Allocating Hard Real Time Tasks †
(An NP-Hard Problem Made Easy)

Ken Tindell
Alan Burns
Andy Wellings

Real Time Systems Research Group
Department of Computer Science
University of York
England

email: ken@minster.york.ac.uk

ABSTRACT

A distributed hard real time system can be composed from a number of communicating tasks. One of the difficulties with building such systems is the problem of where to place the tasks. In general there are P^T ways of allocating T tasks to P processors, and the problem of finding an optimal feasible allocation (where all tasks meet physical and timing constraints) is known to be NP-Hard. This paper describes an approach to solving the task allocation problem using a technique known as *simulated annealing*. It also defines a distributed hard real-time architecture and presents new analysis which enables timing requirements to be guaranteed.

1. INTRODUCTION

Building real-time systems on distributed architectures presents engineers with a number of challenging problems. One issue is that of scheduling the communication media, another concerns the allocation of software components to the available processing resources. Distributed systems typically consist of a mixture of periodic and sporadic tasks, each with an associated deadline and possibly precedence constraints. Failure to meet the deadlines of critical tasks may lead to a catastrophic failure of the system, and consequently off-line analysis of allocation and processor scheduling is required to guarantee task deadlines.

In general, the three activities of task allocation, processor scheduling and network scheduling are all NP-hard problems [5]. This has led to a view that they should be considered separately. Unfortunately, it is often not possible to obtain optimal solutions (or even feasible solutions) if the three activities are treated in isolation. For example, allocating a task T to a processor P will increase the computational load on P, but may reduce the load on the communications media (if T communicates with tasks on P), and hence the response time of the communications media is reduced, allowing communications deadlines elsewhere in the system to be met. The tradeoffs can become very complex as the hard real time architecture becomes more expressive; a simple and scalable approach is required.

† This work was supported in part by British Aerospace (Commercial Aircraft) Ltd, and by the UK Department of Trade and Industry

Previous approaches to solving the task allocation problem have mostly concentrated on graph theoretic algorithms (for example [6] and[7]) or heuristics (for example [3] and[9]). Most have tried to maximise system throughput (i.e. minimise the computational and communication resource requirements for tasks in the system), often by reducing 'bottlenecks', resulting in allocations which may or may not be schedulable. However, these approaches do not take a global view; they rely on the observation that fast systems (i.e. ones which maximise system throughput) usually equate to schedulable systems. Of course, the major requirement for a hard real time system is to meet deadlines, and most previous allocation approaches do not address this — usually a post-allocation phase is needed to determine the schedulability of a given allocation. For example, the current MARS [8] approach separates the allocation and scheduling problems, solving each in turn, which can lead to sub-optimal solutions. Indeed, when looking for allocations which are merely feasible (i.e. all hard constraints — such as guaranteed deadlines — are met) the partitioned approach can fail to find solutions where an algorithm taking the global view can succeed.

One approach that does attempt to address systematically the global allocation problem is the work of Ramamritham [17], where the allocation algorithm directly addresses the schedulability of the tasks, and hence takes a global view. In Ramamritham's architecture multiple processors are connected by a shared broadcast bus which operates a TDMA (time division multiple access) protocol. The schedulability of tasks in the system is determined by evaluating fixed processor and bus schedules. The tasks are allocated to processors according to a set of heuristics which consist of simple rules. For example, one rule moves tasks which communicate to the same processor (so that communication can take place without using the bus). Another rule moves tasks away from unschedulable processors. A problem with heuristics is that complex tradeoffs can occur which the designer must foresee, and the resolution of conflicts (such as in the two rules above) is not trivial. Also, schedule evaluation can be computationally intensive since a schedule must be evaluated over the least common multiple (lcm) of the task periods, and this may equal the multiple of the task periods.

