Early Use of Reversed Rate Monotonic Analysis for the Estimation of Required Computing Power in Hard Real-Time Systems

Ramón Somoza
Support Analysis for Software Manager
Construcciones Aeronáuticas, S.A.
Madrid - SPAIN

Abstract: Hard real-time systems often encounter the problem of insufficient computing power for the software to meet the specified timing constraints. This results in the need for redesign, code optimization, severe support problems, and, of course, higher costs.
The author proposes a methodology to estimate early in the System design cycle the required CPU computing power and CPU clock frequency, as to be able to meet the specified scheduling and timing constraints, and to ensure the future growth of the system. Finally, he also provides guidelines for the assessment of the software growth capacity.

A. TOO MUCH SOFTWARE, TOO LITTLE POWER

It is quite typical that, for airborne computers, and for hard real-time systems in general, the software designers find out to their dismay -usually halfway through the Design- that they will be unable to comply with the timing requirements, as the software will use up far more computing capacity than was initially expected. This usually implies a redesign of the hardware, with the associated costs and programme delays, or an extensive optimization of the software, unfortunately also to a a great cost.

This kind of situation is bad for the designers, but also for those that have to ensure that a particular piece of software is made supportable, i.e., that it can be properly modified in the future. Specially on items with a very long life cycle -and an aircraft has typically a life cycle of 20-30 years- there is an extreme need for growth, as the system will have to be modified in the future, in order to adapt to changing environments, new or modified functionality, or simply faster response times due to the interfacing with more modern equipment. Yet, if optimization is already required during the design, it is unlikely that the software could be enhanced in the future. This will result in a severe growth handicap, specially for advanced weapon systems, and ultimately it will cause replacement of the "limited" computer by a more powerful one - and almost certainly also requiring the development of new software.

Finally, this problem is also very preoccupying for the end users, given the great operational impact that such limitations have, the associated reduced functionality, the relatively small useful life of these systems and the extremely high life-cycle cost for such unsatisfactory results. Keeping in mind that software support can account for up to 80% of the full life cycle cost, a useful life of at best 8 years would imply also an *annual* support cost of 50% the development cost! But, as Wing Commander B.J. Barker and Squadron Leader B. Hambling from the RAF [Barker82] report:

> *Within the UK the Jaguar development was limited from the outset by fully occupied core space and processor time. The Tornado GRI computer power requirement was doubled even before initial development was completed; current analysis shows that the Harrier GR5 is likely to enter service with considerably less spare computing capacity than originally specified.*

While this was stated back in 1982, the problem is still on-going. Many of the B2-A computers (and in particular the Fuel System computer) had severe computing capacity problems even before delivery of the aircraft. On another on-going aircraft development program, a recent survey of 8 major airborne equipment showed an average CPU usage of 68% - halfway through development!

Possibly part of the problem is that, in most cases, the hardware has been selected before the specification of the software, and without a serious assessment of the software size. Lack of software metrics, but also lack of information about compiler efficiency or CPU throughput and, specially, no initial estimation of the required scheduling needs, has lead to the selection of inadequate CPUs, that later on will prove to have insufficient computing power to process the software that will execute on them. Similarly, an excessive standardisation -such as selection of a same single CPU for all computers on an aircraft, independent of their individual computational needs- often contributes to aggravate the problem. Ultimately, the software engineers are faced with an impossible choice - how to shoehorn too much functionality into a computer with too little computing power. Under these circumstances, software growth capability becomes an empty word, and the software users will face increasingly high software costs.

It is, however, possible to predetermine the processing power needs, on the basis of well-established software engineering techniques, as well as to assess the required growth capacity for software support purposes. The method described in this paper, called Reversed Rate Monotonic Analysis (RRMA) is based on the extension of one of such techniques.

B. RATE MONOTONIC ANALYSIS

Liu and Leyland [Liu73] studied the problem of scheduling a set of independent periodic tasks with hard periodic deadlines. In their classic paper, they determined that the optimal fixed priority algorithm was the Rate Monotonic Algorithm, where a task with a shorter period is given a higher priority than tasks with longer ones. Given that the worst case utilization is bound, the Rate Monotonic Analysis (RMA) has been widely used in software engineering for the schedulability analysis of multitasking systems, specially in hard real-time systems. RMA is recognized as a basic engineering tool for software design, and has been widely discussed, such as in [Sha89], [Sha90].

