++ Programs (register): 66.7% power law.

Fig. 3: Outdegree-based log-log plot of the CCDF for the communication networks for various benchmark suites compiled with LLVM `-O0` optimization. V8 and SQLite are not shown. All the results follow heavy-tailed distributions.

patterns. It is too onerous to exhaustively list all combinations of compiler optimizations and qualitatively describe how each of them affects the network. Instead, we quantitatively evaluate if a specific compiler and/or its optimization level affects the distribution of the networks.

For this purpose, we perform the same evaluations with the previous section using binaries compiled with LLVM (`-O0` to `-O2`) and GCC (`-O0` to `-O3`). Since we found that LLVM with `-O1` and `-O2` optimizations have fairly similar results with that of Section III (LLVM with `-O3`), we only demonstrate the results with `-O0` optimization. Also, we do not show the results using GCC (`-O0` to `-O3` optimizations) since the results are consistent with those of LLVM.

Fig. 3 presents the results for all the programs compiled with LLVM `-O0`. For the memory-based networks, the results look very similar to that of `-O3` optimization. On the other hand, for the register networks, we can see that the distributions of `-O3` optimization tend to have heavier tails. This is somewhat expected as longer register lifetimes and aggressive unrolling at this optimization level may increase the number of consumers. Nevertheless, we observed that the distribution of all the programs follow heavy tails for both memory- and register-based networks. We have shown that the choice of the compiler and its optimization level are not the root cause of heavy-tailed distributions, as both LLVM and GCC generate binaries with similar distributions that have heavy tails.



(a) SPEC with LLVM IR. (b) DaCapo with Java bytecode.

Fig. 4: Outdegree-based log-log plot of the CCDF of the program networks for SPEC with LLVM IR network and DaCapo with Java bytecode network. All the results follow heavy-tailed distributions.

One thing we observed is that the compiler optimizations have a moderate impact on the register-based networks when the optimization level is increased from `-O0`. We hypothesize that this is due to the fact that register-based networks are impacted by the limited number of architectural registers, and consequently the register allocation. On the other hand, for memory-based networks the compiler optimization has less impact. This can be partially due to the fact that memory-based network only reflects true data dependencies among instructions. Also data dependencies tend to span across multiple basic blocks which makes the compiler difficult to statically analyze and optimize the code.

VI. Are Heavy Tails Due to the ISA?

As we have seen in the previous section that the register-based networks are more susceptible to compiler optimizations, one might be interested in how the x86–64, a CISC ISA with a relatively small number of architectural registers affect the outdegree distributions. In order to investigate whether heavy tails are due to these restrictions or not, we use the compiler intermediate representation LLVM IR to construct program networks. The LLVM IR is in SSA form where the instructions communicate with each other through infinite virtual registers. Thus it helps us understand the pure behavior of the programs and answers if the x86–64 ISA, specifically the register architecture, is the root cause of the heavy-tailed distributions.

We implement a network generator based on `lli`, an LLVM bitcode interpreter. We performed this analysis on four programs from CortexSuite and seven programs from SPEC CPU2006 (all the programs that ran correctly under the infrastructure), compiled with LLVM `-O0` to `-O2`. We only present the results of SPEC with `-O2` optimization; the results with different optimization levels are similar with each other, and the results of CortexSuite draw the same conclusion with SPEC.

In addition, another ISA we exercise is the Java VM architecture. The Java bytecode is a stack based instruction set where every communication between Java instructions are performed via the stack. This additional analysis is performed to see how the fundamental nature of stack machines which involves frequent pushing and popping affects the network distributions. We implement a Java instruction recorder to construct execution networks at the bytecode level. The analysis is performed on seven single-threaded DaCapo benchmarks (DaCapo–2006–10–MR2) [2].

