Using the usl package

Analyze System Scalability in R with the Universal Scalability Law

Stefan Möding

June 17, 2016

The Universal Scalability Law is used to quantify the scalability of hardware or software systems. It uses sparse measurements from an existing system to predict the throughput for different loads and can be used to learn more about the scalability limitations of the system. This document introduces the usl package for R and shows how easily it can be used to perform the relevant calculations.

Contents

1 Version

This document describes version 1.6.0 of the usl package.

2 Introduction

Every system architect faces the challenge to deliver an application system that meets the requirements. A critical point during the design is the scalability of the system.

Informally scalability can be defined as the ability to support a growing amount of work. A system is said to scale if it handles the changing demand or hardware environment in a reasonable efficient and practical way.

Scalability can have two facets with respect to a computer system. On the one hand, there is software scalability where the focus is about how the system behaves when the demand increases, i.e., when more users are using it or more requests need to be handled. On the other hand, there is hardware scalability where the behavior of an application system running on larger hardware configurations is investigated.

The Universal Scalability Law (USL) has been developed by Dr. Neil J. Gunther to allow the quantification of scalability for the purpose of capacity planning. It provides an analytic model for the scalability of a computer system.

A comprehensive introduction to the Universal Scalability Law including the mathematical grounding has been published in [Gun07].

3 Background

Dr. Gunther shows in [Gun07] how the scalability of every computer system can be described by a common rational function. This function is *universal* in the sense that it does not assume any specific type of software, hardware or system architecture.

Equation (1) has the Universal Scalability Law where $C(N) = X(N)/X(1)$ is the relative capacity given by the ratio of the measured throughput $X(N)$ for load N to the throughput $X(1)$ for load 1.

$$C(N) = \frac{N}{1+\sigma(N-1)+\kappa N(N-1)} \tag{1}$$

The denominator consists of three terms that all have a specific physical interpretation:

Concurrency: The first term models linear scalability that would exist if the different parts of the system (processors, threads …) could work without any interference caused by their interaction.

Contention: The second term of the denominator refers to the contention between different parts of the system. Most common are issues caused by serialization or queueing effects.

Coherency: The last term represents the delay induced by keeping the system in a coherent and consistent state. This is necessary when writable data is shared in different parts of the system. Predominant factors for such a delay are caches implemented in software and hardware.

In other words: σ and κ represent two concrete physical issues that limit the achievable speedup for parallel execution. Note that the contention and coherency terms grow linearly respectively quadratically with N. As a consequence their influence becomes larger with an increasing N.

Due to the quadratic characteristic of the coherency term there will be a point where the throughput of the system will start to go retrograde, i.e., will start to decrease with further increasing load.

In [Gun07] Dr. Gunther proves that eq. (1) is reduced to Amdahl's Law for $\kappa = 0$. Therefore the Universal Scalability Law can be seen as a generalization of Amdahl's Law for speedup in parallel computing.

We could solve this nonlinear equation to estimate the coefficients σ and κ using a sparse set of measurements for the throughput X_i at different loads N_i. The computations used to solve the equation for the measured values are discussed in [Gun07].

The usl package has been created to subsume the computation into one simple function call. This greatly reduces the manual work that previously was needed to perform the scalability analysis.

The function provided by the package also includes some sanity checks to help the analyst with the data quality of the measurements.

Note that in [Gun07] the coefficients are called σ and κ when hardware scalability is evaluated but α and β when software scalability is analyzed. The usl package only uses `sigma` and `kappa` as names of the coefficients.

4 Examples of Scalability Analysis

The following sections present some examples of how the usl package can be used when performing a scalability analysis. They also explain typical function calls and their arguments.

4.1 Case Study: Hardware Scalability

The usl package contains a demo dataset with benchmark measurements from a raytracer software[1]. The data was gathered on an SGI Origin 2000 with 64 R12000 processors running at 300 MHz.

[1] `http://sourceforge.net/projects/brlcad/`

A number of reference images with different levels of complexity were computed for the benchmark. The measurements contain the average number of calculated ray-geometry intersections per second for the number of used processors.

It is important to note that with changing hardware configurations the relative number of *homogeneous* application processes per processor is to be held constant. So when k application processes were used for the N processor benchmark then $2k$ processes must be used to get the result for $2N$ processors.

Start the analysis by loading the usl package and look at the supplied dataset.

```
R> library(usl)
R> data(raytracer)
R> raytracer


   processors throughput
1           1         20
2           4         78
3           8        130
4          12        170
5          16        190
6          20        200
7          24        210
8          28        230
9          32        260
10         48        280
11         64        310
```

The data shows the throughput for different hardware configurations covering the available range from one to 64 processors. We can easily see that the benefit for switching from one processor to four processors is much larger than the gain for upgrading from 48 to 64 processors.