The basic theorem for the Rate Monotonic Scheduling as per [Liu73] is given by:

Theorem 1: A set of *n* independent tasks scheduled by the Rate Monotonic Algorithm will always meet its deadlines, for all task phasings, if

$$\frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} \leq n(2^{\frac{1}{n}} - 1) = U(n)$$

where C_i and T_i are the execution time and period of task τ_i respectively.

This Theorem offers a sufficient worst case condition that characterizes the schedulability of a set of tasks under the Rate Monotonic Algorithm. U(n) indicates the bound of processor utilisation rate that is required by the different tasks. It is possible to use the additional processing capacity, say, for a set of background tasks with no strict timing requirements, but none of the background tasks will be able to meet any kind of hard deadlines.

Now, the RMA methodology has been used so far to determine during the early software design phases the schedulability of a certain set of tasks with hard deadlines, by estimating their execution time. One of the possible methods to estimate this execution time is, obviously, to estimate the size of the individual tasks (in effectively executed LOC), and then multiply this value by an average execution time per LOC, for that particular CPU.

The Reversed Rate Monotonic Analysis (RRMA) takes however the opposite approach, by assuming that a set of tasks will be schedulable under the Rate Monotonic Algorithm, and then calculating backwards the computing power that would be required to meet this assumption. Given the source code to be executed for each individual task, it would suffice to determine the average execution time per LOC to identify whether a specific CPU could provide the necessary throughput.

While John Lehoczky et al. [Lehoczky89] determined that the algorithm proposed by Liu and Leyland was quite pessimistic, and showed that the threshold of schedulability by the Rate Monotonic Algorithm is located at 88% of the CPU throughput, it must be pointed out that early in the design -and specially in the architectural design- there might be no such thing as a defined solution, and, given also the little reliability of estimations at this stage, it would be advisable to consider the absolute worst case, given the high risk associated to this selection.

C. TASK EXECUTION TIME

The Rate Monotonic Algorithms is based on two factors for each task τ_i: the execution time C_i and the period T_i of that particular task. Now, the periodicity of specific functions can usually be determined quite early in the architectural design - after all, you cannot specify an architecture without knowing what it will do. The estimation of the execution time is however more inaccurate, as it will depend both on the functionality and on the CPU processing power. Practice shows that the estimation of one single factor can be reasonably accurate - but a combination of two is usually not very realistic.

On the other hand, it is possible to reasonably estimate the actually executed code of a task, be in Lines of Code (LOC), Bangs, Function Points or any other appropriate metric that ultimately provides a size of the code that will be executed during each task iteration. In fact, the amount of executed code is actually easier to estimate than the total of developed code, which is what is usually done, as it is strictly related to the functionality that it has to embody. What is the required is the conversion of the executed code size into an execution time.

Theorem 2: The execution time C_i of any running periodic task τ_i is always such that

$$C_i = \frac{R_i \times S_i + m_s}{P}$$

where C_i is the execution time, S_i the number of executed high-level statements of task τ_i, R_i the conversion ratio of source statements into machine code instructions for the used compiler, m_s the number of machine code instructions required for context switching, and P is the processing throughput (number of machine code instructions/second) for that particular CPU.

Proof: A periodic task executes during each of its periods the application code of the task, as well as the necessary code to switch between that task and the next one, if there is such. Given that each of the high-level code statements is converted by the compiler into R_i machine code statements, the application code will consist of $R_i \times S_i$ machine code statements. If the context switching takes on average m_s machine code instructions, the total machine code to be executed during each task period shall be $R_i \times S_i + m_s$. A CPU with a processing power P will execute p machine code instructions per second. Hence, the total processing time C_i will be the aforementioned total number of machine code instructions divided by p.

While in case of one single task the context switching is obviously 0, in case of multiple tasks the context switching time can be quite important, specially when there are tasks with a very short period, which adds a quite heavy task switching overhead.

The context switching in Theorem 2 is given in machine code instructions, as in most cases this will be performed by the compiler run-time system. The m_s value can be obtained either from the compiler vendor, by effectively counting the number of instructions in the context switching routine of the run-time system, or by benchmarking the context switching time c_s and making $m_s = c_s \times P$, being P in this case the throughput of the CPU where the benchmarking took place. In case the context switching were performed by a specifically written scheduler, then m_s would be replaced by $S_s \times R_s$, where S_s would be the number of effectively executed source lines of code of the scheduler, and R_s the LOC-to-machine code ratio of the compiler used for the compilation of the scheduler.

