Comparing Linear and Nonlinear Models for Forecasting of Computer System Performance

Joshua Garland
CSCI 5753 Final Paper
University of Colorado, Boulder

Abstract

Prior work has found computer systems to be nonlinear dynamical systems that exhibit complex and sometimes even chaotic behavior. This makes prediction difficult—and effectively impossible if one uses methods that rely on linearity and/or time invariance. In this paper, we construct both linear and nonlinear models of computer performance on an Intel i7. I then use these models to predict instructions per cycle of two benchmarks in the SPEC cpu2006 suite as a method for comparing the effectiveness of each model. I demonstrate that both models are effective in their own domain and propose directions for future work.

1 Introduction

To fully understand modern computer performance it is vital to first build effective and informative models which capture the true complexity of the underlying system. What this model should be however is up for debate. Traditional approaches use linear, time-invariant (and often stochastic) methods, e.g., autoregressive-moving-average(ARMA), multiple regression, etc. [10, 14]. While these models are widely accepted, and for the most part easy to construct, they do not take into account any of the nonlinear interactions which also have an effect on the computers performance. As stated in [14]: "Though many other factors, such as temperature and voltage levels, may influence processor performance, they were not taken into consideration in this work". Moreover, [14] goes on to say that "Deriving such a model can be complex, because it may require considerable insight into the internal machine organization".

An alternative approach, which also captures the complex nonlinear interactions, is to model computer performance as a nonlinear deterministic dynamical system [2, 3, 9, 15, 16], or as a collection of nonlinear dynamical systems, i.e., an iterated function system [1]. In this view, the register and memory contents are treated as state variables of these dynamical systems. The logic hard-wired into the computer, combined with the software executing on that hardware, defines the system's dynamics—that is, how the state variables change after each cycle of the microprocessor.

To gain insight into which of these models is more appropriate, I will build and evaluate two models of computer performance, one linear (multiple regression) and one nonlinear (lorenz method of analogues). To evaluate the effectiveness of these models I will use them to predict the instructions per cycle (IPC) of two programs[1] from the SPEC cpu2006 benchmark suite[2].

The rest of this paper is organized as follows. Section 2 will go over the experimental methods and data collection techniques I used to collect the performance traces. Section 3 reviews the ideas of multiple regression, which will closely follow [14]. Section 3 will also introduce the nonlinear dynamics techniques needed to construct the nonlinear model. Section 4 will go through the actual construction of these models. Section 5 we will make predictions using these models and compare their effectiveness. Section 6 will discuss future directions for this work.

2 Experimental Configuration

The time-series data for these experiments was collected on an Intel Core® i7-2600 CPU running the 2.6.38-8 Linux kernel. This Nehalem CPU has eight cores running at 3.40Ghz and a cache size of 8192 KB. These performance traces are recorded from the chips hardware performance monitors (HPM). HPMs are specialty registers on board most modern cpus whose sole purpose is to log hardware event information, e.g., instructions executed, cache misses, branches mispredicted, etc. Unlike past linux kernel, which required kernel modification to access these counters, this kernel natively supports the collection of data from HPMs through the linux `perf_events` interface. While these counters can be accessed directly it is useful to have helper libraries and programs to ac-

[1] 482.sphink3 and 403.gcc
[2] A similar evaluation technique as the one used in [14].

cess these registers while not perturbing the underlying dynamics.

To this end, I utilize a helper library, libpfm4 as well as PAPI v. 4.4.0[5] (Performance Application Programming Interface). PAPI is a performance API which allows "easy" interface to the linux `perf_events` through libpfm4. One major advantage to using PAPI is that it provides reliable portability across several platforms. PAPI accomplishes this by abstracting away native event codes, which are often platform dependent, by giving a set of predefined event names to common hardware events which can be accessed through the API. These preset events give access to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit, and pipeline status.[5] This layer of abstraction allows similar source code to record performance traces from multiple platforms without the need to remap all the native event codes. This is a huge advantage when computer performance traces must be done on multiple platforms. In addition to portability PAPI also provides safeguards for register overflow, which can be quite common depending on what events are being recorded.

PAPI provides two profiling designs. The first involves injecting PAPI calls into the source code of a program. This allows a programmer to see where time is being spent in particular sections of code. The second (and the one I use) is to periodically interrupt a programs execution and record hardware performance counters. As I am interested in profiling the overall dynamics of a program as it executes and as I do not want to modify the source code of these programs this is the ideal setup.

To accomplish this period icing sampling I use a python script that interfaces with PAPI. The script takes as input, an executable to profile, the name of the events we want PAPI to record, for example, `PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_L2_TCM`, and the number of instructions to execute between interrupts. Note that the choice to interrupt after a number of instructions versus after a number of cycles effects the temporal dynamics of the observations. With this in mind, after some thought it seems that the more natural choice is to interrupt on instructions rather than on cycles. This choice is corroborated in [15, 16], although interrupt on cycles is also done in these types of experiments as can be seen in [14]. It seems that either choice is fine as long as the temporal flow[3] of the resulting model is understood in the proper units[4].

The next consideration is how frequently I should interrupt and record data. If I interrupt too frequently I may taint the performance I am attempting to observe, but if I interrupt too infrequently I can easily miss any dynamics that occurs on a quick time scale. To make this choice I will follow the lead of [15, 16] and interrupt the same program at several different instruction counts and compare the Fourier spectrums of the resultant time series. According to [15], what I should see is a range of interrupt values for which the power spectrum is persistent. Then at some point the spectrum will change, this will signify the performance is being effected by the measurement device. I then choose the lowest interrupt value which resulted in a persistent spectrum.