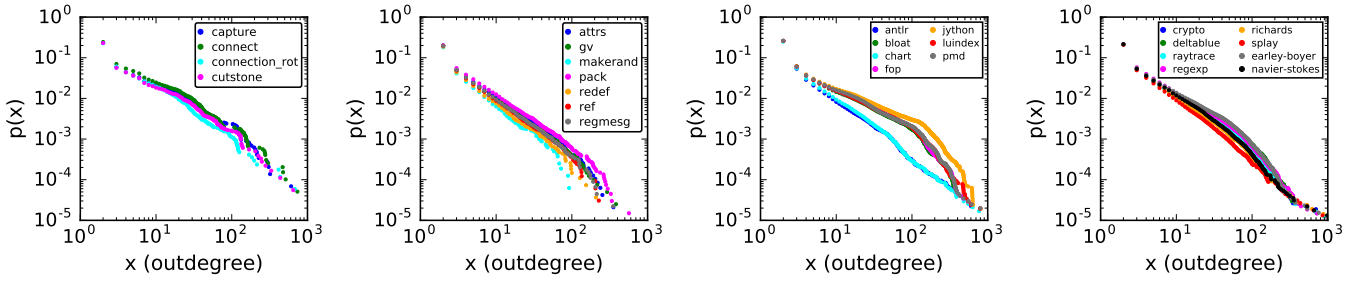
The results of LLVM IR and Java bytecode are shown in Figures 4(a) and 4(b), respectively. Again, we observe that all the benchmarks from both set of networks follow heavy-tailed distributions. This implies that the fact that heavy tails are found in various networks is a more fundamental property of program structures rather than an artifact of specific choices of the compiler, the compiler optimizations and/or the ISA.

VII. Are Heavy Tails Due to Inputs?

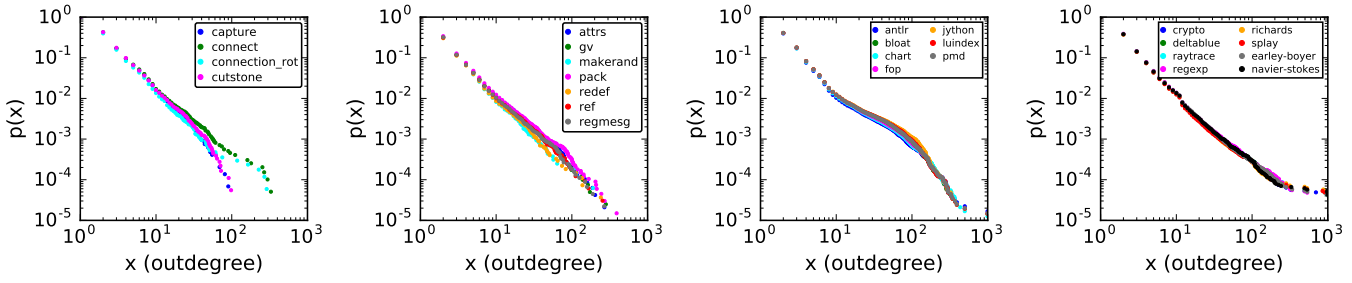
Another angle of program execution we explore is using different inputs for the same program. When considering program inputs there are two aspects: size and variation. Although these two are not completely independent, intuitively we expect program size to have less impact on the distribution of program networks. This is because the control path taken by programs tends to remain similar with different input sizes (e.g., imagine a loop count being varied). On the other hand, input variation has a better chance of affecting program networks because it is possible that completely different control paths can be exercised and result in different networks. This can happen in large programs (i.e., real-world programs opposed to computation kernels) which have complex control flows.

To study different input sizes, we analyze CortexSuite with three inputs: `small`, `medium` and `large`. The results of input size (CortexSuite) are not presented since the distributions were nearly identical with different inputs as we imagined. For input variation, we provide multiple `test` inputs to `gobmk` and `perlbench` (the core Perl interpreter) from SPEC CPU2006, analyze JVM 1.7.80 at the x86–64 ISA level (as opposed to the Java bytecode level presented in the previous section) with DaCapo programs as inputs, and also V8 with eight JavaScript programs.

Fig. 5 presents the results for the study of input variation. Overall, the output distributions are fairly consistent with different input variations. This can be seen that there is less variation compared to the results with previous CCDFs (Figures 1 to 4). Interestingly, the memory networks vary more than the register networks; the register networks are nearly identical with different inputs. For register networks, Fig. 5(e) shows that the results of `gobmk` is an exception, but actually there are two unique distributions where each of them contains two inputs. These two input pairs have mutual actions with each other; `connect` and `connection_rot` involve computing connection distances, and `capture` and `cutstone` involve reading the result and proposing edge moves. We believe that this is an example of input variations exercising different control paths.