It should be noted that the conversion ratio R_i might be different for each different task, as each of them might have been written in a different programming language, and use therefore a different compiler. If written in Assembler, R_i would obviously be 1, but for other languages one might expect a higher value. The number of executed code statements S_i would of course be dependent on each individual task, but the throughput P would not be related to any individual task, as it is a hardware characteristic that would apply to all tasks executed on that CPU.

Theorem 3: A set of *n* independent tasks scheduled by the Rate Monotonic Algorithm will always meet its deadlines, for all task phasings, if

$$P \geq \frac{1}{U(n)} \sum_{i=1}^{n} \frac{R_i \times S_i + m_s}{T_i}$$

where P is the processing throughput, m_s is the number of machine code instructions required for context switching, and R_i, S_i and T_i are respectively the compiler instruction conversion ratio, the number of executed source code statements and the period of task τ_i respectively.

Proof: If we multiply both sides of the equation by U(n)/P, we obtain:

$$U(n) \geq \sum_{i=1}^{n} \left(\frac{R_i \times S_i + m_s}{P \times T_i} \right)$$

In this equation we now replace $(R_i \times S_i + m_s) / P$ by C_i as per Theorem 2, and we obtain the equation of Theorem 1, which states that this set of tasks is schedulable as per the Rate Monotonic Algorithm.

It should be observed that the required computing power depends only on the size of the individual tasks (S_i), the period of those tasks (T_i) and the compiler efficiency (R_i and m_s), which is also what one could reasonably expect. Obviously, the size of the executed source code S_i will not depend on the target CPU, in case it is written in a high-level language. However, both R_i and m_s will greatly depend not only on the individual compiler, but also on the individual instruction set of the specific CPU. It should be evident that both will be quite high on, say, a RISC machine (more individual operations required), but this will be offset by the great processing power of such machines. On the other hand, a CISC computer with a more sophisticated instruction set will have less computing capacity, which will be compensated by much lower R_i and m_s values. The RRMA approach cannot be used for comparison purposes between different CPUs, as the throughput value will in that case reference different instruction sets, and these comparisons become meaningless.

E.. CPU FREQUENCY DETERMINATION

The RRMA is therefore used to determine the computing power requirements on a specific CPU, and the result is given in machine code instructions per second. Given that this CPU is clocked at a certain frequency, the execution of each machine code instruction will be function of the CPU clock frequency, and therefore:

$$f = P \times c_m \tag{4}$$

where c_m is the average number of machine cycles required for the execution of one machine code instruction.

Equation (4) shows a linear relationship between the frequency and the processing power of a specific CPU. Thus, and given that c_m is constant, doubling the frequency would also imply doubling the processing power. Unfortunately, things are not that simple. The frequency cannot increase indefinitely, as the CPU has been designed with a maximum operating frequency in mind, and it is not possible to go beyond the operating range. But, even in case this range is respected, the CPU will often be constrained by the computer architecture. For example, slow memory will introduce additional waitstates, slowing down the processing throughput. Similarly, shared memory with DMA characteristics or reduced databus bandwidth will also generate serious processing bottlenecks. Experimentally, the relationship between frequency and processing power has been shown to be:

$$\lambda(f) \times f = P \times c_m \tag{5}$$

where $\lambda(f)$ is a complex function depending on the computer architecture, with $0 < \lambda(f) \leq 1$.

Theorem 6: A set of *n* independent tasks scheduled by the Rate Monotonic Algorithm will always meet its deadlines, for all task phasings, if

$$f \geq \frac{c_m}{\lambda(f) \times U(n)} \sum_{i=1}^{n} \frac{R_i \times S_i + m_s}{T_i}$$

where f is the CPU frequency, c_m is the average number of cycles per machine code instruction, m_s is the number of machine code instructions required for context switching, $\lambda(f)$ is a complex function depending on the computer architecture, with $0 < \lambda(f) \leq 1$, and R_i, S_i and T_i are respectively the compiler instruction conversion ratio, the number of executed source code statements and the period of task τ_i respectively.

Proof: By multiplying both sides of the equation by $\lambda(f)/c_m$, and then replacing the left-hand side of the equation by P, as indicated in equation (5), we obtain the equation given in Theorem 3, which states that these tasks will be schedulable as per the Rate Monotonic Algorithm.

