# MULTIPROGRAMMED MEMORY MANAGEMENT FOR RANDOM-SIZED PROGRAMS*

Bernhard Walke
AEG-TELEFUNKEN, Research Institute
D-7900 Ulm, W-Germany

## Abstract

We consider a probabilistic model of a computer system with multiprogramming and paging. The applied work-load is derived from measurements in scientific computer applications and is characterized by a great variance of compute time. Throughput of a cyclic model is computed approximately presuming program sizes with negative exponential distribution. After a review of previous results for a memory allocation policy with prescribed number, n, of working sets at least to be loaded, an adaptive memory allocation policy is introduced which dynamically changes the number, n. Thereby, it is possible to reach the goal of having always enough memory available to load the parachor of each program. Simulation results establish our approximations as being very good. CPU scheduling is chosen to be throughput optimal. Our results are useful to demonstrate the benefits of allocation policies with adaptive controlled degree of multiprogramming. Previous contributions to this problem are to that date only by means of simulation [5].

## 1. Introduction

Throughput of a computer can be enhanced by increasing the degree of multiprogramming, which results in better parallel work of central resources (CPU, channels). But generally, this is only true if the main memory is also enlarged adequately. Each of the multiprogrammed jobs must be given enough space to keep its working set there, otherwise thrashing [2] would occur and throughput would vanish.

In this paper, we introduce a policy for controlling the degree of multiprogramming in a paged memory with respect to maximum throughput. This policy, ACM-n (<u>a</u>daptive <u>c</u>ontrol of the <u>m</u>ultiprogramming degree of a number, not greater <u>n</u>, of partly loaded programs) is a working set policy in that it aims at keeping the parachor [3] of each program in the main memory and dynamically changes the degree

*) This work was partly supported by the 2nd EDP-Program of the Federal Republic of Germany.

of multiprogramming to reach that goal.

A queueing network model of a computer system is developed and throughput for a given workload is computed for the ACM-n algorithm. Program size is assumed to be negative exponentially distributed. From this assumption follows that for a memory of given size, the number of fully (or partly) loadable programs changes over time, e.g. sometimes only one working set could be loaded (when a program happens to be very large), at other times many working sets together could be kept there. We use a new technique (of the decomposition type [1]) to approximately compute the steady-state utilization of the CPU and from this the throughput.
A special case of the ACM-n policy which is called MPM-n policy (minimal prescribed multiprogramming degree, n, is fixed) was already dealt with in [7]. Under this policy a number, n, or more programs are multiprogrammed whenever they happen to fit completely together in the main memory, otherwise n uncompletely loaded programs must be multiprogrammed. The results are reviewed in chapter 4 of this paper.

The subject of this paper is the case of a dynamically changing, ACM-n controlled, multiprogramming degree of partly loaded programs. For reasons of simplification of computation we restrict our model to the case where this degree, n, is limited, $n \leq 2$. More than two programs are admitted whenever they happen to be small enough to be completely loaded into the main memory. Simulation results show that our computational results are acceptable.

## 2. Model of the workload

We assume all programs being of the same type, there are no different classes of programs. Program size is assumed to be given by an independent and identically distributed random variable, G, with probability distribution function (p.d.f.)

$$P(G \leq g) = 1 - \exp(-g/E_G) \qquad (2.1)$$

with mean $E_G$. (Big letters without index denote random variables throughout this paper, the corresponding small letter denotes an assumed value of such a variable). The work for a program loaded completely into the main memory consists exactly of two service intervals: one transport from the background to the main memory and one compute interval, which may be preempted but could be serviced uninterruptedly. The term 'program' in this context is chosen for such parts of service for jobs, for which the given description is appropriate (e.g. compilation, execution, etc.). The transport back from main memory to the secondary storage is not modeled explicitly. It should be thought of as included in the loading transport of the program.

If the main memory, available for a distinct program, is limited so that program size is greater than memory space, only a part of the program (which is usually called the working set) is loaded. In this situation a compute interval is limited by two I/O-demands, one of which in front of the compute interval the other at the end. For example page exception, segment call, or buffer overflow are reasons to limit a compute interval (provided they could not be serviced in parallel to CPU service for the same program). We call the sequence of I/O-demand and compute interval a service interval of a program. Fig. 1 illustrates this model for the execution of a program: An alternating sequency of I/O-demand, T (transport), and CPU service, C (compute).

