Determinism, Complexity, and Predictability in Computer Performance

January 3, 2014

Abstract

Computers are deterministic dynamical systems [**?**]. Among other things, that implies that one should be able to use deterministic forecast rules to predict aspects of their behavior. That statement is sometimes—but not always—true. The memory and processor loads of some simple programs are easy to predict, for example, but those of more-complex programs like `gcc` are not. The goal of this paper is to determine why that is the case. We conjecture that, in practice, complexity can effectively overwhelm the predictive power of deterministic forecast models. To explore that, we build models of a number of performance traces from different programs running on different Intel-based computers. We then calculate the *permutation entropy*—a temporal entropy metric that uses ordinal analysis—of those traces and correlate those values against the prediction success.

1 Introduction

Computers are among the most complex engineered artifacts in current use. Modern microprocessor chips contain multiple processing units and multi-layer memories, for instance, and they use complicated hardware/software strategies to move data and threads of computation across those resources. These features—along with all the others that go into the design of these chips—make the patterns of their processor loads and memory accesses highly complex and hard to predict. Accurate forecasts of these quantities, if one could construct them, could be used to improve computer design. If one could predict that a particular computational thread would be bogged down for the next 0.6 seconds waiting for data from main memory, for instance, one could save power by putting that thread on hold for that time period (e.g., by migrating it to a processing unit whose clock speed is scaled back). Computer performance traces are, however, very complex. Even a simple "microkernel," like a three-line loop that repeatedly initializes a matrix in column-major order, can produce *chaotic* performance traces [**?**], as shown in Figure 1, and chaos places fundamental limits on predictability.

The computer systems community has applied a variety of prediction strategies to traces like this, most of which employ regression. An appealing alternative builds on the recently established fact that computers can be effectively modeled as deterministic nonlinear dynamical systems [**?**]. This result implies the existence of a deterministic forecast rule for those dynamics. In particular, one can use *delay-coordinate embedding* to reconstruct the underlying dynamics of computer performance, then use the resulting model to forecast the future values of computer performance metrics such as memory or processor loads [**?**]. In the case of simple microkernels like the one that produced

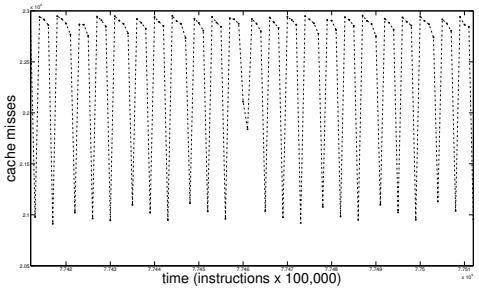

Figure 1: A small snippet of the L2 cache miss rate of `col_major`, a three-line C program that repeatedly initializes a matrix in column-major order, running on an Intel Core Duo®-based machine. Even this simple program exhibits chaotic performance dynamics.

the trace in Figure 1, this deterministic modeling and forecast strategy works very well. In more-complicated programs, however, such as speech recognition software or compilers, this forecast strategy—as well as the traditional methods—break down quickly.

This paper is a first step in understanding when, why, and how deterministic forecast strategies fail when they are applied to deterministic systems. We focus here on the specific example of computer performance. We conjecture that the complexity of traces from these systems—which results from the inherent dimension, nonlinearity, and nonstationarity of the dynamics, as well as from measurement issues like noise, aggregation, and finite data length—can make those deterministic signals *effectively* unpredictable. We argue that *permutation entropy* [**?**], a method for measuring the entropy of a real-valued-finite-length time series through ordinal analysis, is an effective way to explore that conjecture. We study four examples—two simple microkernels and two complex programs from the SPEC benchmark suite—running on different Intel-based machines. For each program, we calculate the permutation entropy of the processor load (instructions per cycle) and memory-use efficiency (cache-miss rates), then compare that to the prediction accuracy attainable for that trace using a simple deterministic model.

It is worth taking a moment to consider the theoretical possibility of this task. We are not attempting to predict the state of the CPU at an arbitrary point in the future — this, at least with perfect accuracy, would be tantamount to solving the halting problem. What we are attempting is to predict aspects or functions of the running of the CPU: instructions executed per second, cache misses per 100,000 instructions, and similar statistics. Prediction of these quantities at some finite time in the future, even with perfect accuracy, does not violate the Rice-Shapiro theorem.