F. PROCESSING POWER CALCULATION EXAMPLE

Let us assume that, early in a development programme, we have to assess the throughput requirements for a certain CPU. On the basis of the processing specification, it is determined that there will be a total of six different tasks, with the characteristics outlined in Table 1. We also assume that we will use a single programming language, and benchmarking of individual compilers shows that the typical number of generated instructions per LOC for this type of application is $R = 5.7$ instructions. Our compiler vendors, on the other hand, inform us that the typical number of context switching instructions $m_s = 215$.

Table 1

	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6
Effective Lines of Code S_i	3,600	1,700	9700	8,900	9,800	4,800
Period T_i	40 ms	20 ms	320 ms	160 ms	100 ms	40 ms

By applying the equation of Theorem 3, we obtain that the required processing power P:

$$P \geq \frac{1}{U(6)}\left(\frac{5.7\text{x}3600 + 215}{0.040} + \frac{5.7\text{x}1700 + 215}{0.020} + \frac{5.7\text{x}9700 + 215}{0.320} + \frac{5.7\text{x}8900 + 215}{0.160} + \frac{5.7\text{x}9800 + 215}{0.100} + \frac{5.7\text{x}4800 + 215}{0.40}\right)$$

$$P \geq 1.361^*(518{,}375 + 495{,}250 + 173{,}453 + 318{,}406 + 560{,}750 + 344{,}688) = 3.28 \text{ MIPS}$$

A processing power P of 3.3 MIPS will therefore suffice to meet the execution requirements. Unfortunately, we cannot benchmark any computer running this CPU, so it is difficult to calculate the required frequency in order to obtain the above required processing power. However, the CPU manufacturer states in the microprocessor manual and Technical Data Sheets that the observed (weighted) instruction timing takes on average 6.7 clock cycles. Therefore, we can state that $c_m = 6.7$ cycles/instruction. We will assume that the computer architecture will put no constraints on the CPU throughput, that is, $\lambda(f) = 1$. By applying Theorem 6 we therefore obtain:

$$f \geq \frac{6.7}{1 \times U(6)}\left(\frac{5.7\text{x}3600 + 215}{0.040} + \frac{5.7\text{x}1700 + 215}{0.020} + \frac{5.7\text{x}9700 + 215}{0.320} + \frac{5.7\text{x}8900 + 215}{0.160} + \frac{5.7\text{x}9800 + 215}{0.100} + \frac{5.7\text{x}4800 + 215}{0.40}\right)$$

$$f \geq 9.1187^*(518{,}375 + 495{,}250 + 173{,}453 + 318{,}406 + 560{,}750 + 344{,}688) = 21.98 \text{ MHz}$$

Given that it is unlikely that the CPU manufacturer can supply a processor for such a weird frequency, a CPU clocked at 25 MHz will obviously do the job.

G. SOFTWARE SUPPORTABILITY ASPECTS AND RRMA

So far we have discussed how to calculate early in the design the required computing power for the scheduling of a set of predefined tasks, that will implement a specified functionality. While this is no doubt quite useful, it is also very necessary to assess the application of RRMA to software supportability aspects, i.e., the capability to be able to extend that software in the future.

At the beginning of this paper, we stated the need for growth, given that a limited growth also implies a short operational life and a very high life-cycle cost. Note that it is more expensive to "fit" the possible enhancements into an already overburdened computer, due to the need for redesign and optimization - but, on the other hand, it is usually not cost-effective to provide a massive computing power that will not necessarily be required in the future. A certain amount of power is required - but not too much!

It is for this reason that software supportability requirements are included in system specifications, quite often as a requirement such as "*The CPU shall have a 100% growth capacity in processing power.*"

Such a requirement is unfortunately not clear and, whatever it means, is usually not met. The intention of the writers of such specifications is probably that the software execution should not burden more than 50% of the CPU time, and sometimes the specification even uses this wording. The idea, apparently, is that if the software takes only 50% of the CPU utilization, it could process twice as much functionality, or twice as much code.

This idea is wrong. While this could be true in a linear program, modern real-time systems are multitasking entities, and the sequential approach can no longer be applied. John Lehoczky *et al.* [Lehoczky89] showed that the threshold of schedulability by the Rate Monotonic Algorithm is located at 88% of the CPU throughput. Though Liu and Leyland [Liu73] showed that the optimal scheduling algorithm was the *nearest deadline algorithm,* which can have a worst case bound of 1.000 (i.e., can meet all deadlines up to full processor utilization), the RMA is more realistic, not only because of its simplicity, but also as it can still meet the scheduling requirements with transient overloads, or even with aperiodic tasks [Lehoczky87], [Sprunt89].