In summary, the general allocation problem has yet to be adequately addressed. Not only must an allocation satisfy certain hard constraints, but it should also aim to optimise some aspect of the system model. In addition, a proposed solution to the allocation problem must be scalable and be able to encompass a variety of complex system architectures (perhaps even choosing between architectures as part of the allocation process).

As indicated earlier, task allocation can be viewed as a global optimisation problem. It is similar in nature to other problems found in computer science, such as the travelling salesman problem. These problems have been successfully solved by the global optimisation technique known as *simulated annealing* [10]. Simulated annealing is not a heuristic algorithm — it is sufficient to state *what* makes a good solution not *how* to get one, and therefore does not suffer the disadvantages of applying inadequate heuristics. This paper describes how the simulated annealing algorithm can be used to solve the task allocation problem. We believe the algorithm is scalable and able to encompass complex hard real time architectures.

The next section describes the algorithm. In order to focus the application of the algorithm more clearly Section 3 gives a distributed hard real time architecture and presents an analysis of the

architecture. The section also clearly states the task allocation problem for the example architecture. Section 4 shows how the simulated annealing algorithm is applied to solving the task allocation problem. Section 5 presents the results of implementing the algorithm, and Section 6 draws conclusions and indicates how further work is using the simulated annealing algorithm. Appendix 1 gives a large task and processor set and shows the results of applying the algorithm.

2. THE ALGORITHM

Simulated annealing [16, 10, 1] is a global optimisation technique, which attempts to find the lowest point in an energy landscape. The technique was derived from observations of how slowly cooled molten metal can result in a regular crystalline structure. The distinctive feature of the algorithm is that it incorporates random jumps to potential new solutions. This ability is controlled and reduced as the algorithm progresses.

In order to describe the algorithm some definitions are needed. The set of all possible allocations for a given set of tasks and processors is called the *problem space*. A *point* in the problem space is a mapping of tasks to processors. The *neighbour space* of a point is the set of all points that are reachable by moving any single task to any other processor. The *energy* of a point is a measure of the suitability of the allocation represented by that point (poor allocations are high energy points). The *energy function*, with parameters, determines the shape of the problem space — it can be visualised as a rugged landscape, with deep valleys representing good solutions, and high peaks representing poor or infeasible ones. The allocation problem is that of finding the lowest energy point in the problem space.

A random starting point is chosen, and the energy, E_s, evaluated. A random point in the neighbour space is then chosen, and the energy, E_n, evaluated. This point becomes the new starting point if either $E_n \le E_s$, or if:

$$e^x \ge \text{random}(0,1)$$

Where

$$x = \frac{E_s - E_n}{C}$$

C is the control variable

and 'random' is a uniform random number generator

The control variable C is analogous to the temperature factor in a thermodynamic system. During the annealing process C is slowly reduced ('cooling' the system), making higher energy jumps less likely. Eventually, the system 'freezes' into a low energy state. The structure of the algorithm is sketched below:


```
choose random starting point P_0
choose starting temperature C_0
repeat
    repeat
        E_P := Energy at point P_n
        choose T, a neighbour of P_n
        E_T := Energy at point T
        if E_T < E_P then
            P_{n+1} = T
        else
            x := (E_P - E_T) / C_n
            if e^x ≥ random(0,1) then
                P_{n+1} := T
            else
                P_{n+1} := P_n
            fi
        fi
    until thermal equilibrium
    C_{n+1} = f(C_n)
until some stopping criterion
```

As can be seen from above, the algorithm requires a neighbour function, an energy function, and a cooling function.

The energy function is the heart of the allocation algorithm. It shapes the energy landscape, which affects how the annealing algorithm reaches a solution. An example energy function for the architecture given in the next section is described in Section 4.