2 Modeling Computer Performance

Delay-coordinate embedding allows one to reconstruct a system's full state-space dynamics from a *single* scalar time-series measurement—provided that some conditions hold regarding that data. Specifically, if the underlying dynamics and the measurement function—the mapping from the unknown state vector $\vec{X}$ to the scalar value x that one is measuring—are both smooth and generic, Takens [**?**] formally proves that the delay-coordinate map

$$F(\tau, m)(x) = ([x(t)\ x(t+\tau)\ \ldots\ x(t+m\tau)])$$

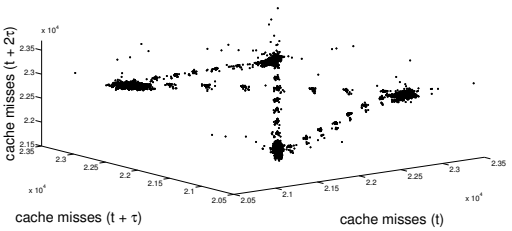

Figure 2: A 3D projection of a delay-coordinate embedding of the trace from Figure 1 with a delay (τ) of 100,000 instructions.

from a d-dimensional smooth compact manifold M to Re^{2d+1}, where t is time, is a diffeomorphism on M—in other words, that the reconstructed dynamics and the true (hidden) dynamics have the same topology.

This is an extremely powerful result: among other things, it means that one can build a formal model of the full system dynamics without measuring (or even knowing) every one of its state variables. This is the foundation of the modeling approach that is used in this paper. The first step in the process is to estimate values for the two free parameters in the delay-coordinate map: the delay τ and the dimension m. We follow standard procedures for this, choosing the first minimum in the average mutual information as an estimate of τ [**?**] and using the false-near(est) neighbor method of [**?**], with a threshold of 10%, to estimate m. A plot of the data from Figure 1, embedded following this procedure, is shown in Figure 2. The coordinates of each point on this plot are differently delayed elements of the `col_major` L2 cache miss rate time series $y(t)$: that is, $y(t)$ on the first axis, $y(t + \tau)$ on the second, $y(t + 2\tau)$ on the third, and so on. Structure in these kinds of plots—clearly visible in Figure 2—is an indication of determinism[1]. That structure can also be used to build a forecast model.

Given a nonlinear model of a deterministic dynamical system in the form of a delay-coordinate embedding like Figure 2, one can build deterministic forecast algorithms by capturing and exploiting the geometry of the embedding. Many techniques have been developed by the dynamical systems community for this purpose (e.g., [**?**, **?**]). Perhaps the most straightforward is the "Lorenz method of analogues" (LMA), which is essentially nearest-neighbor prediction in the embedded state space [**?**]. Even this simple algorithm—which builds predictions by finding the nearest neighbor in the embedded space of the given point, then taking that neighbor's path as the forecast—works quite well on the trace in Figure 1, as shown in Figure 3. On the other hand, if we use the same approach to forecast the processor load[2] of the `482.sphinx3` program from the SPEC cpu2006 benchmark suite, running on an Intel i7®-based machine, the prediction is far less accurate; see Figure 4.

Table 1 presents detailed results about the prediction accuracy of this algorithm on four different examples: the `col_major` and `482.sphinx3` programs in Figures 3 and 4, as well as another simple microkernel that initializes the same matrix as `col_major`, but in row-major order, and another complex program (`403.gcc`) from the SPEC cpu2006 benchmark suite. Both microkernels were run on the Intel Core Duo® machine; both SPEC benchmarks were run on the Intel i7® machine.

[1] A deeper analysis of Figure 2—as alluded to on the previous page—supports that diagnosis, confirming the presence of a chaotic attractor in these cache-miss dynamics, with largest Lyapunov exponent $\lambda_1 = 8000 \pm 200$ instructions, embedded in a 12-dimensional reconstruction space [**?**].