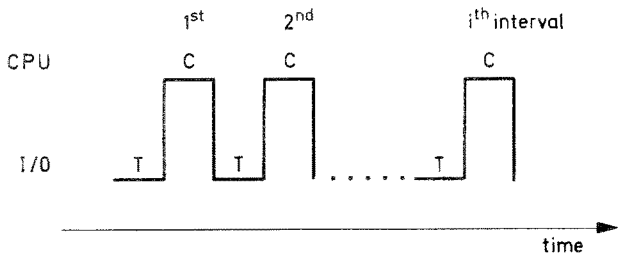

Fig. 1: Model of the progress of service for a not completely loaded program (no concurrent programs assumed).

For our queueing model we assume the variable, T, to be an independent random variable with p.d.f.

$$P(T \leq t) = 1 - \exp(-t/E_T) \qquad (2.2)$$

and mean $E_T$.

Compute intervals, C, are defined by a degenerate exponential d.f. [6], fig. 2,

$$P(C \leq t) = 1 - (1-p) \exp(-t/E_{C'}) \qquad (2.3)$$

with mean

$$E_C = (1-p)\, E_{C'} \qquad (2.4)$$

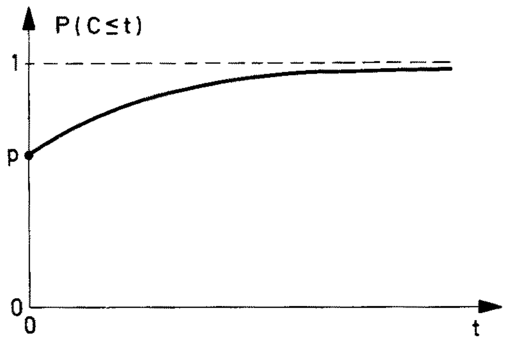

Fig. 2: Degenerate exponential distribution function.

This distribution is a special case of the well-known two-phase hyperexponential d.f. and is much more suitable to approximate measured cumulative d.f.'s of compute intervals, C, than the exponential function (wich is a limiting case of eq. (2.3) for $p = 0$). 'Short' compute intervals are approximated by compute intervals of the length zero, which appear in eq. (2.3) with probability p. Non-zero compute intervals are approximated by a negative exponential d.f. with mean $E_{C'}$. Together with the LCFS-P (last come first serve preemptive) CPU-scheduling algorithm [4] this d.f. can be handled computationally in the same way as a simple negative exponential d.f. [6].

We have chosen these two simple d.f.'s (eqs. 2.2, 2.3) for reasons of mathematical tractability. As will be seen later, refinements at these points seem to be of subordinate influence on throughput compared with the influence of the parachor curve of programs.

Next we assume that the probability for each service interval being the last one of a program is constant. From this follows that the number, I, of intervals per program is given from a geometric distribution. The mean, m, depends on the portion, x, of the whole program which is brought to the main memory. For fully loaded programs, $x = 1$, there results exactly one service interval per program and therefore m equals one. For very uncompletely loaded programs, $x \longrightarrow 0$, the mean number of intervals approximates infinity, $m \longrightarrow \infty$, fig. 3. The function m(x) has been approximated in the literature in numerous ways, e.g. [2], and was sometimes called parachor curve.

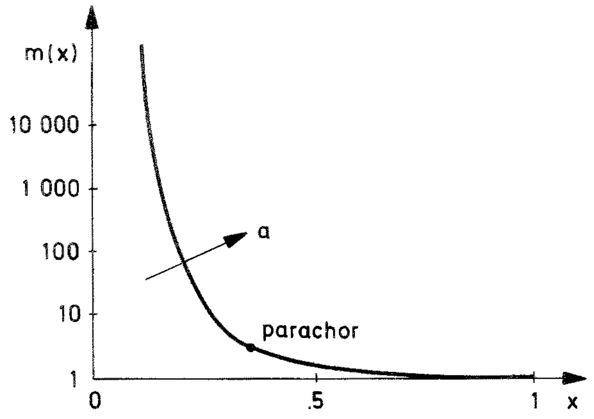

Fig. 3: Mean number of service intervals, m, dependent on the portion, x, of a program loaded to main memory (parachor curve).

For a demand paging system, for instance, the expression (m(x)-1) may be the number of page faults during program execution. We choose a simple function to describe the dependency of m(x) on x

$$m(x) = x^{-a} \qquad (2.5)$$

where the variable, a, is a parameter. Measured parachor curves can be approximated [7] by eq. (2.5) with parameter values $a \geq 2$. The reciprocal, 1/m(x), is sometimes called life-time function.