The initial temperature, C_0, is chosen so that virtually all proposed jumps are taken, and this temperature can be chosen by the algorithm: Kirkpatrick *et al* [10] pick a low temperature and repeatedly double it until the acceptance ratio (the number of accepted jumps over the number of proposed jumps) is near to 100%. Laarhoven and Aarts [11] take a more mathematical approach and produce a recursive equation which rapidly converges to the ideal starting temperature. The temperature decrease function, $f(C_n)$, is usually a simple multiplication by α, where $0 \le \alpha < 1$. Again, Laarhoven and Aarts propose a more complex function, which dynamically changes the rate depending on the performance of the algorithm.

As can be seen from the algorithm description above, the temperature remains the same over a number of trials until equilibrium is reached. These trials can be modelled as a markov chain, and an equilibrium condition obtained. Kirkpatrick *et al* use a simple condition: the number of downward (energy decreasing) jumps are counted and equilibrium is said to be achieved when the count exceeds a threshold. Laarhoven and Aarts analyse the algorithm mathematically and propose a more complex condition. Both approaches lead to potentially infinite chains (especially at low temperatures) and so an upper bound on the number of trials is enforced (usually about four times the size of the neighbour space).

The stopping criterion can also be determined automatically, and a simple approach is to terminate when no upward or downward jumps have been taken over a number of successive chains.

A plot of Energy against Temperature for a typical problem is shown below (the results were

produced from an actual run of a task allocation program). Temperature is plotted on a log scale so the scale can also represent linear time flowing from right to left. The temperature starts to decrease rapidly at about 10.0 — this temperature is the 'freezing point' of the system in this example, and cooling must be slow enough to prevent the system being trapped in a local minimum, so the cooling rate α is usually between 0.95 and 0.99.

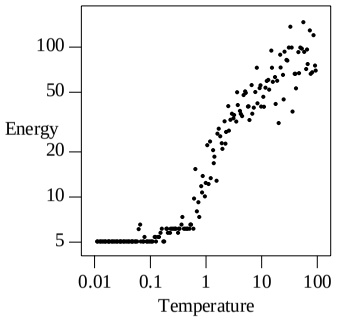

Figure 1

3. A DISTRIBUTED HARD REAL TIME ARCHITECTURE

In order to illustrate the approach, this section describes an example distributed hard real time architecture. Although this architecture is simple we believe that simulated annealing can easily be applied to more complex architectures (work at the Real Time Systems Research Group at York has already applied simulated annealing to the problem of allocating and scheduling precedence constrained hard real time tasks).

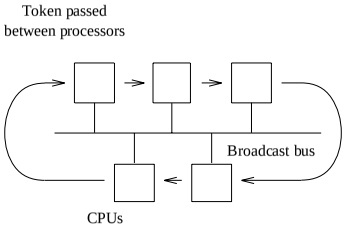

Figure 2: Physical architecture

Figure 2 shows the physical architecture — a number of processors are connected to a broadcast bus, each processor having a fixed processing speed and memory capacity. A number of hard real time periodic tasks execute on each processor. Every period a task executes a piece of code within a bounded number of CPU cycles, and communicates the results of the computation by sending messages to other tasks. Each task has a fixed memory requirement (there is no run-time memory contention). In each period a task may send a bounded number of messages of a bounded size to other tasks. A message sent between two tasks on the same processor is assumed to take zero time to arrive. It is also assumed that message arrivals cause no overhead at the receiving processor

(these two assumptions have been removed in current work, but the analysis is beyond the scope of this paper).

For this simple architecture a distributed version of the rate monotonic algorithm is required — a message sent from a task arrives at the destination task by the end of the period of the sending task (just as with a single processor rate monotonic system the results of computation are made available by the end of the period of the task). Schedulability analysis for distributed rate monotonic scheduling is given below:

SCHEDULABILITY ANALYSIS

To implement distributed rate monotonic scheduling the results of a task must be made available to the destination tasks by the end of the period of the sending task. If the source and destination tasks are located on different processors then the transmission time for messages containing the results must be allowed for (Figure 3).