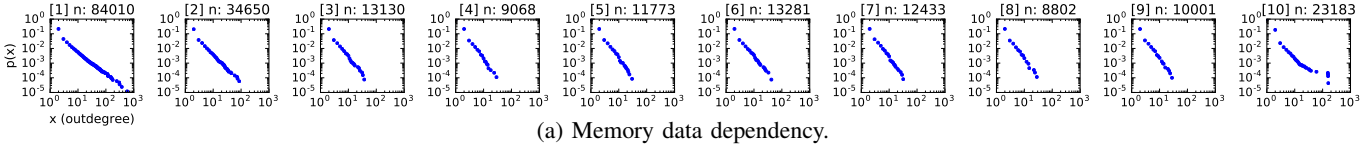


(a) `gobmk` (memory). (b) `perlbench` (memory). (c) DaCapo (memory). (d) V8 (memory).

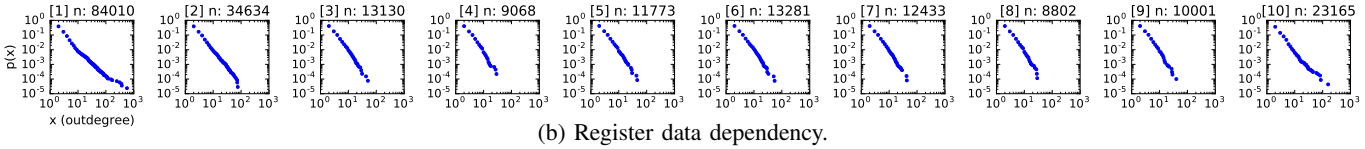


(e) `gobmk` (register). (f) `perlbench` (register). (g) DaCapo (register). (h) V8 (register).

Fig. 5: Outdegree-based log-log plot of the CCDF of the program networks for four benchmarks with different inputs. All the results follow heavy-tailed distributions. The distributions are mostly resilient with inputs.

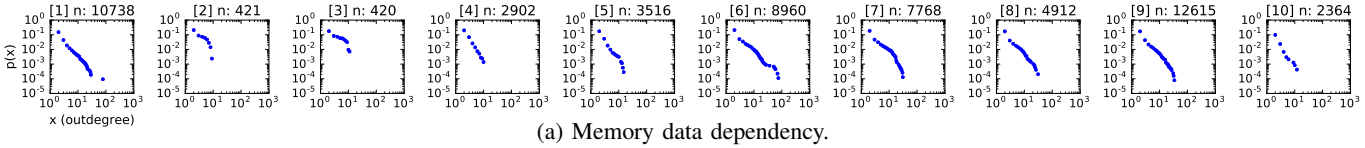


(a) Memory data dependency.

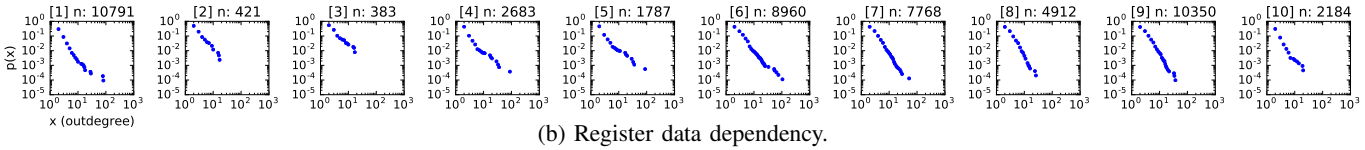


(b) Register data dependency.

Fig. 6: Outdegree-based log-log plot of the CCDF divided into 10 time series for `xalancbmk`'s communication networks.



(a) Memory data dependency.



(b) Register data dependency.

Fig. 7: Outdegree-based log-log plot of the CCDF divided into 10 time series for `calculix`'s communication networks.

From Sections III to VII we have examined several key factors that determine network characteristics to see if any of them are the root causes of heavy-tailed distributions. None of them appears to be the root cause. In the following two sections we further investigate into the details of program networks from two dimensions: time and space.

VIII. Are Heavy Tails Phase Dependent?

One possible hypothesis that explains heavy-tailed distributions when we focus on time is that few instructions executed in initial execution phases generate read-only and heavily used data that is consumed by a great number of following instructions. In order to investigate this hypothesis, we present

a time series of the outdegree distributions of `xalancbmk` and `calculix` from the SPEC CPU2006 benchmark suite.

We divide the execution of `xalancbmk` into 10 epochs (each epoch corresponds to about 33M dynamic instructions) and show each CCDF in Fig. 6. The subfigures represent consecutive execution epochs from left to right. On top of each figure, the number in the square bracket presents the *i*-th epoch along with *n*, the number of vertices (i.e., static instructions) in the network. The dependencies which cross the epoch boundary are removed from the network, which means that each subfigure contains only the producer-consumer relationships within the epoch.

