Generation of realistic benchmark instances with known optima for grid scheduling

Satu Elisa Schaeffer[1,†], Sara Elena Garza Villarreal[1,‡]
1: Postgraduate Division of Computation and Mechatronics (DCM)
School of Mechanical y Electrical Engineering (FIME)
Universidad Autónoma de Nuevo León (UANL)
Av. Universidad, Cd. Universitaria, San Nicolás de los Garza, NL, Mexico
[†]elisa.schaeffer@gmail.com, [‡]saraelena@gmail.com

SUMMARY

We study the statistical properties of real-world grid workload logs and model some of the essential measures as probability distributions for the purpose of artificially generating finite workloads of arbitrary size for arbitrary grid configurations (in terms of node counts and capacities) in such a way that the optimum in terms of total processing time is known for each generated workload. This first version of the proposed generator assumes 100% occupancy, which is something we hope to attend in future work. An implementation of the generator is described and some experimental results are provided, as well as a small example on how to employ the instances in evaluating the performance of a grid-scheduling algorithm.

1 Introduction

In grid scheduling, scalability and performance studies are crucial. One main difficulty when performing these studies consists of generating problem instances that combine objectives: first, having an instance that is highly similar to a real-world grid, and second, having sufficient knowledge over this instance in order to accurately assess algorithm efficiency. To resolve this, workload modeling (i.e., constructing a statistical model after a real workload trace) has been employed. However, knowledge of lower or upper bounds that provide a clear reference to how far from the optimum an algorithm actually is, would be desirable.

In this work, we focus on a *problem instance generation process* that combines *modeling* with the *assignment of a desired optimum* for total processing time, thus permitting the creation of realistic arbitrary-size scheduling problems for a given resource configuration and a known optimum. The aim is to provide greater control over the instance data, without having to sacrifice the resemblance to a real workload.

We first model a workload, then create problem instances with known local optima from this model, and finally experiment with the approach. For the modeling step, we analyze the properties of composed grid log files and propose a generation process that produces a job listing that has a similar structure; to do this, we model the distributions of the requested and actual run time, the CPU time requested, the number of CPUs requested and assigned, and the time between job requests. We then formulate pseudo-random generators to match these distributions.

In order to generate workloads with known local optima, we propose a method based on generation of “blocks” that fill “buckets”. An arrival sequence can be obtained by “emptying” the buckets. Additionally, as a proof of concept, we provide an example of how to use the instances thus generated in evaluating the scalability and performance of scheduling algorithms using a simple heuristic.

The rest of the paper is structured as follows: in Section 2 we provide a background and briefly discuss existing work on the area, followed by grid trace analysis in 3. In Section 4 we describe our proposed generator and provide some experimental results on it, and in Section 5 we conclude the present work and discuss opportunities for future research.

2 Background

The current section introduces grid terminology, overviews synthetic workload generation, and briefly describes related approaches. A grid — according to the definition by Baker [1] —- is "a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed autonomous and heterogeneous resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements." In summary, grids can be compared to federations, where no centralized control exists [9, 15].

Regarding to its components, a grid can be seen in terms of assignable work units, properties, and executing entities. Atomic work units are called *tasks*, and several tasks compose a *job*; a job exhibits several properties or *parameters*, such as arrival time, priority, and number of processors required; according to these properties, the job is assigned or *scheduled* to a different number of resources in order to be executed; these resources may be grouped into an autonomous site or *node* [5].

Because of unique features such as autonomy, heterogeneity, dynamic levels of node availability, and communication costs among nodes, grid scheduling is complex. Hence realistic workload data is required for driving simulations where scheduling algorithms are tested, although this is a difficult endeavor [10]. Two comparable techniques have been used (cf. [18]): the first consists of using *real* production traces, which can be directly collected or searched in repositories like the Parallel Workload Archive [8] and the Grid Workloads Archive [13]; the second consists of generating *synthetic* workloads.

The use of statistics for workload generation is commonly known as *workload modeling*; the goal is to produce a model that resembles — as closely as possible — the original workload. The usual process [3] consists of

1. formulating the model and its parameters,
2. collecting data associated to the formulated parameters,
3. statistically analyzing the data (choosing parameter distributions, fitting data to these distributions, sampling, etc.), and
4. evaluating the model.

Even though modeling is not trivial, it allows for a better understanding and control over the cleaner than the actual trace.

Some approaches propose a specific methodology for developing the model (e.g. the seminal work by Lublin and Feitelson [19] or the study by Downey and Feitelson [6]). Others draw their attention towards certain parameters, such as arrival time [23], request time and job cancellation [4], or task dependency [12]. Works that focus on proposing different means to improve the modeling process can also be found [12, 22]. For a deeper insight on workload modeling, please refer to the works by Calzarossa and Serazzi [3], Feitelson [7], and Li [15, 16, 17].

3 Properties of real-world grid loads

We analyzed the properties of composed grid log files, constructed by Tchernykh and González [12]: one composed of a mixture of seven logs with $175{,}337$ jobs and another with five logs with a total of $512{,}060$ jobs. For each job, the logs include the following data: job number (an identifier), submit time (when the job arrives at queue), wait time (the amount of time the job waits in queue), run time (the amount of calendar time the job is executed), number of allocated processors, average CPU time used (total processor time used), used memory (the amount of memory used by the job), requested number of processors (possibly different from that allocated), requested time (possibly different from the real run time), requested memory (possibly different from that used), status, an identifier of the user who submitted the job, an identifier of the user group from which the job comes, executable (application) number, queue number, partition number, preceding job number, as well as the think time from preceding job.

Our goal is to define a generation process that produces a job listing with a similar distribution in three aspects: intervals between jobs, run time, and the degree of parallelism. The analysis was performed for both log files independently. For purposes of a step towards realistic workload generation, we analyze the following aspects: distribution of requested run time (estimated duration), distribution of the actual run time (actual duration), distribution of the CPU time requested (estimated CPU usage), distribution of the number of CPUs requested (requested degree of parallelism), distribution of the number of CPUs assigned (actual degree of parallelism), and distribution of time between job requests (arrival intervals).

In the analysis, we omitted those jobs that did not have a defined value in the log. Figures 1 and 2 show these distributions in the form of histograms, the former for the smaller log and the latter for the larger log. The distributions show a heavy tail, which indicates that basing any mathematical models on them should be done with *logarithmic binning* to balance the number of samples per discretization step.