Next we assume that the CPU-time to execute a program is subdivided in compute intervals whenever the program is loaded uncompletely, but the whole mean compute time per program, $E_{Cp}$, remains unchanged (no CPU-overhead is involved). For I/O-time, however, the analogue must not be true.

Instead of this we introduce for our computations a function

$$E_T = g(E_{Tp},x) \qquad (2.6)$$

where $E_{Tp}$ is the mean transport time of completely loaded programs. Remember that by our definitions the expected values of compute intervals, $E_C$, and I/O-demands, $E_T$, are functions of x. Our workload description is chosen so that

$$E_{Cp} = m(x)\ E_C(x) \qquad (2.7)$$

## 3. Model of the computer systems

The computer is modeled by a cyclic queueing network with two servers in tandem and n circulating programs, fig. 4. The queue discipline is FCFS for the channel-queue (on the secondary storage) and LCFS-P for the CPU-queue (in main memory).

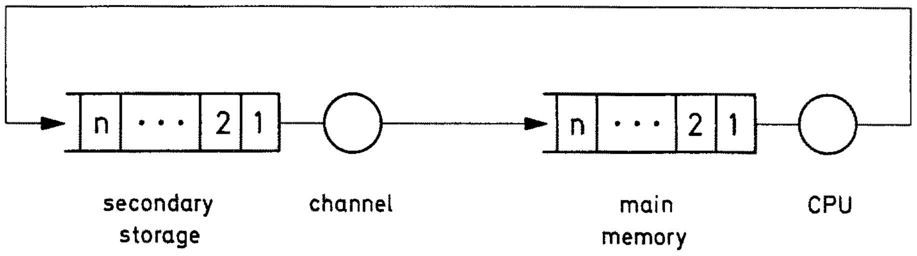

Fig. 4: Cyclic queueing network model of a computer system.

If the number, n, of programs is constant, which is equivalent to a fixed degree of multiprogramming and a fixed program size, CPU-utilization, U, is known from [6] to be

$$U = D\cdot E_{Cp} = \frac{\rho^{n} - 1}{\rho^{n+1} - 1}, \qquad \rho = \begin{cases} g(E_{Tp},x).m(x)/E_{Cp} & (x<1) \\ E_{Tp}/E_{Cp} & (x=1) \end{cases} \qquad (3.1)$$

where the variable, D, is the program throughput. Mean turn-around time, $E_B$, is computed from throughput, D, by

$$E_B = 1/D \qquad (3.2)$$

When program size follows the p.d.f. given by eq. (2.1) and the memory size, s, is limited, the number of completely loadable programs becomes a random variable. The probability, $p_n$, for "n[and not (n+1)] independent programs completely fit in the main memory" is computable [7] to be Poisson distributed

$$p_n = \frac{\gamma^n}{n!} \exp(-\gamma), \qquad \gamma = s/E_G \qquad (3.3)$$

Independent of memory size there is a certain probability, $p_o$, that no complete program can be loaded. If we demand a given degree, $n > 1$, of multiprogramming being upheld, the probability for "n complete programs are not loadable" increases over $p_o$. In what follows we decide to load programs only in such a manner that all of them have the same portion, x, of their full size loaded. Then the portion, X, which is now a random variable can be computed from

$$X = \begin{cases} s/\sum_{i=1}^{n} G_i & \text{for } s < \left(\sum_{i=1}^{n} G_i\right) = G_n \\ 1 & \text{otherwise} \end{cases} \qquad (3.4)$$

where the variables, $G_i$, are distinct random variables with p.d.f. given by eq. (2.1). Our mean values, $E_C(x)$, $E_T(x)$, $m(x)$, are for each program dependent on the actual value, x, of the random variable X. All the random variables, T, C, G, I, are assumed to be independent of each other.

Now we are interested in the p.d.f. of the size of a number, n, of programs which fit together each with a portion, x, in a memory of given size, s. This can be computed to be (cf. [7])

$$P(xG_n < s \mid G_n \geq s) = 1 - \sum_{j=0}^{n-1} \frac{(\gamma/x^2)^j}{j!} \left( \sum_{j=0}^{n-1} \frac{\gamma^j}{j!} \right)^{-1} \exp(-\gamma(1/x-1)) \qquad (3.5)$$

$$(x \leq 1)$$

x, $G_n$ are defined from eq. (3.4). If we now prescribe the degree of multiprogramming, n, which must be upheld, then we have from eq. (3.3) the probability, $p_k(k \geq n)$, of multiprogramming a number, k, of completely loaded programs and from eq. (3.5) the p.d.f. of the portion X, of partly loaded programs under a multiprogramming degree, n.