We can see from Fig. 6(a) that the scale of the first epoch is different from the rest of the epochs. It has more number of static instructions, and the maximum outdegree is an order of magnitude higher. The distributions of the other epochs are fairly similar with each other. Similar trend holds for register data dependency as seen from Fig. 6(b). This supports our hypothesis that the high outdegree instructions are indeed executed in the first epoch, although what is more interesting is the fact that the other computation phases of the execution (i.e., epochs 2 to 10) still have heavy tails.

In Fig. 7 we show the results of `calculix` which have different characteristics from that of `xalancbmk`. The figures are similarly constructed where each epoch includes about 16M dynamic instructions. We can see that distributions have high variations throughout the execution, and seems that the initialization do not account for the highest outdegree instructions. Our hypothesis that the initialization is responsible for heavy tails is not fully correct for some applications. Nonetheless, the distributions of all the epochs have heavy tails. Interestingly, when we compare the distributions of memory and register networks within the same epoch, the distributions have high similarity. This implies that the communication volume of memory and register are correlated with each other.

IX. Are Heavy Tails Function Dependent?

Instead of slicing programs by time next we examine program networks at a code granularity. We explore whether small components out of which programs are constructed also follow heavy-tailed distributions, or instead those pieces follow other distributions (e.g., random distribution) but end up having heavy tails as a whole. This can be studied by decomposing the program networks into smaller units and analyze their outdegree distributions. In our case, we choose function as a unit of program's building blocks.

We extract functions which have more than 200 static instructions from SPEC CPU2006 (GCC with `-O2`) and analyze whether each of them follows a heavy-tailed distribution or not. We choose 200 as a threshold considering the statistical nature of the procedure. Choosing 200 allows us to have mostly consistent results: if the distribution follows power law, it is a better fit than exponential distribution and can be concluded as a heavy-tailed distribution. Similar with the experiment shown in the previous section, the dependencies which cross the function boundary are removed from the network.

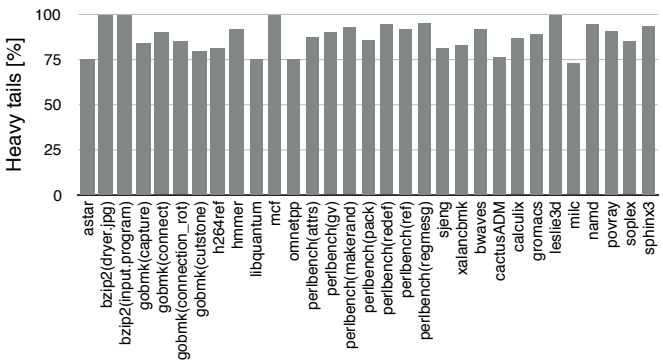


Fig. 8: Percentage of functions with heavy tails.

Fig. 8 presents the percentage of functions which follow heavy-tailed distributions. Interestingly, the majority (72.2% to 100%) of functions' networks are heavy tails. Heavy-tailed distributions not only exist at the program level but also at the function level. Thus heavy tails appear to be fractal in both time (execution phases) and space (functions).

X. Why Do Heavy Tails Emerge?

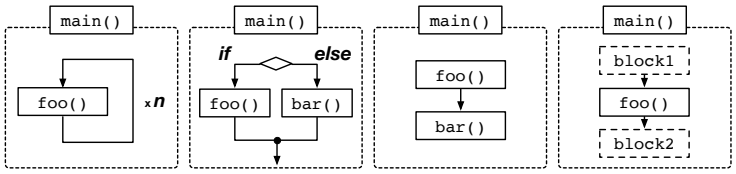
Following the results of the previous section, we develop a mechanistic generative model that explains why programs have heavy tails. The question we try to answer is the following: if functions within a program have heavy tails, can we demonstrate that typical composition of these functions result in a heavy-tailed distribution? Developing a model that tackles this question helps us understand the fractal nature of heavy tails that exists at code granularity.

To appreciate the complexity of this question consider an analogy of combining networks that are heavy tailed, let us say that all these networks have one high outdegree node that has the same cardinality. In this case, when the networks are mixed, there are several nodes with the same outdegree; in the extreme it is possible that the heavy tail can vanish even when the combining networks have heavy tails. However, if the combining networks have heavy tails of different outdegrees it is possible that the mixture still has a heavy tail.