Figure 1 provides the Fourier spectrum and time series traces resulting from measuring 482.sphink3[5] every x instructions[6]. I started the interrupt frequency at every 500 million instructions to approximately emulate the choice of 250 million cycles given in [14]. This however resulted in approximately 5 data points, which is not enough to have any statistical significance. I then lowered our interrupt to every 250 million instructions which resulted in 27 points. While this could be enough to build some models there seems to be a great amount of information being lost, as is clear in **Figure 1**(a). If I however lower the interrupt to every 1 million instructions we begin to see the Fourier spectrum which persists until an interrupt of every 500 instructions. I also see a lot more of the temporal dynamics, that was not present previously. Interestingly enough, if I lower the interrupt to every 100,000 instructions the spectrum is the same, however much more detail can be seen in the second half of the signal. Now the choice comes down to performance over head. Interrupting every 1 million instructions comes with a 12.5% overhead, where as interruption every 100,000 instructions results in a overhead of 39.5%. These seem close enough to me that building a model of both traces and comparing them seems useful. Thus I choose an interrupt rate of every 100,000 instructions and every 1 million instructions and I will compare the results and decide if a 27% increase in performance overhead is worth it.

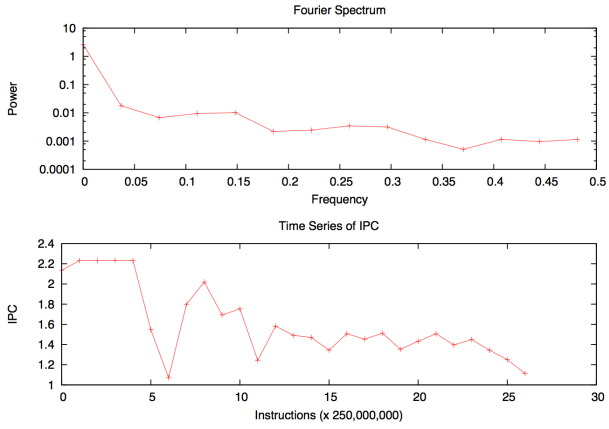
The programs I will construct these models on are 403.gcc and 482.sphink3 from the SPEC cpu2006 benchmark suite. Originally following [14] I wanted to use 176.gcc and 179.art, however after several attempts at building the SPEC cpu2000 benchmark suite on the Nehalem, it would seem that there are several incompatibility issues between the architecture SPEC2000 expects and the new architecture of the Nehalem. Even doing fresh builds of the program from scratch (rewriting all the make files and compiling by hand) still did not work. For this reason I obtained a copy of SPEC cpu2006 and built that on this machine. 403.gcc is an integer benchmark like 176.gcc and 482.sphink3 is a floating point benchmark like 179.art.

[3]This may not seem important for the multiple regression technique but it is very important in the nonlinear case as will be seen later.

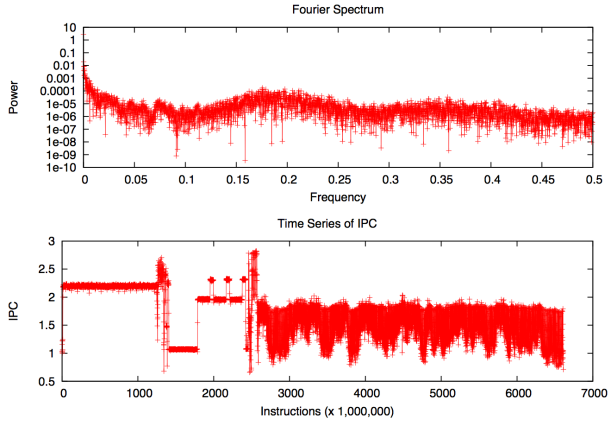
[4]Either instructions or cycles.

[5]This was also done on 403.gcc but results were identical and thus for brevity only these are reported.

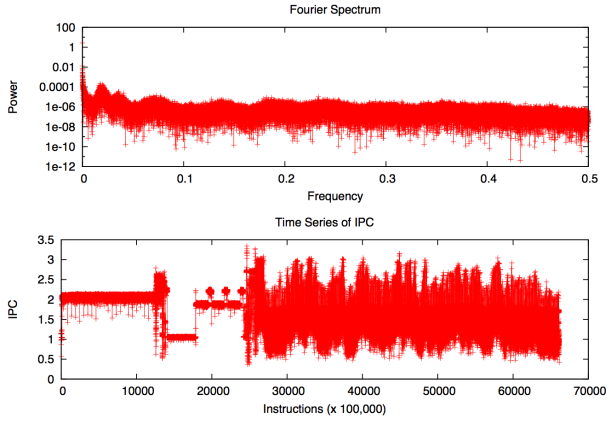
[6]The interrupt rate is given in the x-axis label



(a)



(b)



(c)

Figure 1: Time series and Fourier spectrums for several different interrupt rates.