## 4. Approximate throughput computation by decomposition methods

Decomposition methods are used to compute steady-state values of complex systems with a large number of state variables from subsystems with small groups of variables. From [1] we know that this technique still yields good approximations when interactions among subsystems do exist but are weak compared to the interactions within subsystems. Such systems are called nearly completely decomposable and have the property that short run dynamics can be distinguished from long run dynamics.

In our simple queueing network, fig. 4, a great complexity in time is involved by the assumption that program size is a random variable and from this the degree of multiprogramming, n, becomes a random variable, N. This causes the portion, x, of uncompletely loaded programs (which appear whenever less than a prescribed number, n, of programs are loadable) also to be a random variable, X, eq. (3.4).

We interpret a time interval in which a number, n, of programs is allocatable to main memory (fully, or each with a portion, x) as a subsystem with short run dynamics and describe it by a network given by fig. 4. Whenever a program has completed all its service intervals, the next randomly chosen program with random size, G, is generated and the portion, x, or the degree, n, of multiprogramming is altered, if programs had not been or had completely been loaded, respectively. Such changes can be interpreted as long run dynamics.

Steady-state CPU utilization of the (in time) complex network, fig. 4, is aggregated from the steady-state solutions for the subsystems weighted with the probabilities of appearance of these subsystems. Subsystems each are not time dependent but differ in the portion, x, or the degree, n, of partly or completely loaded programs, respectively. The whole system utilization is only approximately computable because it is not completely decomposable. The decomposition technique for our problem, which is a decomposition in time, has been developed independently of the work reported on in [1] and, by simulation, has proven to yield good results.

Now shortly we review the results for the policy MPM-n from [7]. From eq. (3.1, 3.2) we have the mean turn-around time, $E_B$, for n completely, (i.e. $x = 1$) and also for n not completely ($x < 1$) loaded programs assuming constant program size. In the case of a random program size and a number, n, or more completely loaded programs, one fraction of the whole turn-around time, $E_{Bw}$, is computed by the weighted composition of subsystems with exactly n programs, the weighting factor being the probability $p_n$, (eq. 3.3). The other fraction of $E_{Bw}$ results from all subsystems with exactly n partly loaded programs, each subsystem with another portion, x. The probability density function for a number, n, of programs being partly loaded with exactly the portion, x, can be computed from eq. (3.5) to be

$$p(x) = \gamma^n / \left\{ x^{n+1} [n-1]! \sum_{j=0}^{n-1} \gamma^j / j! ] \right\} \cdot \exp(-\gamma(1/x-1)); \quad (x \leq 1) \qquad (4.1)$$

The whole mean turn-around time of a program, $E_{Bw}$, is composed from

$$E_{Bw} = \sum_{m=n}^{\infty} p_m \, E_B(x=1) + \sum_{j=0}^{n-1} \gamma^j / j! \, \exp(-\gamma) \cdot \int_{x=0}^{1} p(x) . E_B(x<1) dx \qquad (4.2)$$

with $E_B(x=1)$ and $E_B(x<1)$ from eq.(3.1, 3.2) with x=1 and x < 1, respectively, and the multiplicative factor of the integral relating to the probability of the event "n programs do not fit completely in the main memory".

For reasons of discussion of this result, we use approximations for the parachor curve, eq.(2.5), and the dependency of the mean I/O-service time, $E_T$, per demand of partly loaded programs on the portion, x, and on the mean, $E_{Tp}$, (cf. eq. 2.6)

$$E_T = [e + (1-e)\, x^b]\, E_{Tp} \qquad (4.3)$$
$$\text{with } e = 0.2,\ b=2$$

It has been found that the parameter values, e, b, have nearly no influence on the decision which fixed number, n, for the MPM-n policy should be chosen to minimize the mean turn-around time. The main influence comes from the parameter, a, in eq. (2.5).