[2] Instructions per cycle, or IPC

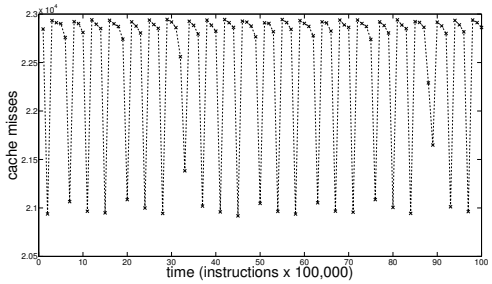

Figure 3: A forecast of the last 4,000 points of the signal in Figure 1 using an LMA-based strategy on the embedding in Figure 2. Red circles and blue ×s are the true and predicted values, respectively; vertical bars show where these values differ.

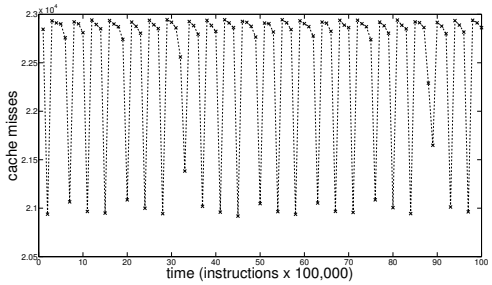

Figure 4: An LMA-based forecast of the last 4,000 points of a processor-load performance trace from the `482.sphinx3` benchmark. Red circles and blue ×s are the true and predicted values, respectively; vertical bars show where these values differ.

We calculated a figure of merit for each prediction as follows. We held back the last k elements[3] of the N points in each measured time series, built the forecast model by embedding the first $N-k$ points, used that embedding and the LMA method to predict the next k points, then computed the Root Mean Squared Error (RMSE) between the true and predicted signals:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{k}(c_i - \hat{p}_i)^2}{k}}$$

To compare the success of predictions across signals with different units, we normalized RMSE as follows:

$$nRMSE = \frac{RMSE}{X_{max,obs} - X_{min,obs}}$$

Table 1: Normalized root mean squared error (nRMSE) of 4000-point predictions of memory & processor performance from different programs.

	cache miss rate	instrs per cycle
`row_major`	0.0324	0.0778
`col_major`	0.0080	0.0161
`403.gcc`	0.1416	0.2033
`482.sphinx3`	0.2032	0.3670

The results in Table 1 show a clear distinction between the two microkernels, whose future behavior can be predicted effectively using this simple deterministic modeling strategy, and the more-complex SPEC benchmarks, for which this prediction strategy does not work nearly as well. This begs the question: If these traces all come from deterministic systems—computers—then why are they not equally predictable? Our conjecture is that the sheer complexity of the dynamics of the SPEC benchmarks running on the Intel i7® machine make them effectively impossible to predict.

3 Measuring Complexity

For the purposes of this paper, one can view entropy as a measure of complexity and predictability in a time series. A high-entropy time series is almost completely unpredictable—and conversely. This can be made more rigorous: Pesin's relation [**?**] states that in chaotic dynamical systems, the Shannon entropy rate is equal to the sum of the positive Lyapunov exponents, λ_i. The Lyapunov exponents directly quantify the rate at which nearby states of the system will diverge with time: $|\Delta x(t)| \approx e^{\lambda t} |\Delta x(0)|$. The faster the divergence, the more difficult prediction becomes.

Utilizing entropy as a measure of temporal complexity is by no means a new idea [**?**, **?**]. Its effective usage requires categorical data: $x_t \in \mathcal{S}$ for some finite or countably infinite *alphabet* $\mathcal{S}$, and data taken from real-world systems is effectively real-valued. To get around this, one must discretize the data—typically by binning. Unfortunately, this is rarely a good solution to the problem, as the binning of the values introduces an additional dynamic on top of the intrinsic dynamics whose

[3]Several different prediction horizons were analyzed in our experiment; the results reported in this paper are for k=4000

entropy is desired. The field of symbolic dynamics studies how to discretize a time series in such a way that the intrinsic behavior is not perverted, but these methods are fragile in the face of noise and require further understanding of the underlying system, which defeats the purpose of measuring the entropy in the first place.