Once I have the time series collected I will compare the models in the following way. For the purposes of validation, I will hold back the last 10% of the time series. I will then build a model using the first 90% of the trace and compare the models predictions to the last 10% of the true signal. As an error metric, I will compute the Root Mean Squared Prediction Error (RMSPE) between the true and predicted signal. An RMSPE of 51 cache misses per 100,000 instructions, for instance, means that, on average, the forecasting scheme predicted the cache-miss rate to within $\pm$ 51 over the next 100,000 instructions

3 Background

3.1 Multiple Linear Regression

Multiple linear regression is a linear statistical technique used to predict and/or explain the outcome of some event, often called a response variable R, through a linear combination of explanatory variables E_i. That is

$$R = \sum_{i=1}^{n} \alpha_i E_i$$

The α_i are typically chosen to minimize the RMSPE[14]. In our case the response variable we wish to predict is IPC, we choose as explanatory variables: instructions retired, total L2 cache misses, number of branches taken, total L2 instruction cache misses, total L2 instruction cache hits and total missed branch predictions.[7]. It is important to verify that there is no pairwise correlation between any explanatory variables as well as between explanatory and response variables. In addition one needs to verify for each model the following assumptions taken from [10]:

1. The true relationship between the response variable and the the explanatory variables is linaer.
2. The explanatory variables are non stochastic and measured without any error.
3. The model errors are statistically independent.
4. The errors are normally distributed with zero mean and a constant standard deviation.

3.2 Dynamical systems

The mathematical machinery developed in the nonlinear dynamics community offers adaptive methods that can aid in reconstructing hidden dynamics and forecasting computer system performance. The following section introduces a few basic concepts from that field. Building upon

[7]These choices are to emulate the work of [14].

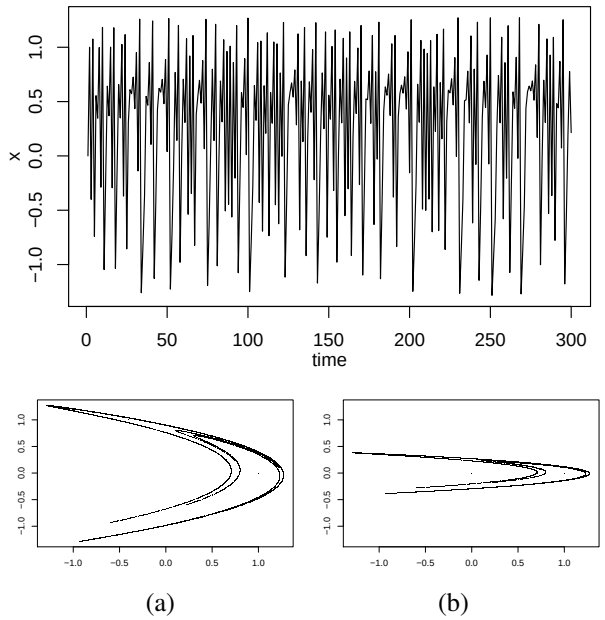


Figure 2: *Top:* a time-series trace of the x variable of the Hénon map. *Bottom:* (a) a state-space plot of x vs. y (b) a delay-coordinate embedding of the x variable

that machinery, we then demonstrate how to use the *time history* of a variable, like IPC, to *adaptively* build *nonlinear* forecast models that accurately capture a computer system's *dynamics*—in ways that static and/or linear models simply cannot.

3.2.1 A simple dynamical system example

An important insight from the domain of nonlinear dynamics is that a simple *deterministic* system can actually appear to have *stochastic* behavior—particularly when one can only measure one of its variables. To illustrate this effect, we use the Hénon map—a two dimensional dynamical system—which maps a point (x_n,y_n) at time n to a new point (x_{n+1},y_{n+1}) at time $n+1$:

$$x_{n+1} = y_n + 1 - 1.4x_n^2$$
$$y_{n+1} = 0.3x_n$$

Though this system of equations contains no source of randomness, a time-series trace of its x variable, as shown in **Figure 2**, appears to show stochastic behavior. The deterministic nature of the Hénon system becomes immediately apparent, however, if one plots x against y—the so-called *state-space view* of the dynamics—as shown in **Figure 2**(a). The challenge in any real-world application, of course, is that one rarely knows, let alone has access to, all of a system's state variables. The *delay-coordinate embedding* technique described in the following section addresses this challenge, allowing one to reconstruct the full dynamics from a single stream of measurements.

3.2.2 Reconstructing hidden dynamics

Another key insight from nonlinear dynamics is that the coupling between a system's state variables can be exploited to reconstruct hidden effects. If we plot x_n vs x_{n-1}, for instance—as shown in **Figure 2**(b)—we can reconstruct some critical information about the dynamics of the Hénon map. Note that while the geometry of the reconstruction is not identical to the true dynamics in part (a), its general shape—formally, its topology—is the same. The implications here are important: a simple mathematical transformation on the x_n allowed us to approximate the full dynamics of the Hénon map even though we never measured y_n. To gain an intuition for why this works, consider the equations for the Hénon map. The evolution of x_n is governed by its previous value (x_{n-1}) and also by the previous value of y; the next value of y is dictated by the previous value of x. Because of this, one can deduce something about the *values* of state variables from the *evolution* of others. One cannot use this trick to reconstruct the unknown system equations, of course, but one *can* use it to reconstruct the topology of their solutions. That is, the shapes in **Figure 2**(a) and (b) are guaranteed to have the same fundamental shape if the delay-coordinate embedding is done correctly.

Computer systems are dynamical systems—like the Hénon map, but obviously far more complex. When one measures a single performance metric, like instructions per cycle or L2 cache misses, that measurement contains the indirect effects of some of the system state variables. Viewed in the time domain, such a trace can *appear* stochastic, just like x_n appears stochastic in the Hénon system. But we can use delay-coordinate embedding to reconstruct its hidden deterministic dynamics, as described in the following section, and then build forecast models that capture and exploit the geometry of the reconstruction.

3.2.3 Delay-coordinate embedding

Delay-coordinate embedding [17, 18, 19] allows one to reconstruct a system's full state-space dynamics from data like the time series at the top of Figure 2—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 [19] formally proves that the delay-coordinate map

$$F(\tau, m)(x) = ([x(t)\ x(t+\tau)\ \ldots\ x(t+m\tau)]) \quad (1)$$

from a d-dimensional smooth compact manifold M to Re^{2d+1}, where t is time, is a diffeomorphism on M [18]—in other words, that the reconstructed dynamics and the true (hidden) dynamics have the same topology.