The requested and allocated CPU counts behave very similarly to each other, which allows us to model only one of them: we examined the correlations between the requested and assigned quantities. Figure 4 shows the scatter plots for the two correlations, using only one percent of the points to keep the image size reasonable[1]. Two outliers are not shown on the graphic: one with small requested run time and more than $400{,}000$ seconds in actual run time and another

[1]The graphics for the correlation require 14 MB when all points are included for the larger log file alone.

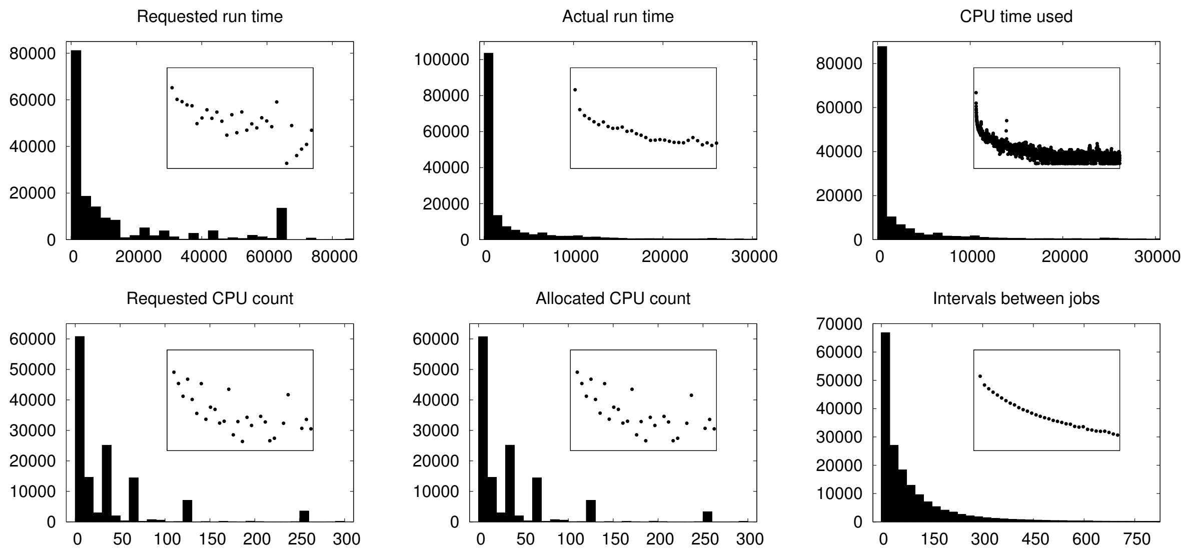

Figure 1: Distributions of five properties in the smaller seven-log combination; the inset the same data on logarithmic scale.

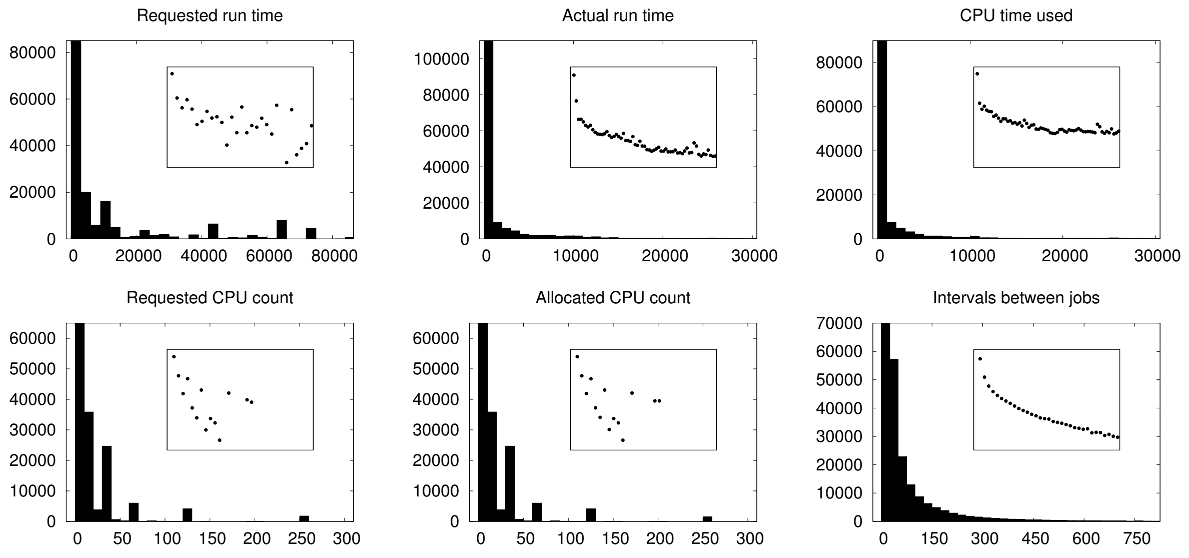

Figure 2: Distributions of five properties in the larger five-log combination; the inset the same data on logarithmic scale.

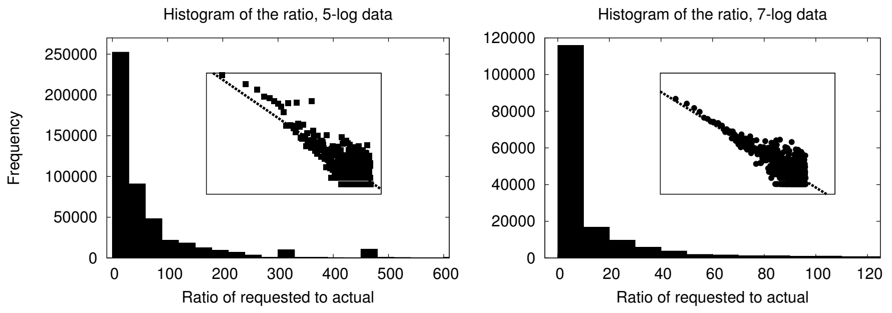

Figure 3: On the left, the five-log data, and on the right, the seven-log data. The upper plots are the histograms of the ratio over the combined log for each case, whereas the lower plot shows the histogram on log-log scale and a curve fitted to it.

with more than $60{,}000{,}000$ seconds in requested but zero actual. These values were, however, kept along for the correlation computation.