Create a simple scatterplot to get a grip on the data.

```
R> plot(throughput ~ processors, data = raytracer)
```

Figure 1 shows the throughput of the system for the different number of processors. This plot is a typical example for the effects of *diminishing returns*, because it clearly shows how the benefit of adding more processors to the system gets smaller for higher numbers of processors.

Our next step builds the USL model from the dataset. The `usl()` function creates an S4 object that encapsulates the computation.

The first argument is a formula with a symbolic description of the model we want to analyze. In this case we would like to analyze how the “throughput” changes with regard to the number of “processors” in the system. The second argument is the dataset with the measured values.

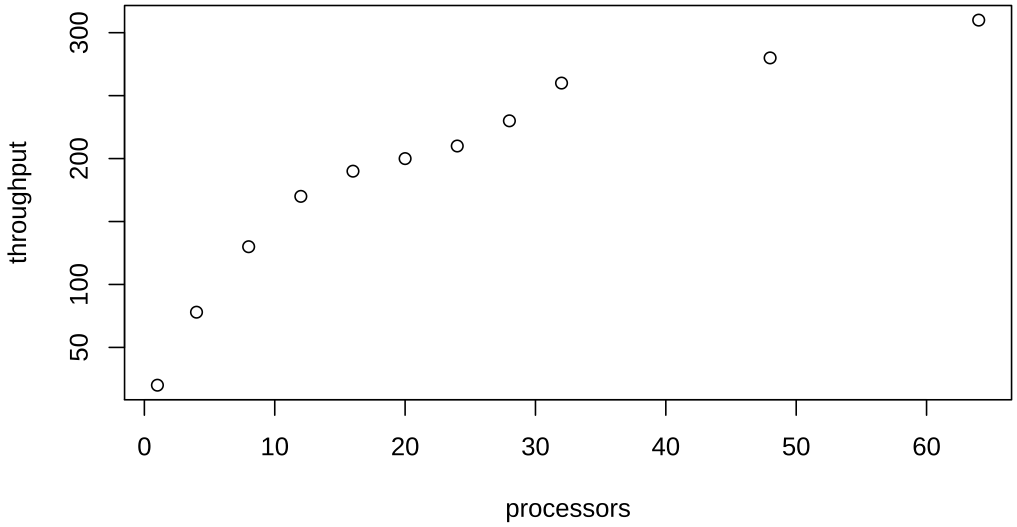

Figure 1: Measured throughput of a raytracing software in relation to the number of available processors

```
R> usl.model <- usl(throughput ~ processors, data = raytracer)
```

The model object can be investigated with the `summary()` function.

```
R> summary(usl.model)



Call:
usl(formula = throughput ~ processors, data = raytracer)

Scale Factor for normalization: 20

Efficiency:
   Min     1Q Median     3Q    Max
 0.242  0.408  0.500  0.760  1.000

Residuals:
   Min     1Q Median     3Q    Max
-12.93  -5.23   3.08   9.00  15.25

Coefficients:
       Estimate    Std. Error
sigma  0.05002394  0.00320929
kappa  0.00000471  0.00006923

Residual standard error: 9.86 on 9 degrees of freedom
Multiple R-squared: 0.988,Adjusted R-squared: 0.987
```

The output of the *summary()* function shows different types of information.

- First of all it includes the call we used to create the model.
- It also includes the scale factor used for normalization. The scale factor is used internally to adjust the measured values to a common scale. It is equal to the value $X(1)$ of the measurements.
- The efficiency tells us something about the ratio of useful work that is performed per processor. It is obvious that two processors might be able to handle twice the work of one processor but not more. Calculating the ratio of the workload per processor should therefore always be less or equal to 1. In order to verify this, we can use the distribution of the efficiency values shown in the summary.
- We are performing a regression on the data to calculate the coefficients and therefore we determine the residuals for the fitted values. The distribution of the residuals is also given as part of the summary.
- The coefficients σ and κ are the result that we are essentially interested in. They tell us the magnitude of the contention and coherency effects within the system.
- Finally R^2 estimates how well the model fits the data. We can see that the model is able to explain more than 98 percent of the data.

The function *efficiency()* extracts the efficiency values from the model and allows us to have a closer look at the specific efficiencies of the different processor configurations.

```
R> efficiency(usl.model)


     1      4      8     12     16     20     24     28     32     48     64 
1.0000 0.9750 0.8125 0.7083 0.5938 0.5000 0.4375 0.4107 0.4062 0.2917 0.2422 
```

A bar plot is useful to visually compare the decreasing efficiencies for the configurations with an increasing number of processors. Figure 2 shows the output diagram.

```
R> barplot(efficiency(usl.model), ylab = "efficiency / processor", xlab = "processors")
```

The efficiency can be used for a first validation and sanity check of the measured values. Values larger than 1.0 usually need a closer investigation. It is also suspicious if the efficiency gets bigger when the load increases.