This is an extremely powerful result because it guarantees that F is a good model of the system. In the context of Figure 2, for instance, it means that the delay-coordinate embedding of part (b) and the true underlying dynamics—part (a)—are topologically equivalent. And *that* means that one can model the full system dynamics without measuring (or even knowing) every one of its state variables. This is the basis for the modeling and forecasting techniques discussed in Section 3.2.4. There are some issues, however; equation 1 contains two free parameters, the delay τ and the dimension m, that must be estimated from the data. All of this is described in more detail in the following section, where we show how to reconstruct and analyze the dynamics of a computer performance trace.

3.2.4 Modeling Computer Performance with Nonlinear Dynamics

The first step in any study of the dynamics of computer system performance is to choose a metric. Delay-coordinate embedding only requires that the time series be a smooth function of some subset of the (unknown) state variables, which makes the choice of metric quite flexible.

Figure 3 shows a trace of the IPC that occurred on an Intel Nehalem® as it executed 403.gcc and 482.sphink3 from the SPEC 2006 benchmark suite. A close look at this time series, reveals interesting structure: patterns that are almost periodic, but not quite.

For the following model, we will consider a "stored-program computer," *i.e.,* a standard von Neumann architecture, as a deterministic nonlinear dynamical system. In a stored-program computer, the current state—both instructions and data—are stored in some form of addressable memory. The contents of this memory are, as established in [15], part of the state X of the computer. Other components of the computer, such as external memory and video cards, also play roles in its state. Those roles depend on the decisions made by the computer designers—how things are implemented and connected—almost all of which are proprietary. In order to distinguish known and unknown effects, we define the state space X as a composition of the addressable memory elements $\vec{x}_m$ and the unknown implementation variables $\vec{x}_u$:

$$X = \{\vec{x} \mid \vec{x} = [\vec{x}_m,\ \vec{x}_u]\}$$

The distinction between $\vec{x}_m$ and $\vec{x}_u$ is important because the dynamics of a running computer have two distinct sources: a map $\vec{F}_{code}$ that acts on the addressable memory $\vec{x}_m$ directly, based on the program instructions, and a map $\vec{F}_{impl}$ that captures how the implementation affects

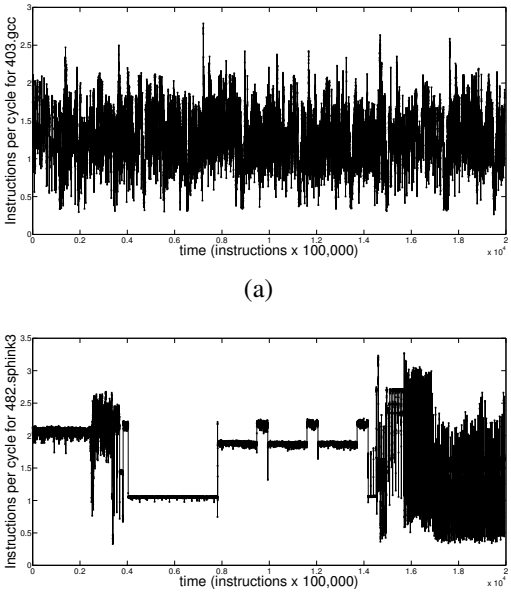


Figure 3: IPC traces of 403.gcc and 482.sphink3

the evolution of the computer state. The overall dynamics of the computer—that is, the mapping from its state at the n^{th} clock cycle to its state at the $n + 1^{st}$ clock cycle—is a composition of these two maps:

$$\vec{x}(t_{n+1}) = \vec{F}_P(\vec{x}(t_n)) = \vec{F}_{impl} \circ \vec{F}_{code}(\vec{x}(t_n))$$

where $\vec{F}_P$ is the overall performance dynamics of the computer. An improved design for the processor, for instance—that is, a "better" $\vec{F}_P$—is a change in $\vec{F}_{impl}$. The form of the map $\vec{F}_{code}$ is dictated by the combination of the computer's formal specification (`x86_64`, for the Intel Nehalem®) and the software that it is running. Both $\vec{F}_{impl}$ and $\vec{F}_{code}$ are nonlinear and deterministic, and their composed dynamics must be modeled and predicted in order to produce an accurate forecast of the state X.

4 Construction of the models

4.1 Multiple regression model

Recall we wish to explain the response variable IPC as a linear combination of possibly several explanatory variables. Thus, we need to decide out of the possible explanatory variables which of these explain the most variation, i.e., which explanatory variables contribute most to the response variable. This is often called a reduction of model. The process I use to generate my reduction in model is called stepwise backward elimination and is

described in [7]. In this technique you start with a full model, i.e., you measure everything you think might be an explanatory variable. In this case I measured, instructions retired, total L2 cache misses, number of branches taken, total L2 instruction cache misses, total L2 instruction cache hits and total missed branch predictions[8]. Each of these measurements were normalized per cycle as was done in [14]. First I choose an $\alpha_{crit} = 5\%$, I then iterate the following method: remove the factor (explanatory variable) with the highest p-value higher than α_{crit}, until every factor has a p-value less than α_{crit}. Upon convergence of this iteration I concluded that the minimal reduced model that predicts IPC (for all four traces) using the other factors is L2 total cache misses, number of branches taken and L2 instruction cache misses. Below I provide the final linear multiple regression model as constructed by R for each of the four performance traces.