The correlations were very high for the CPU counts: 0.995 for the seven-log combination and 0.987 for the five-log combination. This lets us focus on only the assigned CPU count, as the requested number provides no vital additional information. However, the situation is different for the run time: the actual time turns out to be, in general, a value that is distributed between the requested time and zero.

For modeling of the relation between the requested and actual run time, we computed for each job the ratio of requested to actual run time. Figure 3 shows the resulting histograms when using the ratio instead of the value itself, as well as lines fitted on log-log plots of these histograms. We also attempted the fitting of a second-degree polynomial and a cutoff linear fit, but the improvements in this particular case were not significant. The linear fits are given by $\log(y) = -0.74\log(x) + 7.23$ in the five-log case and by $\log(y) = -0.66\log(x) + 5.92$ in the seven-log case. This allows us to model one of these measures (either the requested or the actual run time) in terms of the other, as the ratio between the two is reasonably well-behaved.

In Figures 5 and 6 shows the exponentially-growing step-size histograms obtained by logarithmic binning on base 2 for the seven- and the five-log sets, respectively. In each of the plots, we fitted a second-degree polynomial

$$Y = aX^2 + bX + c \tag{1}$$

of the logarithms (Y, X) of the (x, y) pairs iteratively minimizing the sum-of-squares of residuals in Gnuplot; the resulting fit is drawn in each plot; this yields for the original scale

$$y = 2^c x^{b + a\log_2 x}. \tag{2}$$

Most of these fits are good, although not perfect, but for our purposes the simplicity of the model is an important consideration, as our goal is to artificially reproduce these quantities.

Table 1 shows the values for a, b, and c for the four distributions that we chose to model together with the quality of fit calculated as the R^2 measure for the goodness of fit; for the table we used the R project statistical tools [2]; the fits agree with those produced by Gnuplot

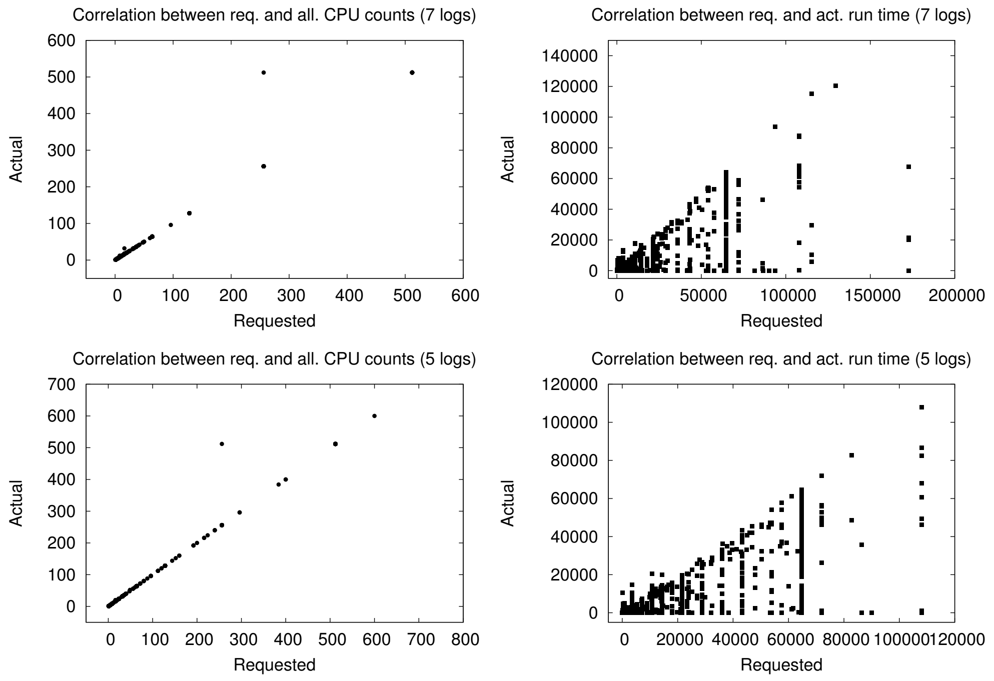

Figure 4: Correlations between the requested and actual quantities for the CPU count (on the left) and the run time (on the right); the upper row shows the values for the smaller seven log combination and the lower row for the larger five log combination.

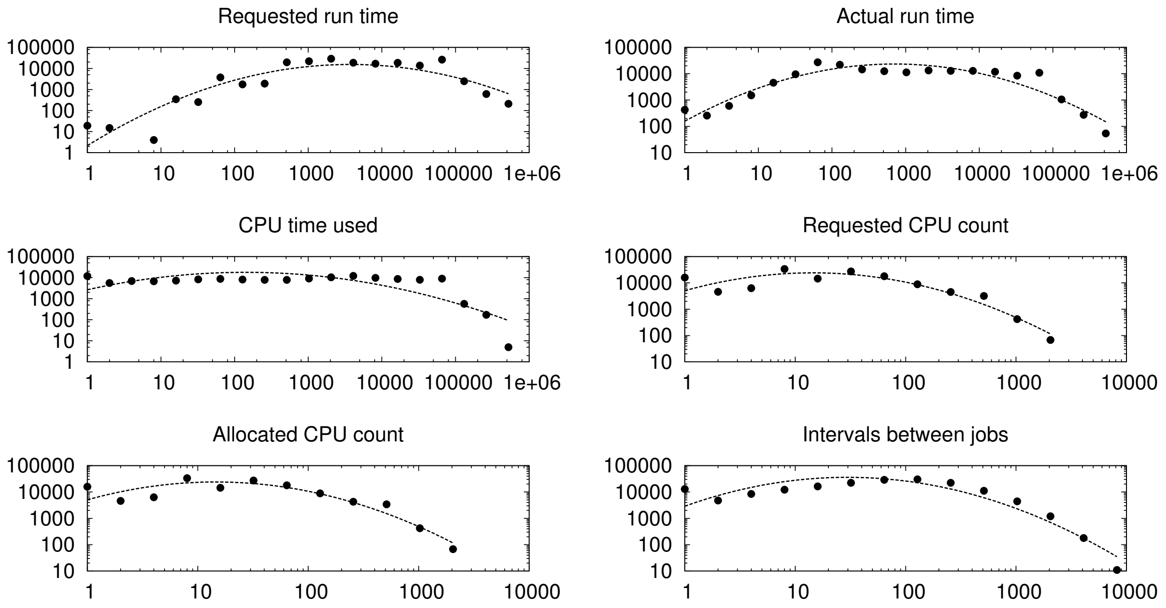

Figure 5: Logarithmic binning and a second-degree polynomial fit for the seven-log data set.