But, if only 88% of the CPU power is effectively usable for multitasking systems, then a system using 50% of the CPU will have a growth capacity of only 88-50=38% of the CPU throughput! Still worse, if we consider the absolute worst case, where the effectively usable CPU power would be equal to U(n), this growth capability would be even worse - for six tasks, we would have a growth of only U(6)-50% = 73.5%-50% = 23.5%. Given that the worst case utilization bound of U(n) converges to $\log_e 2$ = 0.693, in extreme cases (many tasks) the growth might be of only 69.3%-50% = 19.3%!

When starting to specify growth, it should be clear *what* should grow. For example, to obtain a "100% growth", the most easy approach would be to double the requirement for the processing power P, or alternatively the frequency f. In our previous example, this would imply a frequency of 43.96 MHz. But would we really "double" the capacity?

Two further examples will clarify this. In one case, we will double the size of the executable statements of each task as given in the previous example, in the second case we will duplicate each of the tasks, that is, for each of the existing tasks we will create a second task with the same executable code and the same task period.

In the first case, and given that each task τ_i will ultimately execute 2 × S_i statements, the equation of Theorem 3 will be:

$$f \geq \frac{c_m}{\lambda(f) \times U(6)} \sum_{i=1}^{6} \frac{R_i \times 2S_i + m_s}{T_i} = 43.75 \text{ MHz}$$

Note that this value is slightly lower than the one that we would have obtained if we had doubled the frequency that we had previously obtained. However, if we duplicated each of the tasks, we would obtain:

$$f \geq \frac{c_m}{\lambda(f) \times U(12)} \sum_{i=1}^{6} 2 \times \frac{R_i \times S_i + m_s}{T_i} = 45.28 \text{ MHz}$$

The higher value is explained partly because we have to use U(12) instead of U(6), as we will have now a total of 12 tasks. Similarly, and giving that there are more tasks now, the context switching will take place more often, also contributing to additional computing power requirements.

The two examples given above demonstrate that it is not sufficient to assess the potential growth of the executed code for supportability purposes, but that it is also necessary to assess the future schedulability requirements.

Here, unfortunately, we enter a more uncertain ground, as in most cases it is not clear how the system will be expanded when the design is made. It is however a very fertile ground for Pre-Planned Product Improvement, by establishing during the system architectural phase already the potential baselines for extension of the system. For this purpose, an asessment of potential tasks has to be made, defining both their length and shortest expected periodicity. With this, a set of "Dummy" tasks are created, and the established values are introduced in the RRMA equations as if they did really exist, thus obtaining the supportability requirement.

Of course it would be too much to expect that tasks with exactly the same length and period would be generated during software support. Though the gift (curse?) of prophecy is assumed to be part of the prerequisites for the job of software logistician, a little bit of common sense and imagination are more useful than the crystal ball. It is hard work -but not too difficult- to assess additional data polling or communication with faster computers (given free databus bandwidth), interrupt handlers (free interrupt vectors), additional signals to be processed (free channels) and some possible data manipulation that could be implemented in the future. It is not necessary to have it spelled out and exactly predict what will be added in the future - it would be sufficient to have tasks with a same computational weight, meaning tasks whose processing power requirements would be the same as the postulated ones.

This computational weight for a given task τ_j is given by:

$$W(\tau_j) = \frac{R_j \times S_j + m_s}{T_j \times \sum_{i=1}^{n} \frac{R_i \times S_i + m_s}{T_i}}$$

Thus, for the example given above, task τ_1 would have a computational weight of 21.5%, three times more than task τ_3 (7.2%), despite the fact that this latter executes two and a half times as much code as task τ_1 each time it is executed.

It is for this reason that, during support, this computational weight becomes important: As soon as you need optimization, the tasks with a high computational weight are the ones to optimize first, as there each little optimization will have the greatest impact.

H. A WORD OF CAUTION

The RRMA is not a panacea, and should be applied only while being aware of its limitations. The first one is that it does NOT provide an exact value of the computing power that is required, but rather an upper limit. Thus, it is possible to go for excessive power - though in most cases the excess will be within a very reasonable margin.