gcc models:

```
>gcc.data.100 = read.table('100gccEventsNormalized.dat',header = TRUE)
> M = gcc.data.100
> cutoff=floor(length(M$ipc)*.9)
> MT = M[1:cutoff,]
> MP = M[cutoff:length(M$ipc),]
> TLM = lm(ipc~.,data = MT)
> summary(TLM)

Call:
lm(formula = ipc ~ ., data = MT)

Residuals:
     Min       1Q   Median       3Q      Max
-1.69385 -0.07558 -0.01457  0.07083  1.12422

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.453141   0.003085  146.87   <2e-16 ***
PAPI_L2_TCM -8.283048   0.119255  -69.46   <2e-16 ***
PAPI_BR_TKN  5.655505   0.013711  412.48   <2e-16 ***
PAPI_L2_ICM -7.453756   0.274533  -27.15   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.1323 on 40991 degrees of freedom
Multiple R-squared: 0.8422,Adjusted R-squared: 0.8422
F-statistic: 7.29e+04 on 3 and 40991 DF,  p-value: < 2.2e-16



 > M = gcc.data.1m
> cutoff=floor(length(M$ipc)*.9)
> MT = M[1:cutoff,]
> MP = M[cutoff:length(M$ipc),]
> TLM = lm(ipc~.,data = MT)
> summary(TLM)

Call:
lm(formula = ipc ~ ., data = MT)

Residuals:
     Min       1Q   Median       3Q      Max
-0.72819 -0.06104 -0.01975  0.05313  0.65053

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.441058   0.008751   50.40   <2e-16 ***
PAPI_L2_TCM  -8.546761   0.371141  -23.03   <2e-16 ***
PAPI_BR_TKN   5.743469   0.038726  148.31   <2e-16 ***
PAPI_L2_ICM -13.050847   1.175610  -11.10   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.1056 on 4095 degrees of freedom
Multiple R-squared: 0.877,Adjusted R-squared: 0.8769
F-statistic:  9732 on 3 and 4095 DF,  p-value: < 2.2e-16
```

Sphinx models:

```
> M = sphink.data.100
> cutoff=floor(length(M$ipc)*.9)
> MT = M[1:cutoff,]
> MP = M[cutoff:length(M$ipc),]
> TLM = lm(ipc~.,data = MT)
> summary(TLM)

Call:
lm(formula = ipc ~ ., data = MT)

Residuals:
     Min       1Q   Median       3Q      Max
-0.92359 -0.15500  0.01614  0.13781  2.77624

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.327058   0.002809  472.41  < 2e-16 ***
PAPI_L2_TCM -37.331944   0.183230 -203.74  < 2e-16 ***
PAPI_BR_TKN   2.829377   0.011922  237.33  < 2e-16 ***
PAPI_L2_ICM -21.005419   4.636498   -4.53  5.9e-06 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.2863 on 59519 degrees of freedom
Multiple R-squared: 0.7502,Adjusted R-squared: 0.7502
F-statistic: 5.958e+04 on 3 and 59519 DF,  p-value: < 2.2e-16


> M = sphink.data.1m
> cutoff=floor(length(M$ipc)*.9)
> MT = M[1:cutoff,]
> MP = M[cutoff:length(M$ipc),]
> TLM = lm(ipc~.,data = MT)
> summary(TLM)

Call:
lm(formula = ipc ~ ., data = MT)

Residuals:
     Min       1Q   Median       3Q      Max
-0.71111 -0.10114 -0.00425  0.07449  1.10777

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    1.02584    0.01290  79.494  < 2e-16 ***
PAPI_L2_TCM  -29.67365    0.71161 -41.699  < 2e-16 ***
PAPI_BR_TKN    4.56663    0.05698  80.146  < 2e-16 ***
PAPI_L2_ICM -164.63044   35.67407  -4.615 4.02e-06 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.2379 on 5947 degrees of freedom
Multiple R-squared: 0.8328,Adjusted R-squared: 0.8327
F-statistic:  9874 on 3 and 5947 DF,  p-value: < 2.2e-16
```

One thing we now must verify is that the explanatory variables are not correlated. I use R to check this and the result is as follows [9]:

```
> cor(M$PAPI_L2_TCM,M$PAPI_BR_TKN)
[1] -0.39147
> cor(M$PAPI_L2_TCM,M$PAPI_L2_ICM)
[1] 0.349611
> cor(M$PAPI_BR_TKN,M$PAPI_L2_ICM)
[1] 0.02690624
> cor(M$ipc,M$PAPI_L2_ICM)
[1] -0.1733815
> cor(M$ipc,M$PAPI_L2_TCM)
[1] -0.7200855
> cor(M$ipc,M$PAPI_BR_TKN)
[1] 0.7271015
```

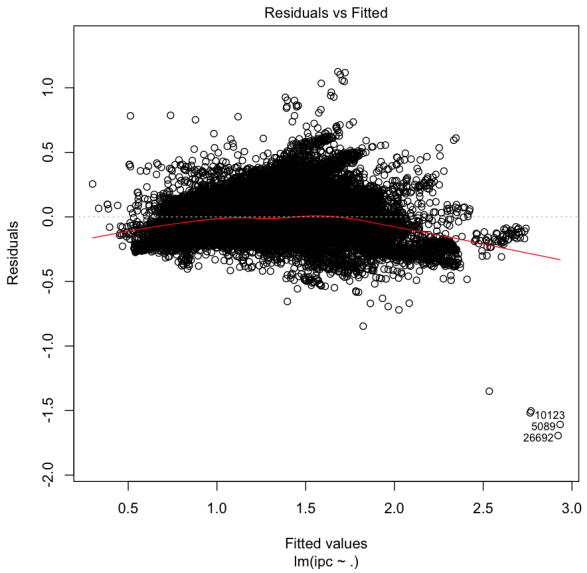
This shows that pairwise, each explanatory random variable is uncorrelated as well as being uncorrelated with the response variable. The final step in constructing a multi regression model is to verify that the model satisfies the underlying assumptions used in constructing such a model. For this I will use the techniques in [10]. The underlying assumptions were given in 3.