In order to understand what happens to programs in which functions have heavy tails, our strategy is to divide-and-conquer: we split programs into common basic structures including loops, branches and sequences of function calls, and combinations of these, to discuss how heavy tails can emerge when each of these structures have a heavy-tailed distribution. Instead of offering mathematical proofs (which are untenable for complex compositions), we conduct Monte Carlo simulations to computationally evaluate if most programs follow heavy tails and/or power laws.

A. Generative Model Development

To simplify our model, we assume that there is no data dependency between functions, and all functions follow power



(a) Loop. (b) Branch. (c) Seq. calls. (d) Code and call.

Chain length	n	Avg. p-value	Power laws [%]	Heavy tails [%]
2	603	0.48	81	98
5	1357	0.44	79	99
10	2551	0.46	79	99
50	12948	0.34	62	100

(e) Power law and heavy tail analysis of synthesized functions.

Fig. 9: The four basic structures for the generative model and the analysis result of the synthesized functions.

laws. We first profile all *real* functions in SPEC CPU2006 compiled with GCC `-O3` by collecting their instruction counts and outdegrees distributions. Among over 11,000 functions in SPEC CPU2006, we extract functions having more than 200 instructions *and* following power laws. There are more than 2,700 functions left after the extraction. We use the distributions of these functions to compose more complex program structures with four general cases and test their distributions.

B. Basic Structures

We discuss four basic program structures using functions as building blocks, namely: loop, branch, sequential calls, and the combination of code and function calls.

- **Loop:** function `main` iteratively invokes function `foo` for n times (Fig. 9(a)). Invoking the same function for multiple times has no impact on the outdegrees of the network. Thus, if function `foo` aligns with a power law, its caller function `main` also exhibits a power law.
- **Branch:** Fig. 9(b) gives an example of the branch structure, an if-else statement, where one path executes function `foo` and the other executes `bar`. If both functions follow power laws, their caller `main` also follows a power law because only one function is executed.

  What if function `main` has a loop outside the if-else statement? If the branch is 100% taken or not-taken during the entire execution, `main` follows a power law as discussed in the loop case. If both branches are taken during the execution, we need to combine the outdegree distributions of `foo` and `bar`, which will be discussed in the next case. The probability of branch being taken does not change the outdegree distribution of the function.
- **Sequential Calls:** a caller function delegates its computations into multiple callee functions in sequence. We assume that functions `foo` and `bar` in Fig. 9(c) are two independent power law distributions.

  Since the distribution of the sum of two power law distributions is theoretically hard to conclude [15], we instead *synthesize* functions and then analyze their outdegree distributions. We synthesize a function by randomly selecting multiple functions from the function set of SPEC CPU2006 and integrate their outdegree distributions together.
- **Combination of Code and Function Calls:** the fourth structure is a generalized case of the sequential call structure. Instead of delegating all computations to its callee function(s), the caller function itself and its callees both perform computations as Fig. 9(d) depicts. While function `main` performs some computations in `block1` and `block2`, it calls another function `foo` in between the two blocks.

  It is difficult to summarize the probability distribution of `main`, because we lack a mathematical way for formalizing the relationship between `block1`, `foo` and `block2`. However, if we disregard these relationships, i.e., assume that `block1`, `foo` and `block2` are independent, we can sum up the outdegree distributions of the caller function and its callee, which becomes the same case as sequential calls.

C. Analysis on the Synthesized Functions

To evaluate the impact of the length of sequential function chains, we test four lengths: 2, 5, 10 and 50. For each length, we synthesize 100 test cases. To identify if the resulting function chain follows a power law, we conduct the same 5,000 Monte Carlo simulations described in Section II. Resulting functions are *not* heavy-tailed if the best fit distribution(s) with the likelihood test contains exponential distribution.

The result is shown in Fig. 9(e). The table shows the average number of nodes of the resulting networks n, the average p-value, the ratio of resulting functions having power laws and heavy tails. Over 99% of the synthesized functions with different number of instructions have heavy tails and about 76% of them follow power laws. This result aligns with the observations we have seen so far that most program structures are indeed heavy tails, and our generative model convinces us that the fractal nature of heavy tails is a fundamental property of program execution.

XI. Heavy Tails in Source Code

A natural question that arises at this point is: why do functions follow power laws or have heavy tails? In this section we perform a semantic analysis of real code to reveal what program structures result in heavy-tailed distributions. For this purpose, we discuss a simple matrix multiplication function first and then two real functions from SPEC CPU2006.