From fig. 5 we learn that for small values, a = 2, there is a dependency of the optimum number, n, of the MPM-n policy on the memory size. Programs with a good locality are represented by such a - values [7]. For greater values, e.g. a = 5, the policy with n = 1 gains the greatest CPU utilization of all MPM-n policies independent of main memory size. These qualitative results are independent of the parameter, ρ.

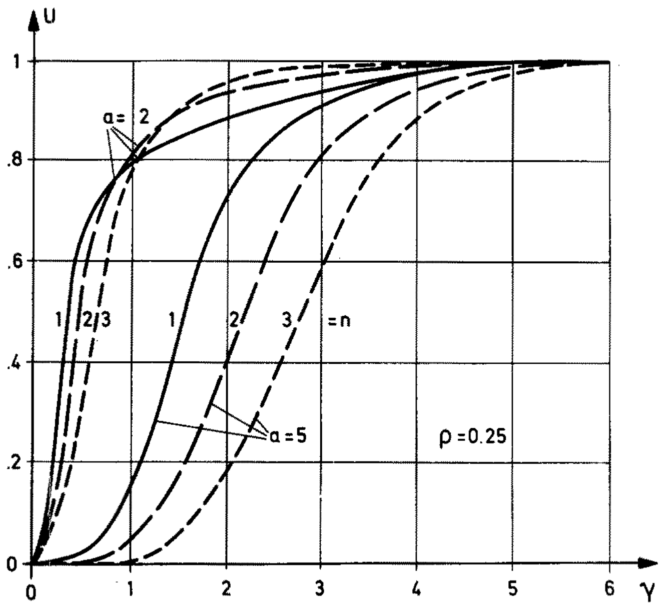

Fig. 5: CPU-utilization, U, over normalized memory size, γ, for the MPM-n policy and two parameter values, a, eq. (2.5). The parameter ρ is given from eq. (3.1).

Under the MPM-n policy with n > 1 sometimes the situation appears that n very large programs must be loaded into a memory of given size which results in a very small portion, x, for each. Then the resulting mean number m(x) of service intervals is very large. It could be argued that in such situations it would be better to temporarily lower the prescribed degree of multiprogramming, n, thereby reducing the number, m(x), substantially. This is the basic idea of the ACM-n policy.

## 5. Throughput under adaptive control of the multiprogramming degree

Instead of prescribing a fixed numer, n, of programs at least to be multiprogrammed, as is done by the MPM-n policy, we now interpret the number, n, as an upper limit. More than n programs are permitted only if they can be loaded completely. Theoretically the number, n, is arbitrary. For reasons of computability we decide to limit it to the smallest possible number, n = 2, for which an adaptive control could be demonstrated.

This means we consider an ACM-2 policy. The number, n, in the ACM-n policy corresponds to n in the MPM-n policy. In addition we introduce an estimated value, of a program's parachor, $x_2$, the portion of a program which corresponds to the working set size to be loaded to avoid thrashing. During our approximate computation of the CPU utilization (and thereby throughput) the parameter, $x_2$, is arbitrary. From the results it is possible to decide whether or not an ACM-n policy is superior to an MPM-n policy and by how much. Moreover, depending on the parachor curve parameter, a, the optimal parachor value, $x_2$, can be determined.

We now compose for the ACM-2 policy the mean turn-around time of programs from three fractions. The first is the same as for the MPM-n policy (cf.eq.4.2) and relates to a number, n, or more fully loaded programs. The second fraction comes from subsystems (in time) where exactly two programs, each with a portion $x_2 \leq x \leq 1$, are multiprogrammed. This fraction is closely related to the second term of the sum in eq. (4.2), the only difference being that the integral is only computed for values $x_2 \leq x \leq 1$. The third fraction is characterized by only one program being loaded completely or partly, because sometimes two programs together happen to be too large, so that they do not fit together with the prescribed portion, $x_2$, in the main memory. The related probability, $P_1$, is (cf.eq.(4.2) with n = 2))

$$P_1 = \sum_{j=o}^{1} \gamma^j/j!\exp(-\gamma) \cdot \int_{x=o}^{x_2} p(x)dx = (1+\frac{\gamma}{x_2}) \exp(-\gamma/x_2) \qquad (5.1)$$