We will verify the model for 403.gcc and 482.sphink3 simultaneously.[10] The figures that are described in Jain for visual verification of the model are provided in **Figure 4** and **Figure 5**. The first assumption for both of these models is that the true relationship between response and explanatory variables is in fact linear. As this is a multi regression model this assumption is very hard to verify.

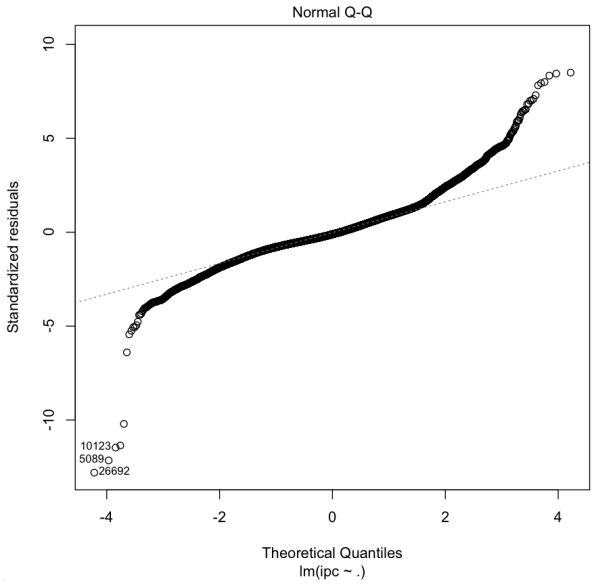
[8] While this was not everything that could be measured this most closely modeled the choices of event set made in [14]

[9] This was done for each trace but for brevity only one is reported.

[10] For both programs discussed here both interrupt values were verified but only one for each program as results were identical.

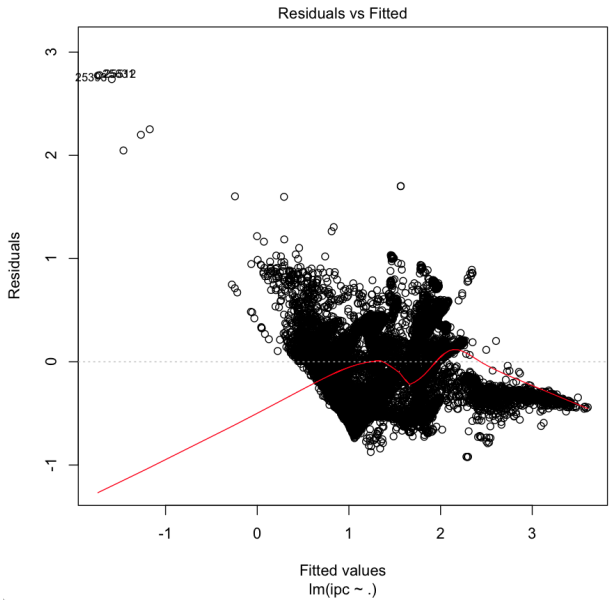


(a)

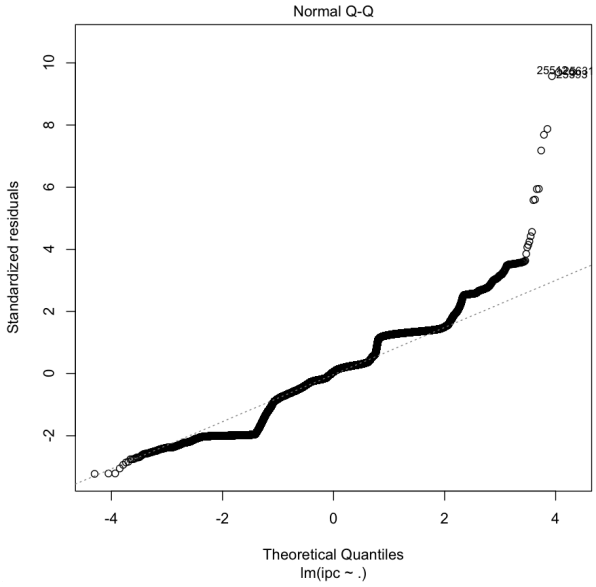


(b)

Figure 4: Visual checks of the assumptions made in [10] for 403.gcc (a) is the independent error visualization (b) is the normality of error distribution check



(a)



(b)

Figure 5: Visual checks of the assumptions made in [10] for 482.sphink3 (a) is the independent error visualization (b) is the normality of error distribution check

However, this is the fundamental assumption of this section of the paper, i.e., a linear model is appropriate. Thus we continue although I find this bothersome. If you consider any of the performance traces they look far from linear. The second assumption is that the predictor variable x is nonstochastic and it is measured without any error. This can be tested by looking for trends in a fitted vs residual plot. If there is a trend in such a plot this would indicate a dependence of error on the predicted response. This plot can be seen in **Figure 4**(a) and **Figure 5**(a) respectively. The lack of trend in each of these plots would suggest this assumption is true. The third assumption is the model errors are statistically independent. To visually check this we plot a normal quantile-quantile plot of error, these can be seen in **Figure 4**(b) and **Figure 5**(b) respectively. This assumption is not verified in my opinion, if you consider both of these figures it would seem that a nonnormal trend is present. More so for 403.gcc than 482.sphink3 but both present nonlinear trends. This would suggest to me that the models may yield spurious conclusions due to the lack of independence in the error. The final assumption is the errors are normally distributed with zero mean and a constant standard deviation. This can be verified by again considering **Figure 4** (a) and **Figure 5**(a) respectively since there is no trend in the spread of the error we have visually verifed this final assumption.