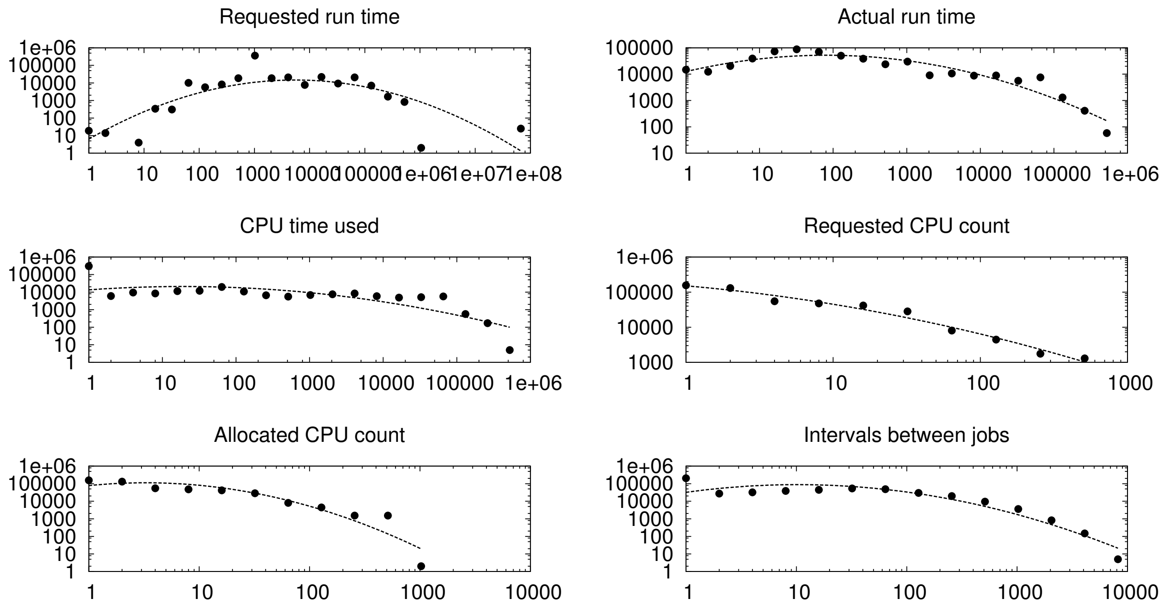

Figure 6: Logarithmic binning and a second-degree polynomial fit for the five-log data set.

[14] in the figures. The requested CPU count will be assumed to coincide with the allocated, due to the high correlation observed, and the actual runtime will be modelled in terms of the requested run time. We also included linear fits in the table, but it is evident from the R^2 statistics that the quadratic fits are superior in quality.

Now, each of these distributions needs to be converted into a pseudo-random number generator. We use the quadratic functions as stand-in functions for our desired probability distribution functions (PDF), integrate each of these functions to obtain a corresponding cumulative distribution function (CDF), cut this curve off at the positive root of the PDF, and the normalize by making this cutoff y-value of the CDF equal to one. We can ignore the integration constant as for a CDF the value at zero is always equal to zero. The roots and the integrals were computed with Wolfram Alpha. The left cutoff, where the final, normalized CDF needs to be equal to one, corresponds to the local maximum of the CDF, located at the computed root of the PDF. The shape of these CDF and their inverse functions is like that shown in the example of in Figure 7. The values involved are given in Table 2; we do not actually need to compute the inverse (which *can* be done, but the function is not very clean and involves quantities with an impractical magnitude for computation) — we will resort to sampling, as explained below.

We first generate uniformly at random in $[0, 1]$ a value Z and compute $Y = 2^Z - 1$ where $Y \in [0, 1]$ but no longer uniformly distributed; this is for translating us back from logarithmic to linear scale. Then, we generate uniformly at random in $[0, \rho]$ a set of R, $|R| = r$, independent trial values x_i; here ρ is the root (cf. Table 2). We compute for each trial using the normalized CDF $ax^3 + bx^2 + c$ the corresponding y_i as

$$y_i = \frac{F(x)}{F(\rho)}, \tag{3}$$

Table 1: The parameters of the fitted second- and first-degree polynomials $y = ax^2 + bx + c$ and $y = \alpha x + \beta$ for the four distributions that need to be modelled for artificial log generation, together with the goodness of each fit. The first rows show the values for the smaller seven-log set, whereas the latter rows contain those of the five-log set.

Quantity	**Set**	a	b	c	R^2_{quad}	α	β	R^2_{lin}
Requested run time	7	-0.09	2.16	1.12	0.82	0.40	6.67	0.34
CPU time used	7	-0.05	0.77	11.4	0.66	-0.26	14.5	0.30
Allocated CPU count	7	-0.15	1.16	12.3	0.88	-0.49	15.1	0.47
Interval between jobs	7	-0.15	1.49	11.5	0.90	-0.49	15.5	0.40
Requested run time	5	-0.07	1.80	2.65	0.64	0.08	9.62	0.01
CPU time used	5	-0.04	0.30	13.7	0.64	-0.37	15.7	0.52
Allocated CPU count	5	-0.18	0.62	16.2	0.87	-1.20	19.0	0.74
Interval between jobs	5	-0.13	0.89	15.0	0.91	-0.81	18.3	0.69

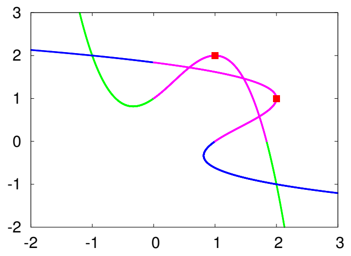

Figure 7: A cubic function $-x^3 + x^2 + x + 1$ (plotted in green) and its inverse (plotted in blue). For generation purposes, we only require the part corresponding to the first quadrant of the cubic, drawn in purple for both the cubic and the inverse curve. The local maximum of the cubic curve and its corresponding point on the inverse are marked with red squares; it is the cutoff point after which the curve is no longer required as the CDF peaks there.