The computation of the third fraction of the mean turn-around time, $E_{Bw}$, is separated into two parts. One of them results from the situation that exactly one program fits completely which appears with probability, $P_1^*$, the other part results from "one program fits only with a portion x, $(0 \leq x < 1)$, in the memory" which has the probability, $P_o^*$. We assume now that programs that are too large to be allocated together and therefore must be monoprogrammed, have the same size d.f. (eq. 2.1) as all other programs. From the appendix (eq. A7, A8) we have

$$P_1^* = \gamma \exp(-\gamma/x_2) \qquad (5.2)$$

and $$P_o^* = [1 + \gamma(1/x_2 - 1)] \exp(-\gamma/x_2) \qquad (5.3)$$

The first term of the third fraction of $E_{Bw}$ is computed from $P_1^* . E_B(x=1)$, cf. eq. (4.2), the second term from

$$P_o^* \cdot \int_{x=o}^{1} p(x)\Big|_{n=1} \cdot E_B(x < 1)\, dx$$

cf.eqs. (4.1, 4.2).
So we have the composed mean turn-around time, $E_{Bw}$, from

$$E_{Bw} = \sum_{k=2}^{\infty} p_k E_B(x=1) + (1+\gamma)\exp(-\gamma) \int_{x=x_2}^{1} p(x) E_B(x<1)dx + P_1^* E_B(x=1) +$$
$$P_o^* \int_{x=o}^{1} \gamma/x^2 . \exp(-\gamma(1/x-1)) . E_B(x<1)dx \qquad (5.4)$$

To discuss our results we again insert our assumptions, eqs.(2.5, 4.3), and compute the mean turn-around time, $E_{Bw}$, and from that the throughput, D, which we normalize on $E_{Cp}$ (cp.eq. 3.1). We do this for different assumed parachor curves represented by the parameter, a, and for distinct assumed parachor values, $x_2$. For two limiting values, $x_2 = 1$ and $x_2 = 0$, we obtain the same results as for the MPM-1 and MPM-2 policies, respectively. This is quite clear from the definition of the ACM-2 policy.

From fig. 6 it becomes evident that the ACM-2 policy is superior in respect to CPU utilization to both policies, MPM-1 ($x_2 = 1$), and MPM-2 ($x_2 = 0$), whenever the parachor value, $x_2$, is chosen appropriately. The utilization gain for small memories, $0.4 \leq \gamma \leq 2$, is sometimes 10 % and more.

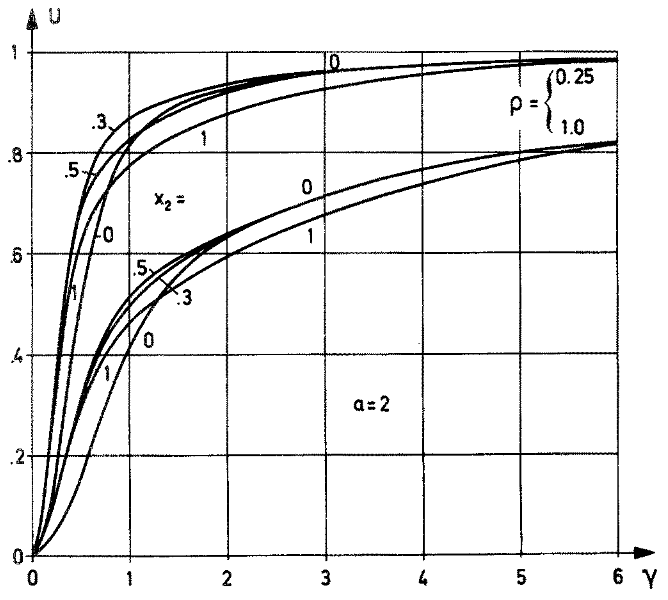

Fig. 6: CPU-utilization, U, over normalized memory size, for the ACM-2 policy and two parameter values, $\rho = 0.25, 1$. The assumed parachor values are chosen to be $x_2 = 0.3, 0.5$. $a = 2$.

Remember that the CPU utilization of the ACM-2 policy does not depend critically on the estimate of the parachor, $x_2$. A wrong estimate, for example, does not result in thrashing. Our results correspond in some sense to the work, published in [3], where also the degree of multiprogramming, n, may change dynamically, but we have used an analytic model while simulation methods are used in [5].

From the graph, fig. 5, we obtain for an assumed parachor curve, described by a value, a = 5, that under the MPM-n policy it would be optimum to allocate only one program to main memory whenever more than one program does not fit completely into it. Under the ACM-2 policy it results from eq. (5.4) that for a value, a = 5, it is also superior to the best MPM-n policy, namely the MPM-1 policy. Our findings establish the adaptive control of the degree of multiprogramming being much better than a fixed degree of multiprogramming for partly loaded programs whenever the parachor value could be estimated approximately in advance. The well-known rule of thumb is verified that multiprogramming is only advantageous if the programs each have their parachor loaded into the main memory.