So while each model yields an R^2 which is very high some assumptions about the model could not be visually verified while others could. This is an art however and as such we proceed. If we get results which seem off, the failed tests above will be a good place to being investigation.

4.2 The nonlinear model

The framework outlined in Section 3.2.4 lets us use nonlinear time-series analysis techniques to model and analyze $\vec{F}_P$.

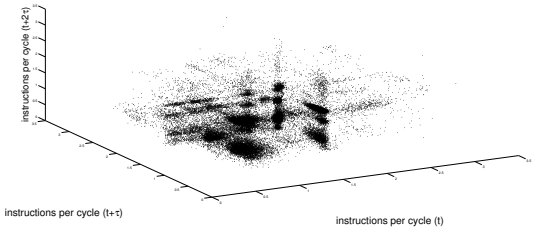
4.2.1 Delay Coordinate Embedding

To construct the nonlinear models, we need to perform delay-coordinate embedding on each of the four IPC traces (two of which are depicted in **Figure 3** which comes down to estimating the free parameters τ and m and constructing delay vectors. The standard method [4, 11] to estimate the delay τ is to choose the first minimum of the mutual information curve [8]. This choice in theory, minimizes the amount of mutual information being shared across coordinates of the delay vector. This is the nonlinear equivalent of choosing approximately orthogonal vectors to span some space. Choosing the minimum ensures that they are as "orthogonal" as possible. More on this can be found in [8]. The theoretical specification on τ only requires it to be positive, however in practice the choice of τ can have drastic effects on the usefulness of the model[11]. In the case where we interrupt every 100,000 instructions the first minimum occurs at $\tau = 10$. This implies the delay which minimizes mutual information and maximizes the nonlinear orthogonality of the spanning set is 1 million instructions. If we consider the mutual curve for the 1 million instruction interrupt trace we see the first minimum occurs at $\tau = 1$, which corroborates the findings of the 100,000 instruction trace. Thus, for traces where interrupts are performed every 100,000 instructions we choose $\tau = 10$ and traces where interrupts were performed every 1 million instructions a τ of 1 is chosen.

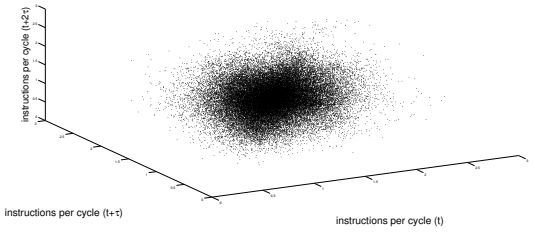
The second parameter we much estimate is the embedding dimension m. There are very strict requirements on this parameter in theory [12]. The embedding dimension m must be greater than $2d+1$ where d is the dimension of the original dynamics. As these dynamics and thus this dimension d is unknown this seems to be an impossible task. Fortunately, there are approximations which can be made for m the most standard method is known as false-near neighbors and is covered in detail in [12]. The big idea is choose a range of dimensions for which the fraction of false-near neighbors, drops below an error threshold. This threshold is highly heuristic and in practice can range from 10-30%. Although depending on the error present in the measurement it can actually be higher than this. When I did this analysis I concluded that the Nehalem seems to have a dynamic which can be properly embedded in a 12 dimensional space. This dimension must be verified using persistence of dynamical invariants. This discussion and technique are beyond the scope of this paper. Hence I choose $m = 12$.

A three-dimensional projection of two of these 12 dimensional models can be seen in **Figure 6**. The coordinates of each point on the resulting plot, shown in **Figure 6**, are differently delayed elements of the IPC time series[11] $y(t)$: 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 6 (a)—is an indication of determinism. The lack of structure in **Figure 6** (b) is bothersome and may represent large amounts of noise present in this trace.

[11](a) is the embedding of 482.sphink3 and (b) is the embedding of 403.gcc



(a)



(b)

Figure 6: 3D projections of a delay-coordinate embedding of the traces from Figure 3 with a delay (τ) of 100,000 instructions (a) is the model of 482.sphink3 and (b) is the model of 403.gcc.

5 Forecasting Computer Performance

5.1 Multiple Regression: Linear forecasting

Now that we have a linear model of each time series constructed we can use this to predict future IPC values. To do this we use the `predict` function in R. This allows us to predict forward the final 10% of each signal based on the model we constructed using the first 90%. We then compute the RMSPE for each signal.

Program	Interrupt Rate	RMSPE
403.gcc	100,000	0.2495
403.gcc	1,000,000	0.2416
482.sphink3	100,000	0.4267
482.sphink3	1,000,000	0.3079

Table 1: The RMSPE for the linear forecasting models

These are fairly good prediction results. Notice that the 1 million interrupt traces did better in both cases than the 100,000 interrupt rate. Arguing that an interrupt of 1 million is actually better for modeling. This would imply that the extra overhead is definitely not worth it.

5.2 LMA: Using dynamics in forecasting

In Section 4.2, I constructed a model of computer performance using nonlinear dynamics. In this section, I present a forecasting technique that captures and exploits the geometry of those dynamical models. I show that this technique is quite effective at predicting a computer's behavior.