Table 2: The quadratic PDF, its positive root, the cubic CDF and its peak value (at the root) in the log-log domain. All values are rounded, both in representation and in use, as the generation in itself is approximate to begin with. Note the deliberate heavy rounding.

Quantity	**Set**	**Quadratic PDF**	**Root**	**Cubic CDF**	**Peak**
Req. run time	7	$-0.09x^2 + 2.16x + 1.12$	24.51	$-0.03x^3 + 1.08x^2 + 1.12x$	234.53
CPU time used	7	$-0.05x^2 + 0.77x + 11.4$	24.65	$-0.01x^3 + 0.39x^2 + 11.4x$	368.20
All. CPU count	7	$-0.15x^2 + 1.16x + 12.3$	13.71	$-0.05x^3 + 0.58x^2 + 12.3x$	148.80
Int. betw. jobs	7	$-0.15x^2 + 1.49x + 11.5$	15.03	$-0.05x^3 + 0.75x^2 + 11.5x$	172.51
Req. run time	5	$-0.07x^2 + 1.80x + 2.65$	27.11	$-0.02x^3 + 0.90x^2 + 2.65x$	334.81
CPU time used	5	$-0.04x^2 + 0.30x + 13.7$	22.63	$-0.01x^3 + 0.15x^2 + 13.7x$	270.96
All. CPU count	5	$-0.18x^2 + 0.62x + 16.2$	11.36	$-0.06x^3 + 0.31x^2 + 16.2x$	136.08
Int. betw. jobs	5	$-0.13x^2 + 0.98x + 15.0$	15.15	$-0.04x^3 + 0.49x^2 + 15.0x$	200.63

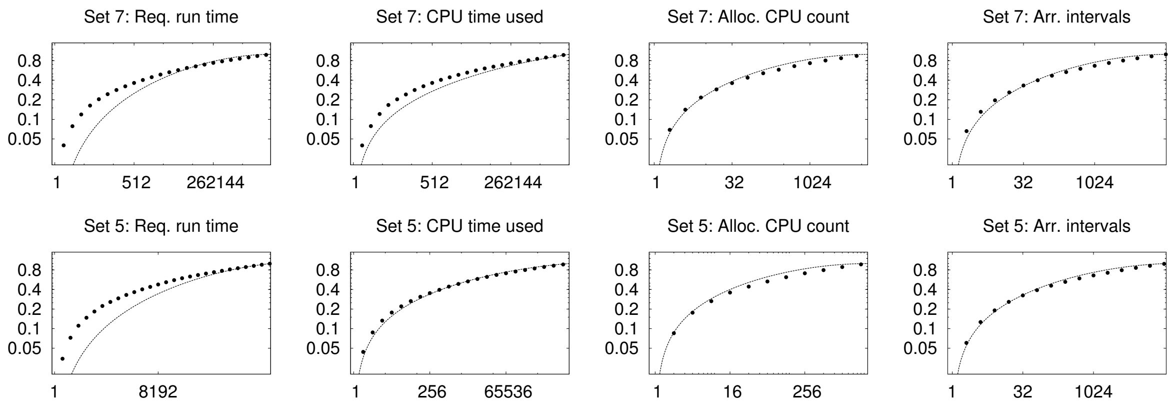

Figure 8: An experimentally obtained CDF (that is, a normalized accumulative histogram over $10,000$ pseudo-random numbers generated as described from the data in Table 2; drawn with dots), together with the with the corresponding analytical CDF, individually for each the eight distributions modeled.

where

$$F(x) = x^{c+b\log_2 x + a(\log_2 x)^2} - 1 \tag{4}$$

is a linear-domain version of the cubic CDF, translated to yield zero at $x = 1$. Note that $y_i \in [0, 1]$ necessarily when $x \in [1, 2^\rho]$ Among these r trials, we choose that x_i that generated the value $f(x_i) = y_i$ nearest to Y:

$$x^* = \arg\min_{x_i \in R}\{|f(x_i) - Y|\}. \tag{5}$$

and use $X = 2^{x^*}$ as the desired linear pseudo-random number. In our experiments, we use $r = 100$ for efficiency (larger values provide higher precision for the model).

An example of the outcome of the described generator is given in Figure 8 that shows log-binned histograms corresponding to numbers that were generated for the modeled distributions of the seven- and five-log data sets. It is evident from the figure that the shape of the distribution is adequately captured, for our purposes, by this simple formulation. It is easy to see that mostly the distribution is captured quite well regardless of the decimals lost in rounding, although the quick-and-dirty inversion sampling together with the heavy rounding does result in moderate over-estimation of the requested runtimes on the lower end. For our purposes, this generation is quite adequate. We leave the improvement of precision in modelling these distributions to further work.

4 Generation of instances with known optima

In order to generate workloads with known local optima, we propose a method based on random generation "blocks" that fill "buckets" in two dimensions. Each block corresponds to a job, the block width being the degree of parallelism, that is, the number of CPUs it occupies, and the block height being the job run time in some arbitrary but fixed time units. Figure 9 shows an example with a single bucket, filled with blocks.

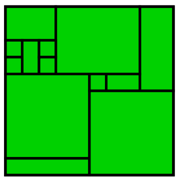

Figure 9: A single bucket with total width of ten units and total height also of ten units. There are several blocks, the smallest being one by one in size and the largest one having width and height both equal to three.

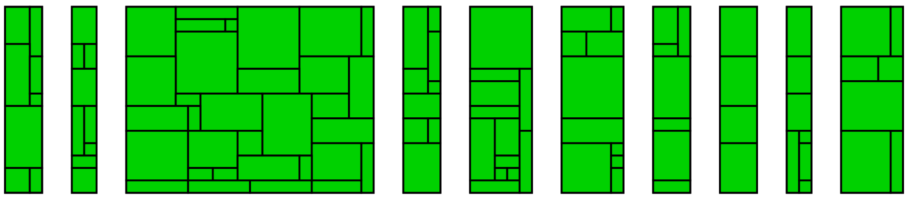

Figure 10: A workload generated with the simplistic Bernoulli filling for a grid that has three nodes with five CPUs, four nodes with three, two with two, and one with 20 CPUs. The optimum was set to 15 time units.