The model coefficients σ and κ can be retrieved with the *coef()* function.

```
R> coef(usl.model)


      sigma       kappa 
0.050023944 0.000004708 
```

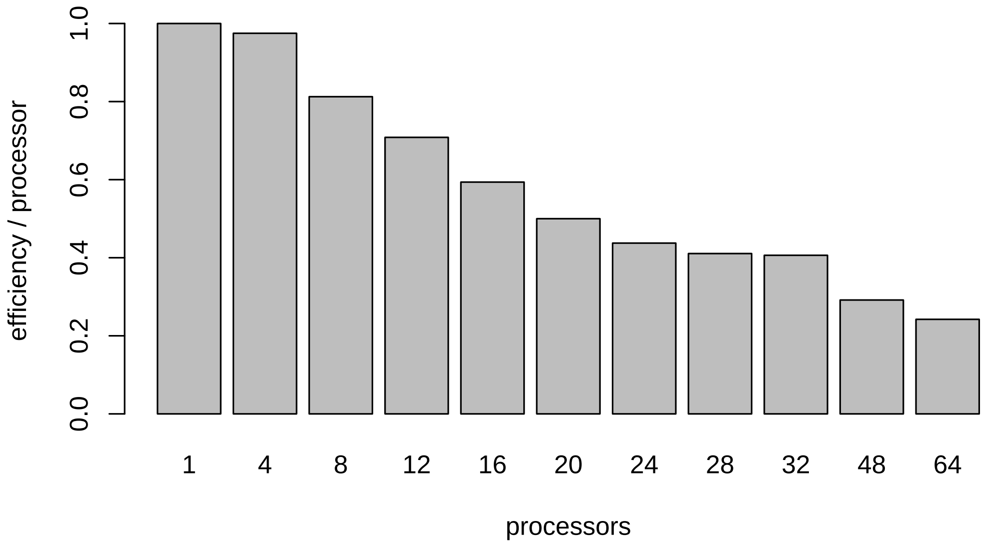

Figure 2: Rate of efficiency per processor for different numbers of processors running the raytracing software

The corresponding confidence intervals for the model coefficients are returned by calling the *confint()* function.

```
R> confint(usl.model, level = 0.95)


           2.5 %    97.5 %
sigma  0.0442072 0.0558407
kappa -0.0001208 0.0001302
```

Earlier releases of the usl package used bootstrapping to estmate the confidence intervals. This has been changed since bootstrapping with a small sample size may not give the desired accuracy. Currently the confidence intervals are calculated from the standard errors of the parameters.

To get an impression of the scalability function we can use the *plot()* function and create a combined graph with the original data as dots and the calculated scalability function as a solid line. Figure 3 has the result of that plot.

```
R> plot(throughput ~ processors, data = raytracer, pch = 16)
R> plot(usl.model, add = TRUE)
```

SGI marketed the Origin 2000 with up to 128 processors. Let's assume that going from 64 to 128 processors does not introduce any additional limitations to the system architecture. Then we can use the existing model and forecast the system throughput for other numbers like 96 and 128 processors using the *predict()* function.

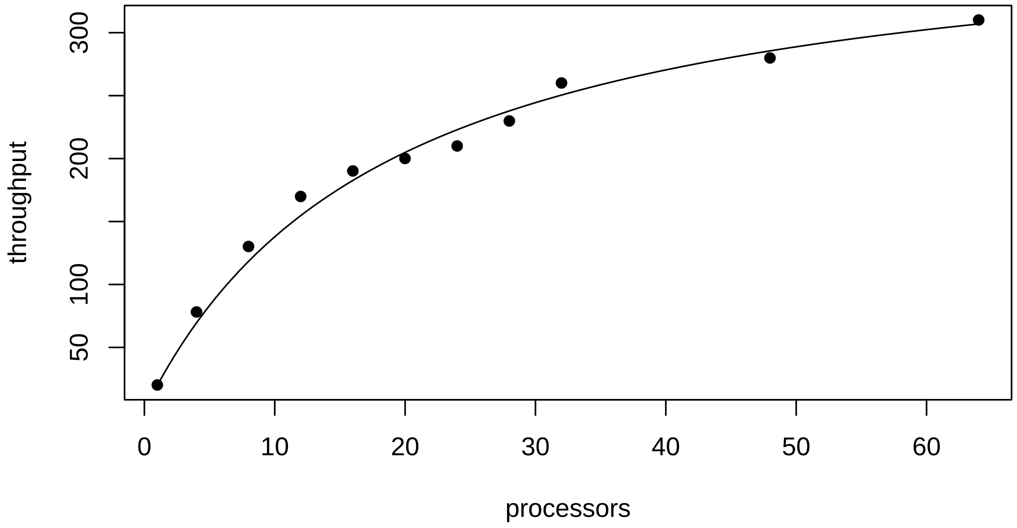

Figure 3: Throughput of a raytracing software using different numbers of processors

```
R> predict(usl.model, data.frame(processors = c(96, 128)))


    1     2
331.3 344.6
```

We can see from the prediction that there is still an increase in throughput achievable with that number of processors. So we use the *peak.scalability()* function now to determine the point where the maximum throughput is reached.

```
R> peak.scalability(usl.model)


[1] 449.2
```

According to the model, the system would achieve its highest throughput with 449 processors. This is certainly a result that could not easily be deduced from the original dataset.