Bandt and Pompe introduced *permutation entropy* (PE) as a "natural complexity measure for time series" [**?**]. Permutation entropy employs a method of discretizing real-valued time series that follows the intrinsic behavior of the system under examination. Rather than looking at the statistics of sequences of values, as is done when computing the Shannon entropy, permutation entropy looks at the statistics of the *orderings* of sequences of values using ordinal analysis. Ordinal analysis of a time series is the process of mapping successive time-ordered elements of a time series to their value-ordered permutation of the same size. By way of example, if $(x_1, x_2, x_3) = (9, 1, 7)$ then its *ordinal pattern*, $\phi(x_1, x_2, x_3)$, is 231 since $x_2 \leq x_3 \leq x_1$. This method has many features; among other things, it is robust to noise and requires no knowledge of the underlying mechanisms.

Definition (Permutation Entropy)**.** *Given a time series* $\{x_t\}_{t=1,\ldots,T}$*. Define* $\mathcal{S}_n$ *as all* $n!$ *permutations* π *of order* n*. For each* $\pi \in \mathcal{S}_n$ *we determine the relative frequency of that permutation occurring in* $\{x_t\}_{t=1,\ldots,T}$*:*

$$p(\pi) = \frac{|\{t|t \leq T - n, \phi(x_{t+1}, \ldots, x_{t+n}) = \pi\}|}{T - n + 1}$$

Where $|\cdot|$ *is set cardinality. The* permutation entropy *of order* $n \geq 2$ *is defined as*

$$H(n) = -\sum_{\pi \in \mathcal{S}_n} p(\pi) \log_2 p(\pi)$$

Notice that $0 \leq H(n) \leq \log_2(n!)$ [**?**]. With this in mind, it is common in the literature to normalize permutation entropy as follows: $\frac{H(n)}{\log_2(n!)}$. With this convention, "low" entropy is close to 0 and "high" entropy is close to 1. Finally, it should be noted that the permutation entropy has been shown to be identical to the Shannon entropy for many large classes of systems [**?**].

In practice, calculating permutation entropy involves choosing a good value for the wordlength n. The key consideration here is that the value be large enough that forbidden ordinals are discovered, yet small enough that reasonable statistics over the ordinals are gathered: e.g., $n = \underset{\ell}{\text{argmax}}\{T > 100\ell!\}$, assuming an average of 100 counts per ordinal. In the literature, $3 \leq n \leq 6$ is a standard choice—generally without any formal justification. In theory, the permutation entropy should reach an asymptote with increasing n, but that requires an arbitrarily long time series. In practice, what one should do is calculate the *persistent* permutation entropy by increasing n until the result converges, but data length issues can intrude before that convergence is reached.

Table 2 shows the permutation entropy results for the examples considered in this paper, with the nRMSPE prediction accuracies from the previous section included alongside for easy comparison. The relationship between prediction accuracy and the permutation entropy (PE) is as we conjectured: performance traces with high PE—those whose temporal complexity is high, in the sense that little information is being propagated forward in time—are indeed harder to predict using the simple deterministic forecast model described in the previous section. The effects of changing n are also interesting: using a longer wordlength generally lowers the PE—a natural consequence of finite-length data—but the falloff is less rapid in some traces than in others, suggesting that those values are closer to the theoretical asymptote that exists for perfect data. The persistent PE

Table 2: Prediction error (in nRMSPE) and permutation entropy (for different wordlengths n

cache misses	error	$n = 4$	$n = 5$	$n = 6$
`row_major`	0.0324	0.6751	0.5458	0.4491
`col_major`	0.0080	0.5029	0.4515	0.3955
`403.gcc`	0.1416	0.9916	0.9880	0.9835
`482.sphinx3`	0.2032	0.9913	0.9866	0.9802
insts per cyc	error	$n = 4$	$n = 5$	$n = 6$
`row_major`	0.0778	0.9723	0.9354	0.8876
`col_major`	0.0161	0.8356	0.7601	0.6880
`403.gcc`	0.2033	0.9862	0.9814	0.9764
`482.sphinx3`	0.3670	0.9951	0.9914	0.9849

values of 0.5–0.6 for the `row_major` and `col_major` cache-miss traces are consistent with dynamical chaos, further corroborating the results of [**?**]. (PE values above 0.97 are consistent with white noise.) Interestingly, the processor-load traces for these two microkernels exhibit more temporal complexity than the cache-miss traces. This may be a consequence of the lower baseline value of this time series.