A. Code Patterns for Heavy Tails

We first provide a simple but demonstrative example. We implement a matrix multiplication in two programming languages, C and Java, and study the x86–64 ISA and Java bytecode networks. The memory- and register-based networks when compiled with GCC `-O3` optimization, and also the Java bytecode network all follow heavy-tailed distributions. For

```
1 long read_min(network_t *net) {
2   // net is a central data structure
3   ...
4   net->n_trips = t;
5   net->m_org = h;
6   net->n = (t+t+1);
7   net->m = (t+t+t+h);
8   ...
9   net->nodes = (node_t *)calloc(net->n + 1, sizeof(node_t));
10  net->dummy_arcs = (arc_t *)calloc(net->n, sizeof(arc_t));
11  net->arcs  = (arc_t *)calloc(net->max_m, sizeof(arc_t));
12  ...
13  net->stop_nodes = net->nodes + net->n + 1;
14  net->stop_arcs  = net->arcs + net->m;
15  net->stop_dummy = net->dummy_arcs + net->n;
16  ...
17 }
```

(a) The `read_min` function in `mcf`.

```
1 static void sendMTFValues(EState *s) {
2   ...
3   while (True) {
4     if (nGroups == 6 && 50 == ge-gs+1) {
5       define BZ_ITUR(nn) s->rfreq[bt][mtfv[gs+(nn)]]++
6
7       // The following code block unrolls a loop
8       // to update mtfv
9       BZ_ITUR(0);  BZ_ITUR(1);  ... BZ_ITUR(4);
10      BZ_ITUR(5);  BZ_ITUR(6);  ... BZ_ITUR(9);
11      ...
12      BZ_ITUR(45); BZ_ITUR(46); ... BZ_ITUR(49);
13      ...
14    }
15    ...
16 }
```

(b) The `sendMTFValues` function in `bzip2`.

Fig. 10: Relevant code from `mcf` and `bzip2`.

both programming languages, the source code that corresponds to the highest outdegrees initializes the data arrays (i.e., the instruction that stores the address of the arrays).

We next conduct a semantic analysis for two benchmarks, namely `mcf` and `bzip2` from SPEC CPU2006. We select these programs considering their program sizes (in lines of code) to understand and perform further analysis in a reasonable amount of time.

- **Mcf:** `Mcf` has a central data structure `net` which records the program state and is used by multiple steps across the whole benchmark. Initializing the program state by manipulating `net` between line 4 to line 15 in function `read_min` account for the highest outdegree instructions in the program.
- **Bzip2:** `Bzip2` exhibits heavy tails where it has a unique shape with a huge cliff (Fig. 1(a)). One potential reason for this shape is explained with an example we show in Fig. 10(b). The code is heavily unrolled using a macro `BZ_ITUR`. It updates the `mtfv` array, which is a member of the central data structure `EState`. This unrolling causes multiple instructions having high outdegrees with the exact same value, which breaks the regularity in its shape.

The three examples shown in this section report the existence of a central data structure of a function (matrix multiplication) and programs (`mcf` and `bzip2`). These structures are accessed throughout the execution of the program, and instructions that generate the base address and read-only values in these structures have a great number of consumers. Although this finding is not enough to explain the causes of heavy tails, it reveals why high outdegree instructions exist to some extent. Further investigation is left for future work.

TABLE I: Clustering results based on the distribution similarity of the memory networks of SPEC CPU2006 benchmarks compiled with GCC `-O0` and `-O3`.

Cluster ID	GCC `-O0`	GCC `-O3`
1	bzip2(dryer.jpg) bzip2(input.program) perlbench(makerand) leslie3d, bwaves hmmer, sjeng, milc gobmk perlbench(attrs, gv, pack) perlbench(redef, ref, regmesg) omnetpp, xalancbmk h264ref, cactusADM calculix, gromacs, namd povray, soplex, sphinx3	gobmk perlbench(attrs, gv, pack) perlbench(redef, ref, regmesg) omnetpp, xalancbmk h264ref, cactusADM calculix, gromacs, namd povray, soplex, sphinx3
2	mcf	mcf
3	libquantum astar	astar
4	N/A	libquantum, leslie3d bzip2(input.program)
5	N/A	bwaves
6	N/A	bzip2(dryer.jpg) perlbench(makerand) hmmer, sjeng, milc

XII. Are Heavy Tails Unique Phenomenon?

We have verified throughout the paper that program social networks follow heavy-tailed distributions. Will this observation lead to new advances? This is a difficult question to answer where we provide indirect evidence in the affirmative. One approach to expand on new observations is to perform workload characterization for better understanding of the system behavior. Therefore if the characterization results are different from prior studies, then there is a high chance of building new optimizations based on the dependence skew concept. To evaluate whether the observation is already subsumed by prior work, we perform cluster analysis based on dependence skew property. If the obtained clusters are different then it is likely that program social networks and dependence skew offer a new lens to view program execution that can lead to new advances.