4.2 Case Study: Software Scalability

In this section we will perform an analysis of a SPEC benchmark. A Sun SPARCcenter 2000 with 16 CPUs was used in October 1994 for the SDM91 benchmark[2]. The benchmark simulates a number of users working on a UNIX server (editing files, compiling ...) and measures the number of script executions per hour.

[2] http://www.spec.org/osg/sdm91/results/results.html

First, select the demo dataset with the data from the SPEC SDM91 benchmark.

```
R> library(usl)
R> data(specsdm91)
R> specsdm91


  load throughput
1    1       64.9
2   18      995.9
3   36     1652.4
4   72     1853.2
5  108     1828.9
6  144     1775.0
7  216     1702.2
```

The data provides the measurements made during the benchmark. The column "load" shows the number of virtual users that were simulated by the benchmark and the column "throughput" has the measured number of script executions per hour for that load.

Next we create the USL model for this dataset by calling the *`usl()`* function. Again we specify a symbolic description of the model and the dataset with the measurements. But this time we choose a different method for the analysis.

```
R> usl.model <- usl(throughput ~ load, specsdm91, method = "nlxb")
```

There are currently three possible values for the `method` parameter:

`default`: The default method uses a transformation into a 2nd degree polynomial. It can only be used if the data set contains a value for the normalization where the "throughput" equals 1 for one measurement. This is the original procedure introduced in chapter 5.2.3 of [Gun07].

`nls`: This method uses the *`nls()`* function of the stats package for a nonlinear regression model. It estimates not only the coefficients σ and κ but also the scale factor for the normalization. The nonlinear regression uses constraints for its parameters which means the "port" algorithm is used internally to solve the model. So all restrictions of the "port" algorithm apply.

`nlxb`: A nonlinear regression model is also used in this case. But instead of the *`nls()`* function it uses the *`nlxb()`* function from the nlmrt package (see [Nas13]). This method also estimates both coefficients and the normalization factor. It is expected to be more robust than the `nls` method.

Keep in mind that if there is no measurement where "load" equals 1 then the default method does not work and a warning message will be printed. In this case the *`usl()`* function will automatically apply the `nlxb` method.

We also use the *`summary()`* function to look at the details for the analysis.

```
R> summary(usl.model)


Call:
usl(formula = throughput ~ load, data = specsdm91, method = "nlxb")

Scale Factor for normalization: 90

Efficiency:
   Min     1Q Median     3Q    Max
0.0876 0.1626 0.2860 0.5624 0.7211

Residuals:
   Min     1Q Median     3Q    Max
 -81.7  -48.3  -25.1   29.5  111.1

Coefficients:
       Estimate   Std. Error
sigma  0.0277295  0.0021826
kappa  0.0001044  0.0000172

Residual standard error: 74.1 on 5 degrees of freedom
Multiple R-squared: 0.99,Adjusted R-squared: 0.987
```

Looking at the coefficients we notice that σ is about 0.028 and κ is about 0.0001. The parameter σ indicates that about 2.8 percent of the execution time is strictly serial. Note that this serial fraction is also recognized in Amdahl's Law.

We hypothesize that a proposed change to the system — maybe a redesign of the cache architecture or the elimination of a point to point communication — could reduce κ by half and want to predict how the scalability of the system would change.

We can calculate the point of maximum scalability for the current system and for the hypothetical system with the *peak.scalability()* function.

```
R> peak.scalability(usl.model)

[1] 96.52

R> peak.scalability(usl.model, kappa = 0.00005)

[1] 139.4
```

The function accepts the optional arguments `sigma` and `kappa`. They are useful to do a what-if analysis. Setting these parameters override the calculated model parameters and show how the system would behave with a different contention or coherency coefficient.