4 Conclusions & Future Work

The results presented here suggest that permutation entropy—a ordinal calculation of forward information transfer in a time series—is an effective metric for predictability of computer performance traces. Experimentally, traces with a persistent PE $\gtrapprox 0.97$ have a natural level of complexity that may overshadow the inherent determinism in the system dynamics, whereas traces with PE $\lessapprox 0.7$ seem to be highly predictable (viz., at least an order of magnitude improvement in nRMSPE).

If information is the limit, then gathering and using more information is an obvious next step. There is an equally obvious tension here between data length and prediction speed: a forecast that requires half a second to compute is not useful for the purposes of real-time control of a computer system with a MHz clock rate. Another alternative is to sample several system variables simultaneously and build multivariate delay-coordinate embeddings. Existing approaches to that are computationally prohibitive [**?**]. We are working on alternative methods that sidestep that complexity.

5 New Figures and Tables

Acknowledgment

This work was partially supported by NSF grant #CMMI-1245947 and ARO grant #W911NF-12-1-0288.

Table 3: Embedding Parameters for reference if needed.

	τ	m
gcc	10	13
col_major	2	12
SVD_Full	10	12
SVD_IPC_Regime1	5	14
SVD_IPC_Regime2	10	12
SVD_IPC_Regime3	2	9
SVD_IPC_Regime4	3	11
SVD_IPC_Regime5	23	10
SVD_IPC_Regime6	30	12

Table 4: Average nRMSE over 15 runs for each signal and average (?) pe [[Ryan:is this average over all 15 and is this weighted]] at word length 5 and 6 for each signal.

	Horizon	nRMSE LMA	nRMSE mean	$l = 5$	$l = 6$
gcc	4542	0.1407 ± 0.0063	0.1487 ± 0.0066	-	-
col_major	14,709	0.0252 ± 0.0061	0.1975 ± 0.0436	-	-
SVD_Full	22,072	0.0323 ± 0.0019	0.1784 ± 0.0040	-	-
SVD_IPC_Regime1	2,158	0.1680 ± 0.0317	0.3517 ± 0.5223	0.9786	0.9675
SVD_IPC_Regime2	6,902	0.1716 ± 0.0043	0.1762 ± 0.0012	0.8800	0.8493
SVD_IPC_Regime3	947	0.0507 ± 0.0011	0.5413 ± 0.0005	0.7646	0.7029
SVD_IPC_Regime4	2,929	0.1288 ± 0.0471	0.2308 ± 0.0867	0.8735	0.7770
SVD_IPC_Regime5	3,255	0.0235 ± 0.0022	0.1306 ± 0.0003	0.7281	0.6724
SVD_IPC_Regime6	5,804	0.0196 ± 0.0022	0.0508 ± 0.0003	0.8316	0.7615