A. Clustering on Distribution Distance

In this paper we use the outdegree distributions for clustering. We cluster the programs such that the more similar the *shape* of programs' outdegree distributions (i.e., CCDF) are, the higher opportunity they are in the same cluster. Because each benchmark has different ranges of outdegrees and nodes (i.e., static instruction counts), we need to normalize them to represent the shapes with relative values. First, we normalize the outdegrees of each benchmark with range $[0, 1]$ and separate this range such as when generating a histogram into n bins, where we use 10 bins in this paper. Then for each bin, instead of using the absolute value of nodes, we use how many percentage of nodes each bin contains, whose value is also between $[0, 1]$. We finally compute the Euclidean distance between programs via their normalized outdegree distributions and discover program clusters. We use the hierarchical clustering analysis to group programs in an agglomerative way [10].

B. Program Cluster Analysis

We cluster the memory-based program social networks of SPEC CPU2006 benchmarks compiled using GCC with `-O0`

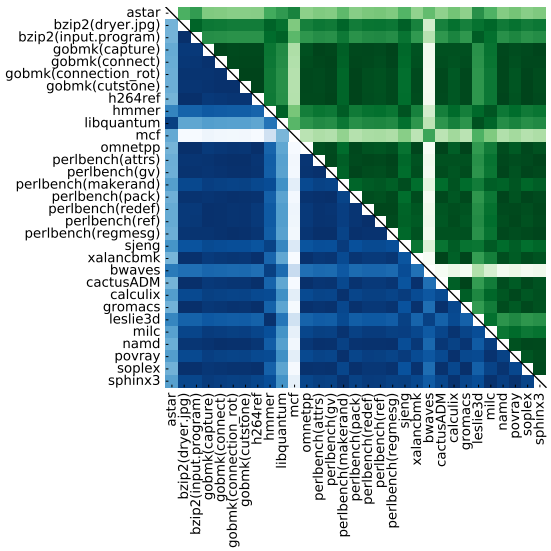


Fig. 11: Heatmaps based on the distribution similarity between the memory network of SPEC CPU2006. Blue (left) and green (right) represent the results of `-O0` and `-O3`, respectively.

and `-O3` optimizations. The clustering result is shown in TABLE I and the similarities are shown as heatmaps in Fig. 11. In the figure, the blue (left half) and the green (right half) areas represent the `-O0` and `-O3` optimizations, respectively. Within each area the color density represents the similarity between a pair of benchmarks: the darker the higher the similarity.

While there is no significant difference in program clusters among the two optimization levels, e.g., the majority of the benchmarks in cluster ID 1 are clustered in the same group for both levels, there are few differences that can be observed. For instance we have seen in Section V that aggressive compiler optimization generates more variations in the outdegree distributions, which we believe is the cause of different cluster counts (three vs. six). Among all the benchmarks, `mcf` and `astar` show relatively higher inconsistency with other benchmarks for both optimization levels which can be seen from the light color in the figure. Additionally, `libquantum` for `-O0` and `bwaves` for `-O3` present dissimilarities.

C. Discussion of Program Clustering

Prior studies use microarchitecture dependent and independent features to cluster programs to explore which programs are similar with each other [8, 12]. In this paper, we have clustered the programs based on a new property of program structures, the dependence skew. Here we compare our results with that of Phansalkar et al. [12]. They also cluster SPEC CPU2006 benchmarks using both microarchitecture dependent features (e.g., L2 cache misses per instruction) and independent features (e.g., number of branches per instruction) to characterize programs. Our clustering analysis shares both similar and dissimilar results with them, which are briefly summarized as follows.