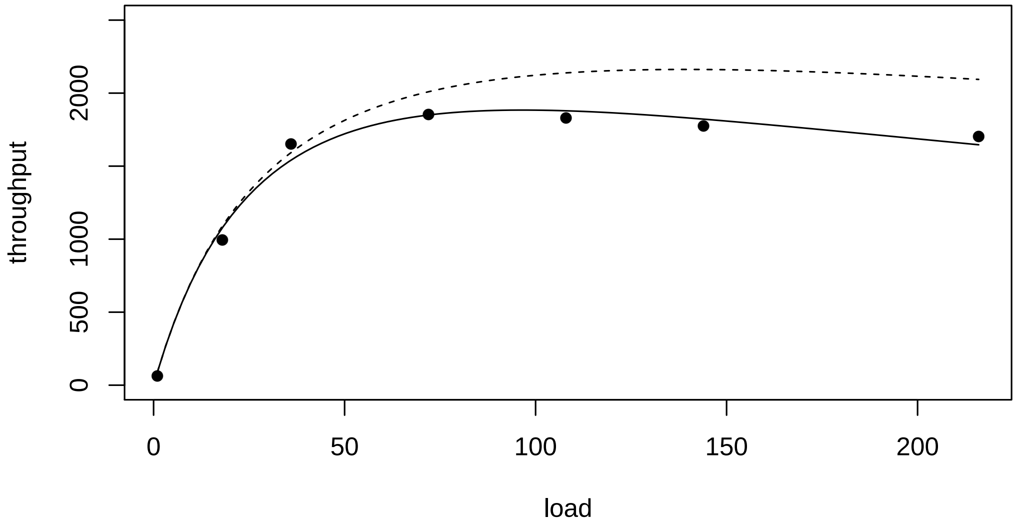

Figure 4: The result of the SPEC SDM91 benchmark for a SPARCcenter 2000 (dots) together with the calculated scalability function (solid line) and a hypothetical scalability function (dashed line)

In this case we learn that the point of peak scalability would move from around 96.5 to about 139 if we would be able to actually build the system with the assumed optimization.

Both calculated scalability functions can be plotted using the `plot()` or `curve()` functions. The following commands create a graph of the original data points and the derived scalability functions. To completely include the scalability of the hypothetical system, we have to increase the range of the plotted values with the first command.

```
R> plot(specsdm91, pch = 16, ylim = c(0,2500))
R> plot(usl.model, add = TRUE)
R> cache.scale <- scalability(usl.model, kappa = 0.00005)
R> curve(cache.scale, lty = 2, add = TRUE)
```

We used the function `scalability()` here. This function is a higher order function returning a function and not just a single value. That makes it possible to use the `curve()` function to plot the values over the specific range.

Figure 4 shows the measured throughput in scripts per hour for a given load, i.e., the number of simulated users. The solid line indicates the derived USL model while the dashed line resembles our hypothetical system using the proposed optimization.

From the figure we can see that the scalability really peaks at one point. Increasing the load beyond that point leads to retrograde behavior, i.e., the throughput decreases again. As we have calculated earlier, the measured system will reach this point sooner than the hypothetical system.

We can combine the `scalability()` and the `peak.scalability()` functions to get the predicted throughput values for the peak values.

```
R> scalability(usl.model)(peak.scalability(usl.model))


[1] 1884


R> scf <- scalability(usl.model, kappa = 0.00005)
R> scf(peak.scalability(usl.model, kappa = 0.00005))


[1] 2162
```

This illustrates how the Universal Scalability Law can help to decide if the system currently is more limited by contention or by coherency issues and also what impact a proposed change would have.

The `predict()` function can also be used to calculate a confidence bands for the scalability function at a specified level. To get a smoother graph it is advisable to predict the values for a higher number of points. Let's start by creating a data frame with the required load values.

```
R> load <- with(specsdm91, expand.grid(load = seq(min(load), max(load))))
```

We use the data frame to determine the fitted values and also the upper and lower confidence bounds at the requested level. The result will be a matrix with column names `fit` for the fitted values, `lwr` for the lower and `upr` for the upper bounds.

```
R> fit <- predict(usl.model, newdata = load, interval = "confidence", level = 0.95)
```

The matrix is used to define the coordinates of a polygon containing the area between the lower and the upper bounds. The polygon connects the points of the lower bounds from lower to higher values and then back using the points of the upper bounds.

```
R> usl.polygon <- matrix(c(load[, 1], rev(load[, 1]), fit[, 'lwr'], rev(fit[, 'upr'])),
+                        nrow = 2 * nrow(load))
```

The plot is composed from multiple single plots. The first plot initializes the canvas and creates the axis. Then the polygon is plotted using a gray area. In the next step the measured values are added as points. Finally a solid line is plotted to indicate the fitted scalability function. See fig. 5 for the entire plot.