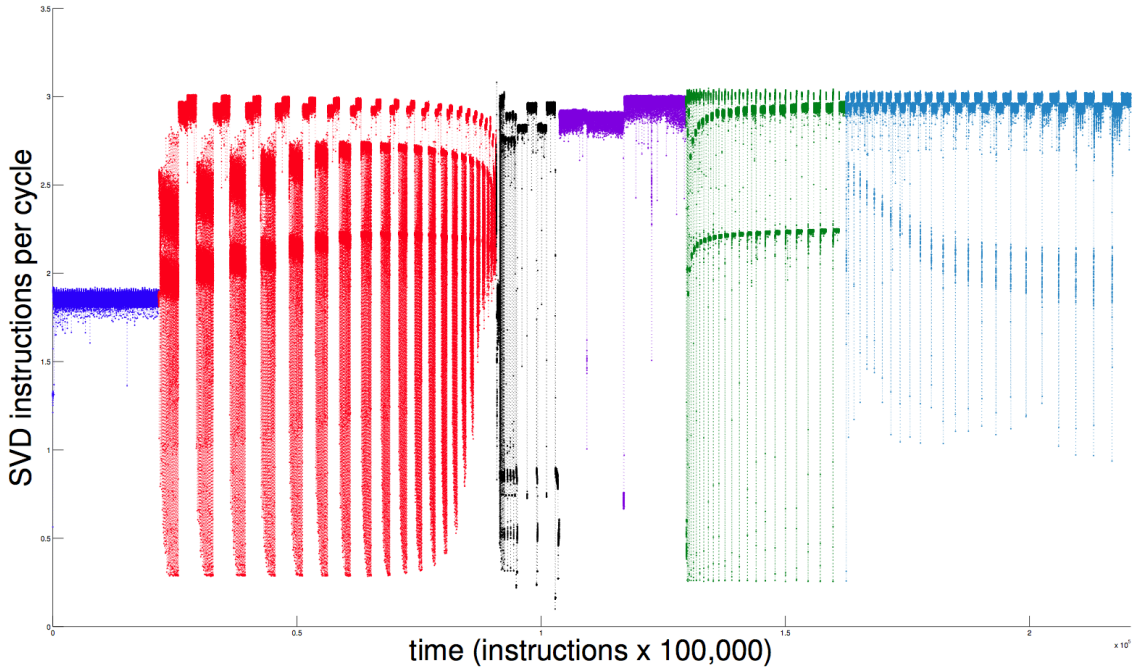

Figure 5: SVD IPC Time Series with Regimes Colored

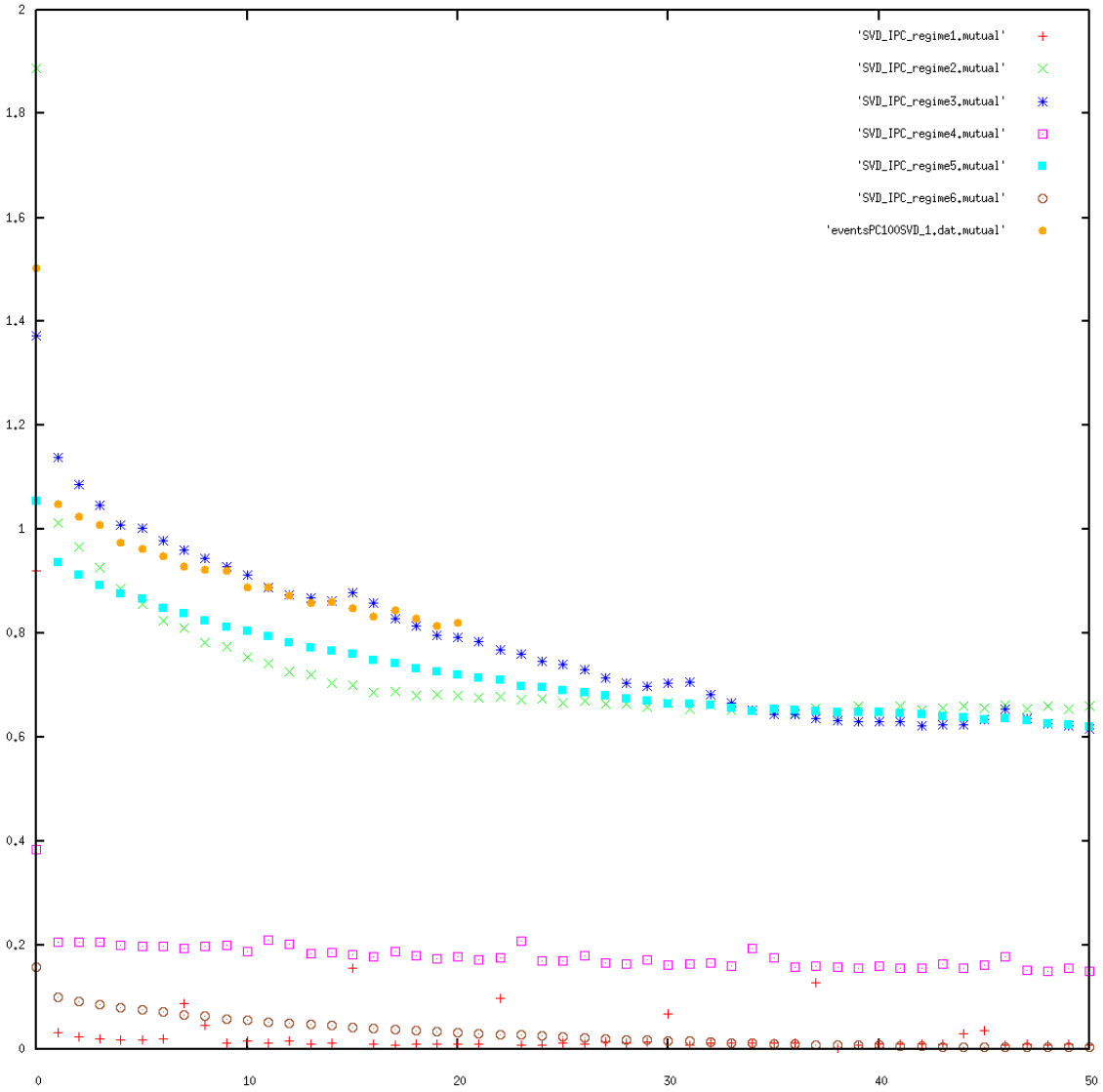

Figure 6: SVD mutuals for full and regimes