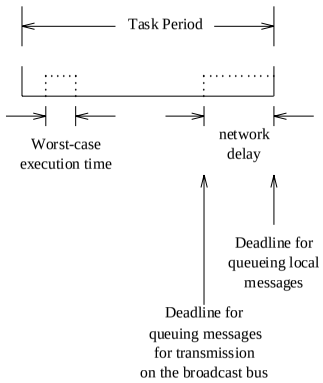

Figure 3

The messages must be guaranteed to arrive within the time allowed (hence a suitable bus protocol is required). To guarantee that a task produces the results sufficiently early the task must execute to a specified deadline. Current rate monotonic scheduling (RMS) theory [14, 12] assumes that the deadline of a task is equal to the period of the task. As can be seen from Figure 3 we need schedulability analysis where the deadline of a task can be less than the period of a task. Deadline monotonic scheduling (DMS) [13] is a static priority scheduling approach (like RMS) where the priority of a task is assigned according to the deadline of the task (a short deadline results in a high priority). Although Leung and Whitehead [13] proved that the deadline monotonic scheduling approach is optimal they did not provide a schedulability test. A schedulability test has recently been derived by Audsley *et al* [2]. The test takes the following form: each of n tasks must complete the execution of their code (in time C) before the deadline D, and hence:

$$\forall_{i,\,1\le i\le n}\ C_i + I_i \le D_i \qquad (1)$$

Where:

n The number of tasks on the processor

C_i The bounded (worst-case) execution time of task i.

D_i The deadline of task i, where $D_i \leq T_i$

T_i The period of task i

I_i The interference due to tasks with a higher priority preempting task i, given by:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j \qquad (2)$$

Tasks are ordered by priority (task 1 has a higher priority than task 2), and priorities are assigned by deadline, such that $\forall_{i<j}\ D_i \leq D_j$.

The test is sufficient — if Equation 1 is true then the task set is schedulable. The deadlines of each of the tasks must be determined before the test can be applied; for a task sending only local messages we have:

$$D_i = T_i$$

For a task which sends a message to tasks on other processors we have:

$$D_i = T_i - N$$

Where N is the maximum delay in sending a message across the bus. More precisely, the delay is the time taken between the message being queued at the sending processor and arriving at the destination processor, and this time must be bounded by using an appropriate protocol. Both MARS and Ramamritham use a statically scheduled TDMA protocol. In this paper we propose a token passing protocol which bounds message delivery, but allows spare bus bandwidth to be used for soft real time messages in a flexible manner.

THE TOKEN PROTOCOL

A processor is only allowed to transmit on the bus if it holds a token, and can only hold the token for a bounded amount of time. After this time, or when the processor has no more data to send, the token is passed on to the next processor in a logical ring. The token holding time at each processor is large enough to guarantee that all messages in a queue on the processor can be sent the next time the token arrives at the processor. Assuming that there is a critical message instant (i.e. all tasks queue all messages simultaneously) the token holding time for processor p, is given by:

$$THT_p = \frac{\sum_{i=1}^{n(p)} M_{i,p} \left\lceil \frac{TRT}{T_{i,p}} \right\rceil}{S} \qquad (3)$$

Where:

$n(p)$ Number of tasks on processor p

$M_{i,p}$ Total size of messages sent on the bus from the ith task residing on processor p

$T_{i,p}$ Period of the ith task on processor p

TRT Token rotation time for the bus

S Speed of the bus

The token rotation time (TRT) is the maximum time taken between successive token arrivals at a processor. Any message queued is guaranteed to arrive at the destination processor within the TRT, and messages can therefore be queued in FIFO order. The TRT is found by summing all the token holding times (plus a small overhead per processor to transmit the token). Thus:

$$TRT = \sum_{j=1}^{P} (THT_j + \tau) \qquad (4)$$

P Number of processors in the system

τ Time taken to transmit the token

Equations 3 and 4 are mutually dependent and a solution can be obtain by iterating to a fixed point. Alternatively, the solution can be found quickly by observing that $TRT \ll T_{i,p}$ if all tasks are schedulable. So:

$$\left\lceil \frac{TRT}{T_{i,p}} \right\rceil = 1$$

$$\therefore THT_p = \sum_{i=1}^{n(p)} \frac{M_{i,p}}{S}$$

□

In summary, using the token protocol analysis and the schedulability test presented earlier the schedulability of tasks in a given allocation can be determined.