From the graphs, fig. 5,6, we find that for large memory sizes, ($\gamma > 1.4$), the MPM-3 policy is advantageous over the ACM-2 policy, independent of the parameter value, $x_2$. This indicates that an ACM-n policy, with $n > 2$, would be better than the ACM-2 policy. Analytical computations to verify this suspicion have not been carried out.

One important advantage of the ACM-n policy is that its CPU utilization is never less than the smallest possible utilization, under any MPM-n policy. A second advantage results from the chance to reach a substantially better CPU utilization by an appropriately chosen parachor value, $x_2$, for the ACM-n policy than under any MPM-n policy. Further work on this subject may be successful for the general ACM-n policy with different assumed parachor values, $x_n$, for the dynamic change of the degree of multiprogramming from n to n-1 programs.

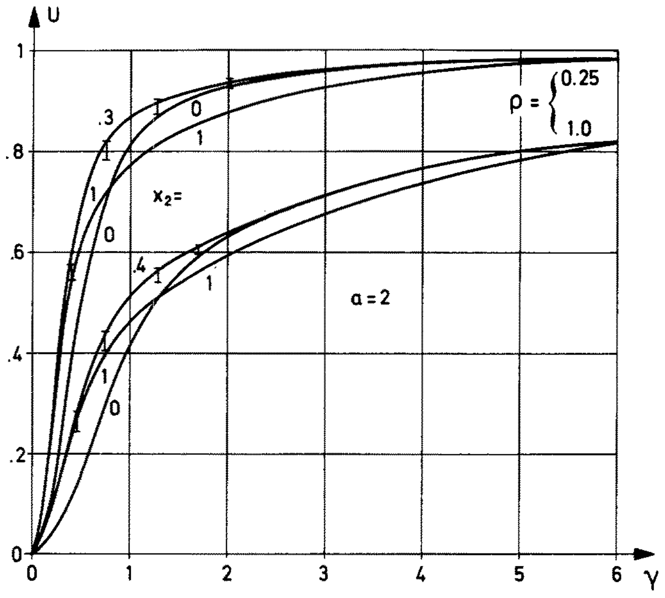

Fig. 7: Computational and simulation results for the CPU-utilization. The marks are the confidence intervals for a 95%-level of confidence. Parameters of the model are as given by fig. 6.

A simulation study made has had two objectives: firstly to verify our computational results and secondly to test whether or not the optimal value $x_2$, could be determined by experiments. From the graph, fig. 7, we learn that the CPU utilization of our simulation model is in-satisfactory agreement with the computed results. The optimal parachor values, $x_2 = 0.3$, $0.4$ for parameters $\rho = 0.25$, $1$, respectively, for which the marks are defined were automatically found by our simulation model. The automatism was obtained by re-running the same workload, characterized by the parameter, a, on the same system, characterized by the parameters, $\rho$, $\gamma$, changing for each run the assumed parachor value, $x_2$, as, long as a nearly maximum CPU utilization was reached. From the graph, fig. 6, we know that the utilization depends uncritically on the chosen value, $x_2$, as long as it is near the optimum value of $x_2$. This was also confirmed by the simulation results. From eq. (5.4) it could be shown that the optimum value, $x_2$, depends on the memory size but this dependency is non-essential and can be neglected in most applications.

## 6. Conclusions

A model of a computer system and its workload is introduced and CPU utilization (and by this throughput) is computed. Program size is assumed to be negative exponentially distributed. This results in a very complex network (in time) with an infinite number of states. The complex network is decomposed in simple subnetworks for which closed solutions from queueing theory are available. Composing these solutions we obtain an approximate outcome for the complex network.

The degree of multiprogramming is controlled by an adaptive policy, ACM-n, which aims at always keeping the parachor of each program in the main memory and maintains this by dynamically changing the degree of multiprogramming. CPU utilization under this policy is compared with the outcome of a non adaptive allocation policy, MPM-n, and the superiority of the ACM-n policy is demonstrated.

Acknowledgments

The author is indebted to Prof. Dr. J. Swoboda for valuable discussionsand the computations added as an appendix.