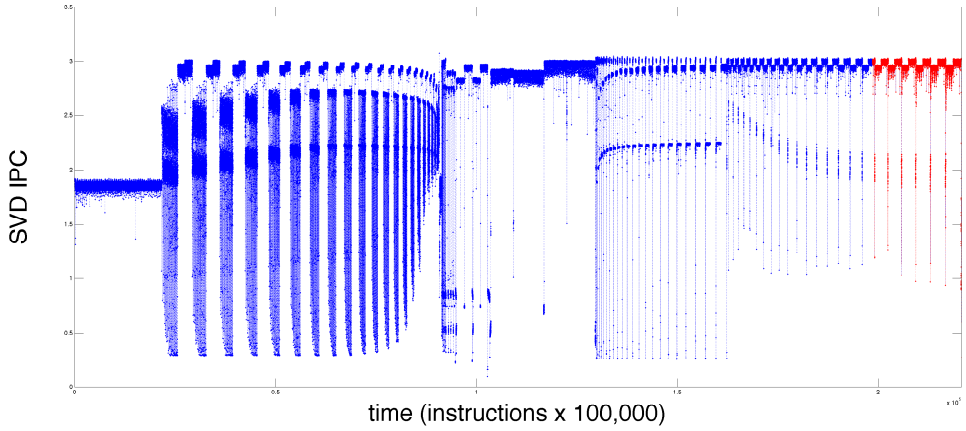

Figure 7: SVD IPC Full Time Series

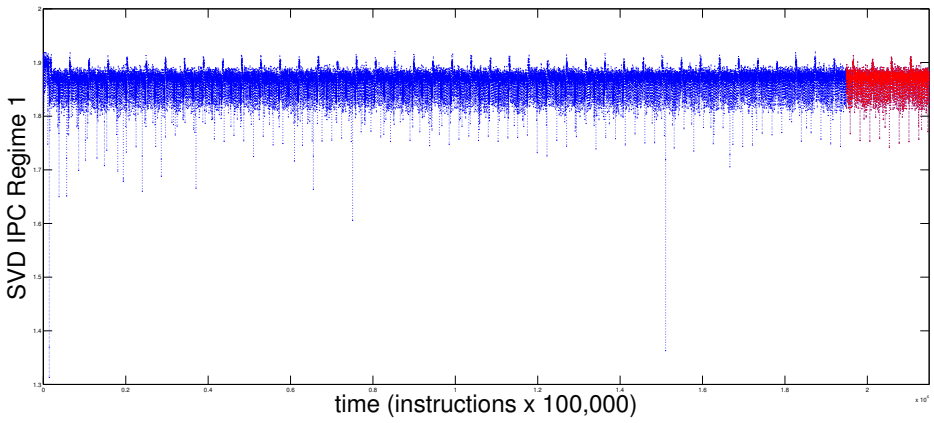

Figure 8: SVD IPC Regime 1 Time Series

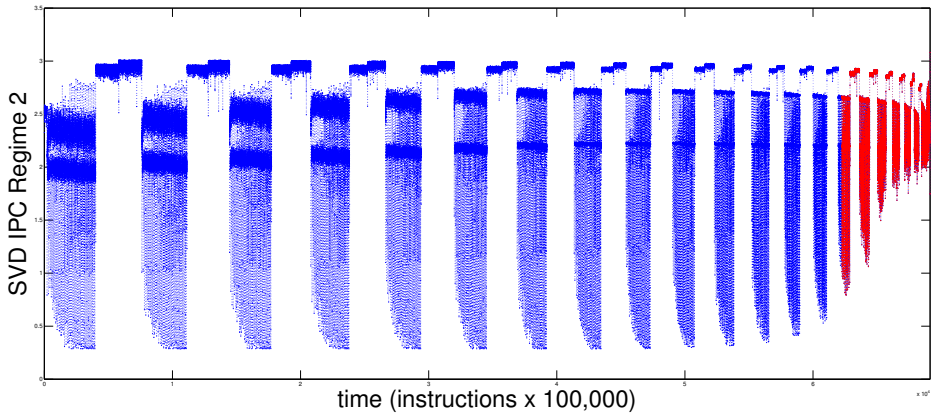