- **Similarities:** under both clustering techniques, `mcf` exhibits distinct characteristics from other benchmarks. Additionally, both clustering techniques show that the characteristics of most benchmarks do not vary significantly given different input sets (also observed in Section VII).
- **Differences:** while `astar` seems to share common features with other benchmarks according to Phansalkar et al.'s study, this benchmark is one of the two that have different characteristics with others under the outdegree distribution based clustering. On the contrary, `xalancbmk` has high similarity with other benchmarks according to the outdegree distribution based clustering, while it is reported to be distant from other benchmarks with theirs.

Analyzing the reasons behind the scene given these two types of clustering can be a challenging task. However, it seems that the proposed outdegree distribution based technique can provide researchers with a new and different view of programs from the existing feature-based clustering. This may open up new optimization opportunities in various aspects including performance, reliability and security.

XIII. Related Work

There exist few studies which explored the communications between instructions by focusing on different metrics from our work. Franklin et al. found that the degrees of use of register instances (or the number of dynamic consumers as opposed to static consumers in our work) are mostly bounded to zero, one or two, which helped them design the distributed register file for multiscalar [5]. Eeckhout et al. statistically studied this characteristic and showed that the distributions of the degrees of use of register instances follow power laws [4]. They modeled the register traffic of programs with the power law function and used it to perform architecture design space exploration.

The degree of use of register instances is a metric which can be extracted from our network view and is indeed of interest. Instead of studying the metric, however, we focused on outdegrees, where the characterization performed in our paper is much broader. In addition to various controlled experiments we also study memory traffic and observe the presence of heavy tails. We note that studying the memory traffic is not a straightforward extension. The use of dynamic traces to study the distribution (as performed in prior work) can be limiting because of the size of the traces, where the novel network representation allows us to overcome this problem.

XIV. Conclusions

A potent program characterization technique can help improve the efficiency of computer systems. In this paper, we propose a new way to characterize programs by viewing them as social networks. By modeling static instructions as vertices and communications between them as edges we analyzed the outdegree distributions in programs. Based on empirical analysis with multiple benchmarks, compilers and their optimizations, languages, and inputs, we observe that

most programs' outdegree distributions have heavy tails and are highly skewed. We also developed a generative model to explain why programs exhibit heavy tails.

Ultimately the usefulness of this observation depends on how it is utilized by designers of computer systems. There is some evidence that the heavy tails property is unique because the clusters it forms are distinct from prior studies and thus it might lead to new architectural techniques. The broader idea of network analysis on program structures may lead to even more exciting opportunities.

ACKNOWLEDGEMENTS

The work was partially supported by grant N00014-15-1-2173, and gifts from Bloomberg and an Alfred P. Sloan Research Fellowship.

REFERENCES

[1] J. Alstott, E. Bullmore, and D. Plenz, "Powerlaw: a Python package for analysis of heavy-tailed distributions," *PloS one*, vol. 9, no. 1, Jan. 2014.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, 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. Stefanovi c, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06*, Oct. 2006, pp. 169–190.

[3] A. Clauset, C. R. Shalizi, and M. Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, pp. 661–703, Nov. 2009.

[4] L. Eeckhout and K. De Bosschere, "Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces," in *PACT '01*, Sep. 2001, pp. 25–34.

[5] M. Franklin and G. S. Sohi, "Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors," in *MICRO-25*, Dec. 1992, pp. 236–245.

[6] J. Gray, "Google Chrome: the making of a cross-platform browser," *Linux Journal*, vol. 2009, no. 185, Sep. 2009.

[7] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM CAN*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[8] K. Hoste and L. Eeckhout, "Comparing benchmarks using key microarchitecture-independent characteristics," in *IISWC '06*, Oct. 2006, pp. 83–92.

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

[10] O. Maimon and L. Rokach, Eds., *The data mining and knowledge discovery handbook*, 2005.

[11] M. Owens, "Embedding an SQL database with SQLite," *Linux Journal*, vol. 2003, no. 110, pp. 2–2, Jun. 2003.

[12] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *ISCA '07*, Jun. 2007, pp. 412–421.

[13] H. Sasaki, F.-H. Su, T. Tanimoto, and S. Sethumadhavan, "Heavy tails in program structure," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 34–37, May 2016.

[14] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "CortexSuite: a synthetic brain benchmark suite," in *IISWC '14*, Oct. 2014, pp. 76–79.

[15] C. Wilke, S. Altmeyer, and T. Martinetz, "Large-scale evolution and extinction in a hierarchically structured environment," in *Artificial Life VI*, 1998, pp. 266–272.