Each node of a grid corresponds to a bucket, with the bucket width being the number of CPUs available in the node. The height of all buckets is given as a parameter and will be the number of time units required by the optimal solution, as the buckets will all be filled with blocks. The number of nodes of each size is also given as a configuration file to the generator. Figure 10 shows an example. For the bucket-block visualizations in the paper we output figures in FIG 3.2 format [21] of XFig and then use the included `fig2dev` tool to convert them to Encapsulated PostScript.

In a simplistic scenario, the blocks can be generated simply by choosing “cells” within the bucket uniformly at random and then ”growing” a job at that position by letting both the job width and the job height expand as a Bernoulli process, thus yielding a geometric distribution independently to both the width and the height of the block. Then, an arrival sequence can be obtained by “emptying” the buckets from the top blocks downwards, only allowing a job to be retracted when there is no other block completely or partially on top of it. This is illustrated in Figure 10.

However, this is all very artificial and there is no guarantee that the resulting workload be somehow reminiscent of real-world scenarios. This is where the models drawn form the log data in the previous section becomes highly useful. We can use the estimated probability distributions to first generate the block sizes and then to guide the order in which a timing sequence is constructed, and also in generating the intervals between job arrivals.

We hence propose the generation of artificial problem instances, that is, job sequences with size requests (defining the CPU count and run time), where each job has a defined arrival time (in terms of time units that pass after the previous job) and where these size and time measures are generated based on the statistics of real job logs, but with the freedom to generate multiple independent instances with an arbitrary number of jobs each. We leave to future work separat-

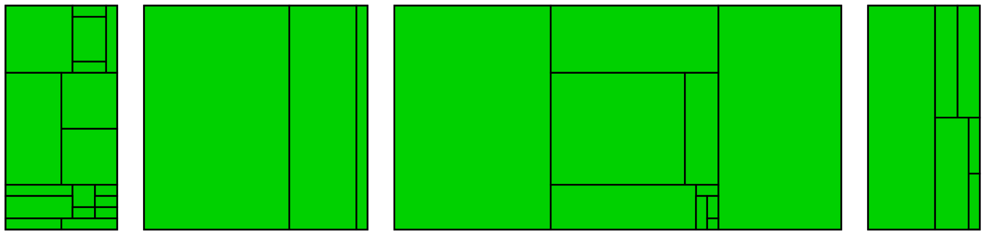

Figure 11: A workload generated with the workload-based filling for a grid that has three nodes with three CPUs: two with 10, one with 20, and one with 40 CPUs. The optimum was set to 20 time units.

ing the requested quantity from the actual one, allowing the algorithm only the requested value at time of job assignment and then revealing the actual value as the job begins execution; the previous sections already report the modelling aspects required to achieve this, but we leave it out for brevity of the presentation in this first work.

4.1 Proposed generator

The generation models takes the following parameters:

1. the model for *job duration* — we model this in terms of the CPU time used,
2. the model for *degree of parallelism* — we model this in terms of the requested CPU count,
3. the model for *inter-arrival time*,
4. the *desired optimum* for the total duration $\mathcal{D}$, and
5. a grid configuration, given as *node sets*. Each node set is represented as a list of pairs (n, d) where n is the number of nodes and d is the number of CPUs per node.

Note that instead of using the CPU time duration, we could use the actual run time. As we know from the results reported in the previous section, the measures of the models involved follow power-law distributions, we now expect more variability in block sizes, as shown in Figure 11.

When generating the degree of parallelism, job duration or arrival delay, we round up to the next integer value for purposes of the boxing model used; for the core count this is obviously always so, whereas in grid computing individual jobs that take less than one second to complete are not efficient anyhow due to the communication overhead in real-world grid systems.

The generation proceeds in the following manner: buckets are created with the widths and multiplicities defined in the configuration file and the height determined by the optimum. Then the buckets are filled completely with jobs that are created one by one using the statistical parameters and are fitted on a greedy first-fit bases onto the buckets. If the distribution suggests a duration greater than the optimum, the job duration is cut down to the exact optimum; similarly, if the degree of parallelism generated exceeds the maximum of the nodes present, it is equally cut down to that maximum value. This is to ensure that all generated jobs could at least hypothetically be included in the scheduling.

The greedy fit attempts from the lowest level of a bucket upwards, seeking a feasible initial position left to right, considering the buckets in random order on each round. When a job does not fit, it is *not* rejected, but instead cut down (choosing uniformly at random whether to reduce the duration or the degree of parallelism to one half of the present value, not letting either one hit zero) and retried until a place is found (cropping again as many times as necessary). The cropping is done in such drastic steps to ensure that the number of retried is logarithmic instead of linear of the maximum block dimension. The bias introduced by this reduction step is discussed in the next section in conjunction with the experiments. This cropping mechanism is rather slow at present; we hope to explore faster accommodation mechanisms in future work.

The arrival-sequence generation in this first version of our proposed generator functions as follows. An arrival time is assigned to each job, using the beginning of its execution in the generated optimal scheduling as a deadline for the job arrival, drawing the intervals between two consecutive jobs from the modelled distribution whenever the deadlines permit doing so. All jobs that initiate their execution at the beginning of the optimal scheduling must arrive at time zero; however, also other jobs may already arrive at time zero as well, although their optimal execution (in terms of the total throughput) initiates at a later time. The selection of which jobs arrive at what time is done by attempting to generate an arrival interval from the modelled distribution, filtering out the jobs that would make it, and then choosing one at random. If none fulfill, the interval is decremented by one unit and tried again.

4.2 Experimental results

We implemented the instance generator in Python 2.7 [20] and generated a set of 30 instances with three different optimums: 50, 150, and 450. All the generated instances share the following grid configuration: 32 nodes with four cores, 16 nodes with eight cores, 8 nodes with 16 cores, four nodes with 32 cores, two nodes with 64 cores, and one node with 128 cores. The beauty in the generation models being power-law shaped is that they are nearly scale-invariant: making smaller or bigger nodes and elevating or decreasing the optimum value does not alter the qualitative statistical properties of the instance. We used the parameters of the seven-log data set as input to the generator.

All of the reported experiments were ran in a single thread on a Mac Mini with an Intel 2.40 GHz Core 2 Duo processor with just 2 GB of RAM non-exclusively (the machine doubles as a web server, among other duties) under OS X 10.7.2 (Lion).