Figure 9: SVD IPC Regime 2 Time Series

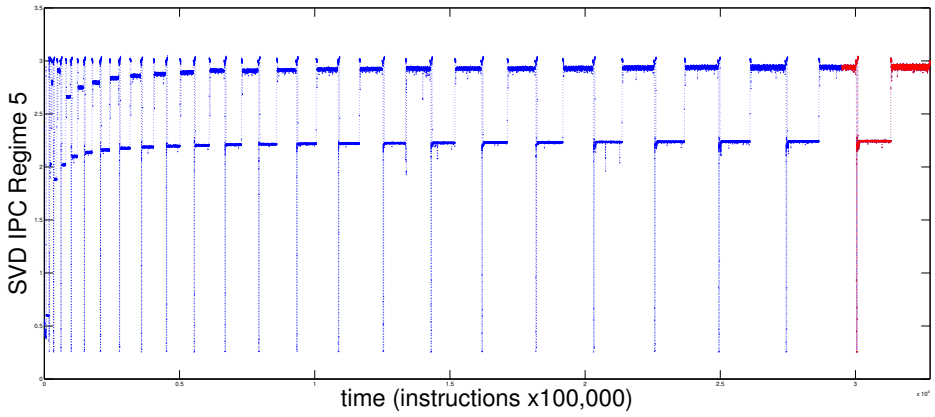

Figure 10: SVD IPC Regime 5 Time Series

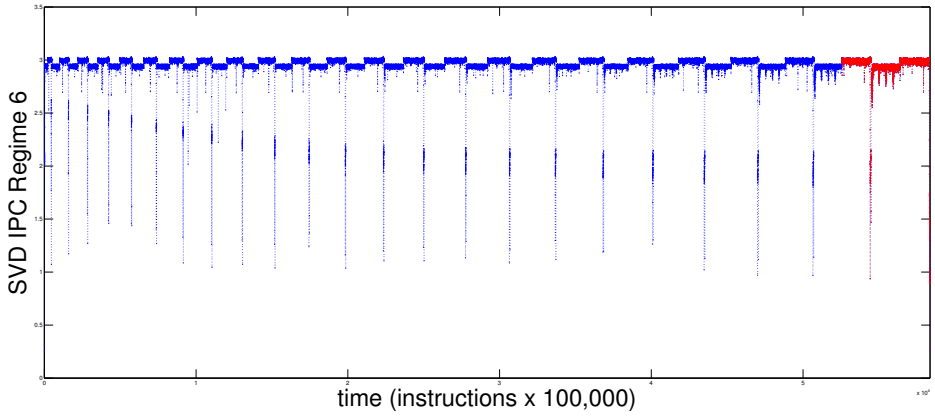

Figure 11: SVD IPC Regime 6 Time Series