Given a nonlinear model of a deterministic dynamical system, in the form of a delay-coordinate embedding like **Figure 6**, one can build effective forecast algorithms by capturing and exploiting the geometry of that model. Many nonlinear forecasting techniques have been developed by the dynamical systems community for this purpose (e.g., [6, 20]); perhaps the simplest is the "Lorenz method of analogues" (LMA), which is essentially nearest-neighbor prediction [13] in an embedded state space.

Even this simple algorithm—which builds predictions by looking for the nearest neighbor of a given point, then taking that neighbor's path as the forecast—works quite well when it applied to a delay-coordinate embedding of computer performance data.

Program	Interrupt Rate	RMSPE
403.gcc	100,000	0.47
403.gcc	1,000,000	0.3803
482.sphink3	100,000	0.6337
482.sphink3	1,000,000	0.5226

Table 2: The RMSPE for the nonlinear forecasting models

Note that in this case as well the 1 million interrupt performed better in both cases. Also notice that in each case the linear model did slightly better than this nonlinear model. It is worth considering however if this is a fair comparison. The linear model can only predict one time step ahead at any given time, i.e., given measurements of L2 total cache misses, instructions retired, L2 instruction cache misses and branch retirement rates at time t the multiple regression model can predict forward $t + 1$ fairly accurately. As seen in the previous section. However, this requires you to constantly be monitoring the program and given this model feedback. LMA on the other hand can predict into the future as far as one needs once it is trained. This is a vastly different prediction model and thus makes this comparison a difficult one to make.

6 Conclusions and Future Work

Both linear and nonlinear models were able to predict IPC of 403.gcc and 482.sphink3 with fairly good accuracy. The linear model regularly out predicted the nonlin-

ear model but only by a very small margin each time. The linear model, while it did better, needs far more data measured to construct, 5 HPMs, as well as constant measuring during the prediction process. On the other hand the nonlinear model can do almost as well while only measuring 2 HPMs and then no longer needing feedback from the system once the model is trained.

Personally, I am not satisfied with either of these models. It seems that there are both linear and nonlinear couplings that are occurring under the hood. Some of which are deterministic and some of which appear to be random. It would be interesting from here to develop models that incorporate both of these features to make a superior model. Another direction would be to apply a more sophisticated nonlinear forecasting technique instead of the incredibly naive technique used here. In conclusion, it is very hard to determine which model was superior as they both have advantages and shortcomings and performed very similarly.

References

[1] Z. Alexander, E. Bradley, J. Garland, and J. Meiss. Iterated function system models in data analysis: Detection and separation. *CHAOS*, 22(2), April 2012.

[2] Z. Alexander, T. Mytkowicz, A. Diwan, and E. Bradley. Measurement and dynamical analysis of computer performance data. In N. Adams, M. Berthold, and P. Cohen, editors, *Advances in Intelligent Data Analysis IX*, volume 6065. Springer Lecture Notes in Computer Science, 2010.

[3] H. Berry, D. Perez, and O. Temam. Chaos in computer performance. *Chaos*, 16, 2006.

[4] E. Bradley. Analysis of time series. In *Intelligent Data Analysis: An Introduction*, pages 199–226. Springer-Verlag, 2^{nd} edition, 2000. M. Berthold and D. Hand, eds.

[5] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999.

[6] M. Casdagli and S. Eubank, editors. *Nonlinear Modeling and Forecasting*. Addison Wesley, 1992.

[7] J. Faraway. *Practical Regression Analysis using R*. 2002.

[8] A. Fraser and H. Swinney. Independent coordinates for strange attractors from mutual information. *Physical Review A*, 33(2):1134–1140, 1986.

[9] J. Garland and E. Bradley. Predicting computer performance dynamics. In *Proceedings of the 10th international conference on Advances in intelligent data analysis X*, IDA'11, pages 173–184, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 2nd edition, 1991.

[11] H. Kantz and T. Schreiber. *Nonlinear Time Series Analysis*. Cambridge University Press, Cambridge, 1997.

[12] M. B. Kennel, R. Brown, and H. D. I. Abarbanel. Determining minimum embedding dimension using a geometrical construction. *Physical Review A*, 45:3403–3411, 1992.

[13] E. N. Lorenz. Atmospheric predictability as revealed by naturally occurring analogues. *Journal of the Atmospheric Sciences*, 26:636–646, 1969.

[14] T. Moseley, J. L. Kihm, D. A. Connors, and D. Grunwald. Methods for modeling resource contention on simultaneous multithreading processors. In *Proceedings of the 2005 International Conference on Computer Design (ICCD)*, October 2005.

[15] T. Myktowicz, A. Diwan, and E. Bradley. Computers are dynamical systems. *Chaos*, 19:033124, 2009. doi:10.1063/1.3187791.

[16] T. Mytkowicz. *Supporting experiments in computer systems research*. PhD thesis, University of Colorado, November 2010.

[17] N. Packard, J. Crutchfield, J. Farmer, and R. Shaw. Geometry from a time series. *Physical Review Letters*, 45:712, 1980.

[18] T. Sauer, J. Yorke, and M. Casdagli. Embedology. *Journal of Statistical Physics*, 65:579–616, 1991.

[19] F. Takens. Detecting strange attractors in fluid turbulence. In D. Rand and L.-S. Young, editors, *Dynamical Systems and Turbulence*, pages 366–381. Springer, Berlin, 1981.

[20] A. Weigend and N. Gershenfeld, editors. *Time Series Prediction: Forecasting the Future and Understanding the Past*. Santa Fe Institute Studies in the Sciences of Complexity, Santa Fe, NM, 1993.