The box-fitting heuristic of this first version of the generator is somewhat slow; the average generation time over the set of 30 instances was a little over five minutes for those with optimum at 50, almost fifty minutes for those with optimum at 150, and almost seven hours for those with optimum at 450. The number of jobs generated per log grew non-linearly with the optimum increase when keeping the same configuration, as expected: the 50-optimum logs had a little more than 400 jobs each, the 150-optimum logs has approximately 460 jobs each,

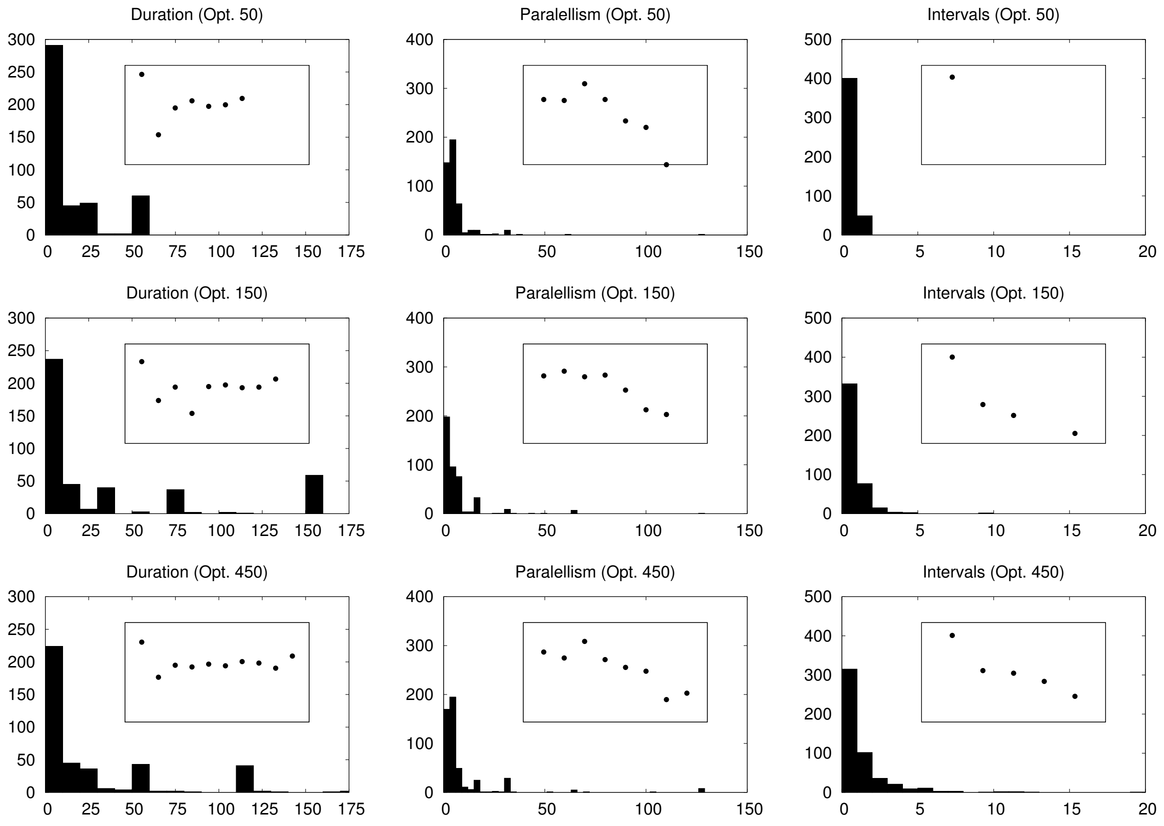

Figure 12: The three distributions — duration (left), parallelism (middle), and arrival intervals (right) — for a 50-optimum log with 449 jobs (top row), a 150-optimum log with 433 jobs (middle row), and a 450-optimum log with 507 jobs (bottom row). All three logs were generated using the parameters modelled from the seven-log real workload and the grid configuration described above.

and the 450-optimum logs nearly 540; higher optimums allow for longer-executing jobs from the runtime distribution.

We computed the distributions of the job duration, job parallelism, and the intervals between jobs in the generated log files. For brevity, we include here those of a single log for each value of optimum used; these are shown in Figure 12. The inset in each figure shows the log-log scale histogram obtained with logarithmic binning, whereas the figure itself is a linear-scale histogram.

For illustrating how these generated instances could be employed, we also implemented in Java [11] a fast first-fit deterministic heuristic based on assigning penalties on empty slots on low levels that reads the generated job listing online, obtaining one job at a time and assigning it to a processor. The heuristic can be ran both in visual and batch mode. We emphasize that the heuristic was *not* meant to compete with state of the art in any way — it is very fast (a couple of seconds at most with visualization included for small instances) and sketched rather than designed and only implemented as a proof of concept of how to use the instances from the proposed generator. Figure 13 shows a screen shot of the heuristic on a small instance.