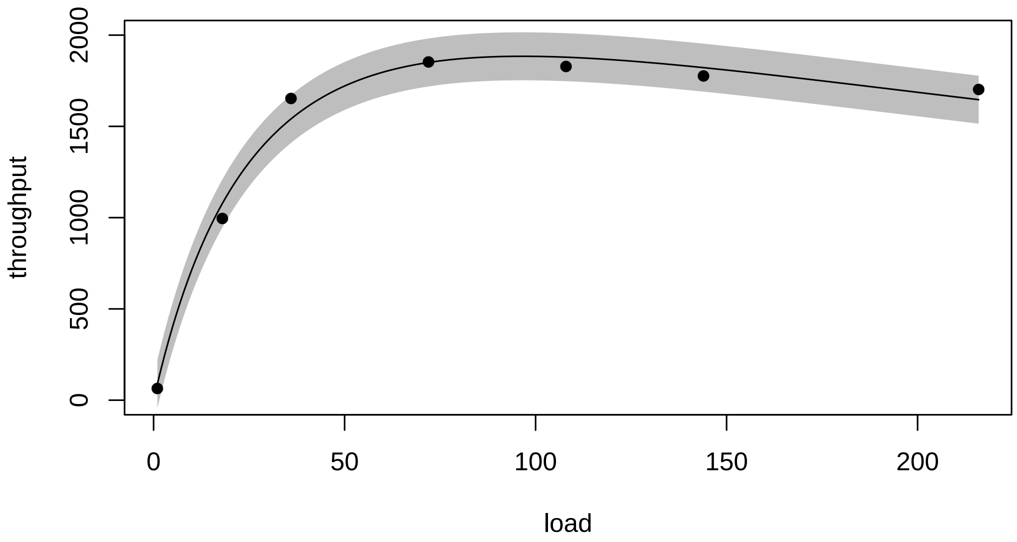

Figure 5: The result of the SPEC SDM91 benchmark with confidence bands for the scalability function at the 95% level

```
R> # Create empty plot (define canvas size, axis, ...)
R> plot(specsdm91, xlab = names(specsdm91)[1], ylab = names(specsdm91)[2],
+       ylim = c(0, 2000), type = "n")
R>
R> # Plot gray polygon indicating the confidence interval
R> polygon(usl.polygon, border = NA, col = "gray")
R>
R> # Plot the measured throughput
R> points(specsdm91, pch = 16)
R>
R> # Plot the fit
R> lines(load[, 1], fit[, 'fit'])
```

Another way to illustrate the impact of the parameters σ and κ on the scalability is by looking at the achievable speedup when a fixed load is parallelized. A naive estimation would be that doubling the degree of parallelization should cut the execution time in halve.

Unfortunately it doesn't work this way. In general there is a range where doubling the parallelization will actually improve the execution time. But the improvement will get smaller and smaller when the degree of parallelism is increased further. This is also an effect of *diminishing returns* as already seen in section 4.1. The real execution time is in fact the sum of the ideal execution time and the overhead for dealing with contention and coherency delays.

Dr. Gunther shows in [Gun08] how the total execution time of a parallelized workload depends on the degree of parallelism p and the coefficients σ and κ of the associated USL model.

Equation 26 in his paper identifies the magnitude of the three components — given as fractions of the serial execution time T_1 — that account for the total execution time of the parallelized workload.

$$T_{ideal} = \frac{1}{p} T_1 \tag{2}$$

$$T_{contention} = \sigma \left(\frac{p-1}{p} \right) T_1 \tag{3}$$

$$T_{coherency} = \kappa \frac{1}{2} (p-1) T_1 \tag{4}$$

The function *overhead()* can be used to calculate the correspondent fractions for a given model. The function has the same interface as the *predict()* function. Calling it with only the model as argument will calculate the overhead for the fitted values. It can also be called with a data frame as second argument. Then the data frame will be used to determine the values for the calculation.

Let's use our current model to calculate the overhead for a load of 10, 20, 100 and 200 simulated users. We create a data frame with the number of users and use the *overhead()* function to estimate the overhead.

```
R> load <- data.frame(load = c(10, 20, 100, 200))
R> ovhd <- overhead(usl.model, newdata = load)
R> ovhd


  ideal contention coherency
1 0.100    0.02496 0.0004696
2 0.050    0.02634 0.0009914
3 0.010    0.02745 0.0051659
4 0.005    0.02759 0.0103840
```

We can see that the ideal execution time for running 10 jobs in parallel is 1/10 of the execution time of running the jobs unparallelized. To get the total fraction we have to add the overhead for contention (2.5%) and for coherency delays (0.047%). This gives a total of 12.54%. So with 10 jobs in parallel we are only about 8 times faster than running the same workload in a serial way.

Equation (3) shows that the percentage of time spent on dealing with contention will converge to the value of σ. Equation (4) explains that coherency delays will grow beyond any limit if the degree of parallelism is large enough. This corresponds to the observation that adding more parallelism will sometimes make performance worse.

A stacked barplot can be used to visualize how the different effects change with an increasing degree of parallelism. Note that the result matrix must be transposed to match the format needed for for the *barplot()* command.