## References

[1] P.J. Courtois, Decomposability, instabilities and saturation in multiprogramming systems, Comm. ACM, Vol. 18, No. 7 (1975) pp. 371-77.

[2] P.J. Denning, G.S. Graham, Multiprogrammed memory management, Proc. IEEE, Vol. 63, No. 6, June 1975, pp. 924-39.

[3] C.J. Kuehner, B. Randell, Demand paging in perspective, Am. Fed. Inform. Proc. FJCC, 1968, pp. 1011-1018.

[4] R.R. Muntz, Analytic models for computer systems analysis, Lecture Notes Comp. Science, 8, 1974, Springer Berlin/ Heidelberg/New York, pp. 246-65.

[5] H. Opderbeck, W.W. Chu, Performance of the page fault frequency replacement algorithm in a multiprogramming, environment, IFIP congress 1974, Stockholm, Inf.Process. 74, North Holland Publishing Company (1974), pp. 235-41.

[6] B. Walke, Queueing networks with degenerate exponential servers, Wiss. Ber. AEG-TELEFUNKEN 48, 1975, H.4, S.153-57.

[7] B. Walke, Durchsatzberechnung für Rechenanlagen bei wählbarer Aufteilung des Arbeitsspeichers unter mehrere Programme unterschiedlichen Platzbedarfs, PhD-Thesis, University of Stuttgart, 1975 (in German).

## Appendix

Assume the random variables, G1, G1', having the p.d.f. given by eq. (2.1). We are interested in the p.d.f. of G1 conditioned on the sum, (G1+G1'), being greater than a given value g2. The abbreviations $G1+G1'=G2$, $B = \{G1 > g1\}$ , $F = \{G2 > g2\}$ are introduced with G1, G2, being independent variables. Then for $g1 = s$ and $g2 = s/x_2$ the p.d.f. P(B|F) means that one program out of two randomly chosen programs fits not completely in the main memory, conditioned on the event "two programs together do not fit each with the portion, $x_2$, $(0 \leq x_2 \leq 1)$" (i.e. $x_2 \cdot G2 > s$ and $G1 > s$). From eq. (2.1) we have

$$P(B) = \exp(-g1/E_G) \qquad (A1)$$

The p.d.f., P(F), can be computed from the density eq. (4.1) inserting parameter values, n = 2, s = 0, by integration $\int_0^1 p(x)dx$ to be an Erlang -2-d.f.

$$P(F) = (1+g2/E_G)\exp(-g2/E_G) \qquad (A2)$$

The probability, $P(F|B)$, is computable and from this the computation of the probability, $P(B|F)$, is possible using the well-known relationship

$$P(B \cap F) = P(B|F) \cdot P(F) = P(F|B) \cdot P(B) \qquad (A3)$$

The condition, B, means that a program of size, G1, has a minimum size, g1, but if it is larger than g1 then the variable G1 is negative exponentially distributed with mean $E_G$. The other program with the size G1' follows the d.f., eq. (2.1). From these considerations we derive

$$P(F|B) = P(\{g1+G2\} > g2) = P(G2 > \{g2-g1\}) \qquad (A4)$$

without a condition in the expression on the right hand side.

Eq. (A4) is related to eq. (A2) in that it also is an Erlang -2-d.f.

$$P(F|B) = \begin{cases} \left(1 + \frac{g2-g1}{E_G}\right) \cdot \exp\left(-\frac{g2-g1}{E_G}\right) & \text{for } g2 \geq g1 \qquad (A5) \\ 1 & \text{otherwise} \end{cases}$$

We are only interested in the solution for $g2 \geq g1$. From eqs.(A1, A2, A3) we then have, with $g2 = s/x_2$ and $g1 = s$,

$$P(B|F) = \frac{1 + \gamma\,(1/x_2 - 1)}{1 + \gamma} \qquad (A6)$$

To compute the unconditional probability, $P_o^*$, i.e.

$$P_o^* = P(\{G1 > s\} \cap \{G2 > s/x_2\})$$

we use the relationship of eq. (A3)

$$P_o^* = P(B \cap F) = P(B|F) \cdot P(F) = [1+\gamma\,(1/x_2-1)] \cdot \exp(-\gamma/x_2) \qquad (A7)$$

From the complementary probability, $1-P(B|F)$, we have the probability, $P_1^*$, for monoprogramming with a completly loaded program

$$P_1^* = [1-P(B|F) \cdot P(F) = \gamma \exp(-\gamma/x_2) \qquad (A8).$$