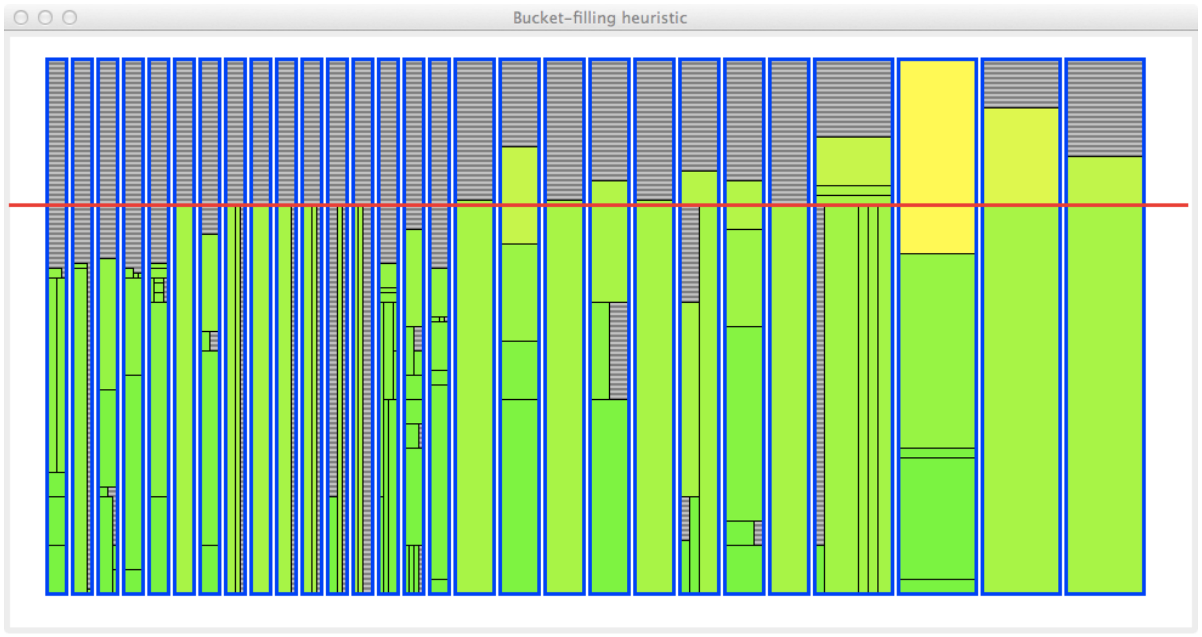

Figure 13: A screen shot of the Java implementation of a simple deterministic heuristic operating on a generated log instance (16 nodes with four cores, eight with eight, and four with 16). The red bar indicates the optimum, the value of which is included in the generated log file. In this case, the optimum was 80 time units and the heuristic solution requires 110 time units, corresponding to 1.375 times the optimum.

5 Conclusions and further work

We have proposed a generator of instances for grid scheduling where the sizes of the jobs, both in terms of duration and CPU count, are modeled on real-world data, and where the optimum is known. We base the parameters of the generator in careful statistical analysis and curve-fitting. We provide experimental results on both the model and the function of the proposed generator, as well as an anecdotal proof-of-concept heuristic to illustrate the use of the proposed generator.

The present version of the proposed generator assumes a non-realistic 100% occupancy rate of the grid cores, which we plan to relax in future work. Also, it is of interest to apply data clustering techniques to the real-world instances before the statistical analysis in order to identify user groups, each with a distinct internal behavior, and then model the job flow from each user group independently instead of treating the job generation as a single random process.

Also dependencies between past and future jobs are of interest for further realism in the generated workloads; presently each job is generated independently from the previous ones, with the exception of the optional filler jobs that are inserted to guarantee an exact optimum, whereas the real workloads new jobs are often follow-up tasks for previous jobs and their arrival time and size may depend on the preceding jobs.

REFERENCES

[1] M. Baker, R. Buyya, and D. Laforenza. “Grids and grid technologies for wide-area distributed computing”. *Software: Practice and Experience*, Vol. 32 No. 15 pp. 1437–1466, (2002).

[2] D. Bates, et al. “The R project for statistical computing”, (2011). `http://www.r-project.org`.

[3] M. Calzarossa and G. Serazzi. “Workload characterization: A survey”. *Proceedings of the IEEE*, Vol. 81 No. 8 pp. 1136–1150, (1993).

[4] W. Cirne and F. Berman. “A comprehensive model of the supercomputer workload”. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 140–148. IEEE, (2001).

[5] F. Dong and S. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical report, School of Computing, Queen’s University, Kingston, Ontario, (2006).

[6] A. Downey and D. Feitelson. The elusive goal of workload characterization. *ACM SIGMETRICS Performance Evaluation Review*, Vol. 26 No. 4 pp. 14–29, (1999).

[7] D. Feitelson. Workload modeling for performance evaluation. In M. Calzarossa and S. Tucci, editors, *Performance Evaluation of Complex Systems: Techniques and Tools*, Vol. 2459 of *Lecture Notes in Computer Science*, pp. 114–141. Springer, (2002).

[8] D. Feitelson, administrator. Parallel Workloads Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/`, accessed (2012).

[9] I. Foster. What is the grid? a three point checklist. *GRID today*, Vol. 1 No. 6, (2002).

[10] G. R. Ganger. “Generating representative synthetic workloads: An unsolved problem”. In *Proceedings of the Computer Measurement Group (CMG) Conference*, pp. 1263–1269, (1995).

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, third edition, (2005). `java.sun.com/docs/books/jls`.

[12] A. Hirales Carbajal, J. L. González García, and A. Tchernykh. “Workload generation for trace based grid simulations”. In *Proceedings of ISUM 2010 First International Supercomputing Conference in Mexico*, Guadalajara, Jalisco, Mexico, (2010).

[13] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. Epema. “The grid workloads archive”. *Future Generation Computer Systems*, 24(7) pp. 672–686, (2008).

[14] P. K. Janert. “*Gnuplot in Action — Understanding Data with Graphs*”. Manning Publications, (2009). `http://www.gnuplot.info`.

[15] H. Li. “*Workload Characterization, Modeling, and Prediction in Grid Computing*”. PhD Thesis, Universiteit Leiden, (2008).

[16] H. Li. “Workload dynamics on clusters and grids”. *The Journal of Supercomputing*, 47(1) pp. 1–20, (2009).

[17] H. Li and R. Buyya. “Model-driven simulation of grid scheduling strategies”. In *e-Science and Grid Computing, IEEE International Conference on*, pp. 287–294. IEEE, (2007).

[18] V. Lo, J. Mache, and K. Windisch. “A comparative study of real workload traces and synthetic workload models for parallel job scheduling”. In D. Feitelson and L. Rudolph, eds., *Job Scheduling Strategies for Parallel Processing*, Vol. 1459 of *Lecture Notes in Computer Science*, pp. 25–46. Springer, (1998). 10.1007/BFb0053979.

[19] U. Lublin and D. Feitelson. “The workload on parallel supercomputers: modeling the characteristics of rigid jobs”. *Journal of Parallel and Distributed Computing*, 63(11) pp. 1105–1122, (2003).

[20] Python Software Foundation. “Python v2.7.2 documentation”, (2012). `http://docs.python.org`.

[21] T. Sato and B. V. Smith. “Fig format 3.2”. In *XFig User Manual, Version 3.2.5*. (2007). `http://epb.lbl.gov/xfig/fig-format.html`.

[22] B. Song, C. Ernemann, and R. Yahyapour. “Parallel computer workload modeling with Markov chains”. In *Job Scheduling Strategies for Parallel Processing*, pp. 9–13. Springer, (2005).

[23] M. Squillante, D. Yao, and L. Zhang. “The impact of job arrival patterns on parallel scheduling”. *ACM SIGMETRICS Performance Evaluation Review*, Vol. 26 No. 4 pp. 52–59, (1999).