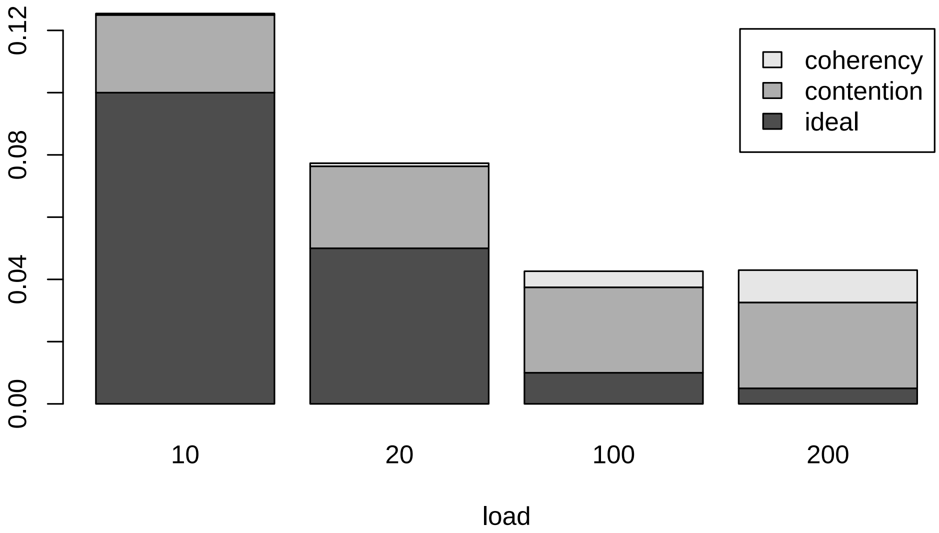

Figure 6: Decomposition of the execution time for parallelized workloads of the SPECSDM91 benchmark. The time is measured as a fraction of the time needed for serial execution of the workload.

```
R> barplot(height = t(ovhd), names.arg = load[, 1], xlab = names(load), legend.text = TRUE)
```

Figure 6 shows the resulting plot. It clearly shows the decrease in ideal execution time when the degree of parallelism is increased. It also shows how initially almost only contention contributes to the total execution time. For higher degrees of parallelism the impact of coherency delays grows. Note how the difference in ideal execution time between 100 and 200 parallel jobs effectively has no effect on the total execution time.

4.3 Case Study: Multivalued Data

It is very common to use multivalued data for a scalability analysis. These measurements are often taken from a live system and may include many different data points for similar load values. This could be the result of a non-homogeneous workload and an analyst has to decide how to take that into account. But for a production system there is usually no feasible way to create a homogeneous workload.

The following data shows a subset of performance data gathered from an Oracle database system providing a login service for multiple web applications. For the analysis we focus on only two of the available metrics:

`txn_rate`: The average number of processed database transactions. This metric is given as transactions per second.

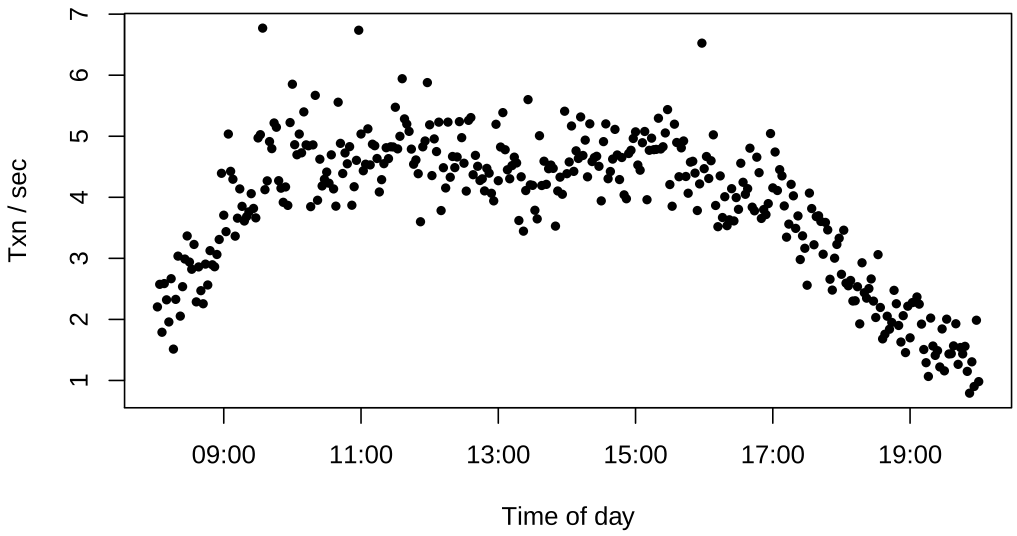

Figure 7: Transaction rates of an Oracle database system during the day of January 19th, 2012

`db_time`: The average time spent inside the database either working on a CPU or waiting for resources (I/O, locks, buffers …). The time is expressed as seconds per second, so two sessions working for exactly one quarter of a second each will contribute a total of half a second for that second. Oracle has coined the term *Average Active Sessions* (AAS) for this metric.