Similarly, it must be understood that the RRMA is used on the basis of estimations - and all estimations are fallible, as they are performed by human beings. For example, some people find it hard to estimate the amount of effectively executed code. If a certain function has to be performed ten times, they think immediately about a loop - and forget that the code in the loop will be executed ten times. An *if* statement, on the other hand, does not execute completely - an average for all different conditions paths should be taken. But they cannot help thinking about an implementation when they should concentrate on the functionality. Estimating the functionality is more accurate and even easier than thinking about the total of the code - but it does take some getting used to.

On the other hand, this kind of analysis forces the system architects to concentrate on what the software does or is supposed to do before the final specification of the hardware, thereby reducing the risk of careless estimations. Given that it is also necessary to determine the periodicity of the individual functionality, in some cases this might even result in a better understanding of what the system does and, should the required computing power be unfeasible to obtain (tasks with extremely short periods), might lead to a different hardware/software allocation, or even to the use of multiple processors. The opposite is also true - the RRMA might show that even in the worst case there is no need for parallel processing.

Note that RRMA does not assume that the tasks will be scheduled under the Rate Monotonic Algorithm - only that they could be scheduled that way. This implies that there is at least one scheduling method that would permit meeting all hard deadlines without having to resort to hardware or software redesign, or extensive optimization. Similarly, it ensures that at least 100-U(n)% of the CPU will be free to perform background tasks without timing requirements, such as Built-In Test (BIT). Should a higher percentage be however required, then the corresponding code could be added as an additional task with a long period.

Finally, RRMA cannot be used either for comparisons between different CPUs - the comparisons are meaningless as the instruction sets are different. MIPS as a power throughput measurement is a lousy metric for comparison purposes, and should be used with care, possibly only with processors of a same family, with a very similar instruction set. However, RRMA will permit to define the minimum computing characteristics when applied to different CPUs and compilers, so that the hardware designers can select a candidate on the basis of cost, reliability, availability, power consumption, etc., without having also to worry whether it will have the necessary performance.

As a final note, it should be noted that this paper considers only the case of a single CPU. However, the RMA Methodology can be also used for Distributed System Design [Sha92]. The application of RRMA is analogous, though applying a P_j for the processing power of the CPU j on which a specific task τ_j is executed.

It should be made clear that RRMA is a tool, and a tool can be both used and abused. But, if properly used, RRMA can provide a helpful hand, and of course it will be much better that estimating the required computing power by observing the flight of the sacred geese.

References

[Barker82] S.J. Barker, B. Hambling; *The Military User View of Software Support Throughout the In-Service Life of Avionic Systems;* Conference Paper 43 in: AGARD-CP-330, *Software for Avionics,* NATO Advisory Group for Aerospace Research and Development, Avionics Panel`s 44th Symposium, The Hague-Kijkduin, Netherlands, 6-10 September 1982.

[Lehoczky87] J. Lehoczky, L. Sha, J. Strosnider;; *Aperiodic Scheduling in a Hard real-Time Environment,*Proceedings, IEEE Real-Time Systems Symposium, 1987, pp 261-270.

[Lehoczky89] J. Lehoczky, L. Sha, Y. Ding; *The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case behaviour,* Proc. IEEE Real-Time Systems Symposium, CS Press, Los Alamitos, California, Order No. CH2803-5/89, 1989, pp. 166-171.

[Liu73] C.L. Liu, J.W. Leyland; *Scheduling Algorithms for Multiprogramming in Hard real Time Environments;* Journal of the ACM, 20(1), January 1973, pp. 46-61.

[Sha89] L. Sha, J.B. Goodenough; *A Review of Analytic Real-Time Scheduling Theory and its Application to Ada,* in: *The Design Choice,* Cambridge University Press, 1989, pp. 137-148.

[Sha90] L. Sha, J.B. Goodenough; *Real-Time Scheduling Theory and Ada,* IEEE Computer, April 1990, pp. 53-62.

[Sha92] L. Sha, S.S. Sathaye; *Distributed System Design Using Generalized Rate Monotonic Theory,* Carnegie Mellon University, Pittsburgh PA 115213, 4th May 1992.

[Sprunt89] B. Sprunt, L. Sha, J. Lehoczky; *Aperiodic task scheduling for hard real-time systems,* Real-Time Systems, 1989, pp 27-60.