Let's have a look at the first couple of data points in our data set. For each time interval of two minutes there is a corresponding value for the average database time per seconds and for the average number of transactions per second in this interval.

```
R> data(oracledb)
R> head(subset(oracledb, select = c(timestamp, db_time, txn_rate)))


            timestamp db_time txn_rate
1 2012-01-19 08:02:00  0.3120    2.205
2 2012-01-19 08:04:00  0.3224    2.574
3 2012-01-19 08:06:00  0.1918    1.790
4 2012-01-19 08:08:00  0.3136    2.587
5 2012-01-19 08:10:00  0.3584    2.321
6 2012-01-19 08:12:00  0.2354    1.958
```

A naive approch would be a plot of the data as a time series (see fig. 7). This plot shows the familiar pattern of an OLTP application that is mostly used during office hours. Unfortunately this type of plot is pretty much useless when performing a scalability analysis.

The Universal Scalability Law correlates a throughput with a load. In this case the throughput is clearly given by the transaction rate and the database time is a taken as the load metric.

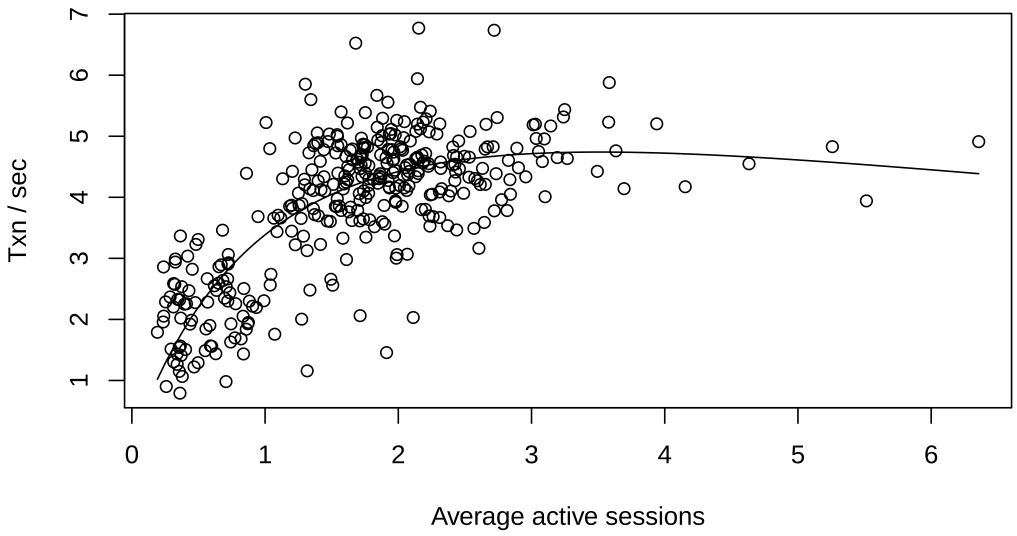

Figure 8: Relationship between the transaction rate and the number of average active sessions in an Oracle database system

The definition above states that the total time spent — either running on a CPU or waiting — is a measurement for the average number of active sessions. So we use that to express the load on the database system.

As usual, we call the *usl()* function to carry out the analysis. See fig. 8 for the scatterplot of the data including the plot of the estimated scalability function.

```
R> usl.oracle <- usl(txn_rate ~ db_time, oracledb, method = "nlxb")
R>
R> plot(txn_rate ~ db_time, oracledb, xlab = "Average active sessions", ylab = "Txn / sec")
R> plot(usl.oracle, add = TRUE)
```

We use the `nlxb` method here because the data does not include a value for exactly one average active session. Current versions of the usl package will automatically switch to the `nlxb` method if the data does not provide the required measurements for the `default` method. So normally you may omit this option.

Now we can retrieve the coefficients for this model.

```
R> coef(usl.oracle)


  sigma   kappa
0.44142 0.04529
```

Our σ here is about an order of magnitude bigger than what we have seen in the previous sections. This indicates a major issue with some kind of serialization or queueing that severely limits the scalability. In fact it is so bad that the impact is already visible with only a few active sessions working at the same time: according to the model the peak throughput is reached at about 3.5 sessions.

```
R> peak.scalability(usl.oracle)


[1] 3.512
```

The confidence interval for σ confirms that there is only a small uncertainty about the magnitude of the calculated coefficients.

```
R> confint(usl.oracle)


        2.5 % 97.5 %
sigma 0.37230 0.5105
kappa 0.01867 0.0719
```

This analysis shows how we can use some of the metrics provided by a live Oracle database system to learn about the scalability. Note that neither the Oracle software nor the application needed any additional instrumentation to collect this data. Also the analysis was done without any internal knowledge about the way the application was using the database.

References

[Gun07] Neil J. Gunther. *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services.* Springer, Heidelberg, Germany, 1st edition, 2007.

[Gun08] Neil J. Gunther. A general theory of computational scalability based on rational functions. *CoRR*, abs/0808.1431, 2008.

[Nas13] John C. Nash. *nlmrt: Functions for nonlinear least squares solutions*, 2013. R package version 2013-9.24.