However, there are other hard constraints on a feasible allocation for this architecture:

- Some tasks can only reside on a subset of the available processors. For example, a task monitoring a sensor or controlling an actuator must reside on a processor directly connected to the physical hardware. Similarly, the processors may be heterogeneous and executable task images of a certain type can only be run on a processor of that type.
- Some tasks may be replicated for fault tolerance and therefore cannot be allocated to the same processor.
- The memory usage of a processor cannot exceed the fixed capacity.

4. APPLYING THE ALGORITHM

This section describes the application of the simulated annealing algorithm to the task allocation problem for the example hard real time architecture given in the previous section. It should be noted that the neighbour and energy functions presented here are not the only possible functions but are simple ones which have been found to work well in practice.

The neighbour function is simple: choose a random task and move it to a randomly chosen processor. However, better allocations can be obtained if the ‘degree of freedom’ of the system is increased [1]. For example, a situation can occur where two processors A and B are heavily loaded and contain tasks X and Y respectively. A better point can be obtained by swapping X and Y. However, a move of X to B followed by another move of Y to A is unlikely to occur because the first move would result in a high energy point (one of the processors becomes unschedulable). If the neighbour function directly implements task swaps then the jump can take place. This can be

likened to a catalyst in a chemical reaction which allows the energy barrier to a viable reaction to be 'tunnelled'.

The energy function is more complex, and has to penalise the following characteristics of an allocation:

(i) Tasks allocated to the wrong processors

(ii) Replicas allocated to the same processor

(iii) Processors with a memory utilisations > 100%

(iv) Tasks which are not guaranteed to meet their deadlines

These are hard constraints (an allocation with two unschedulable tasks is just as infeasible as an allocation with a single unschedulable task). However, a measure of the 'badness' of an allocation must be given, since if all infeasible allocations were given the same energy there would be no path in the energy landscape to follow to a valley where an acceptable allocation might be found.

Characteristic (i) is penalised by returning an energy proportional to the number of misallocated tasks. A short cut can be made by ensuring the neighbour function never chooses an allocation where a task is misallocated — each task has a set of acceptable processors and only processors from this set are chosen.

Characteristic (ii) is penalised by returning an energy component which is a function of the number of replicas on the same processor:

$$E_{replica} = \sum_{p=1}^{P} \sum_{i=1}^{n(p)} \sum_{j=1, j\neq i}^{n(p)} replica(i,j) \qquad (5)$$

Where $replica(i,j)$ returns 1 if i is a replica of j, and zero otherwise.

Characteristic (iii) is penalised by returning an energy component proportional to the memory usage in excess of the capacity of each processor:

$$E_{mem} = \sum_{p=1}^{P} \max[0, mu(p) - mc(p)] \qquad (6)$$

Where:

$mu(p)$ The memory used on processor p

$mc(p)$ The memory capacity of processor p

Characteristic (iv) can be penalised by returning an energy proportional to the overrun on each task missing its deadline:

$$E_{sched} = \sum_{p=1}^{P} \sum_{i=1}^{n(p)} \max[0, C_{i,p} + I_{i,p} - D_{i,p}] \qquad (7)$$

Where:

$n(p)$ Number of tasks on processor p

$D_{i,p}$ Deadline of i th task on processor p (where $D = T - TRT$ for a task sending bus messages, and $D = T$ for a task sending only local messages)

$C_{i,p}$ Worst-case execution time of i th task on processor p

$I_{i,p}$ Interference for i th task on processor p (see Equation 2)

There may be many feasible allocations, and some way of distinguishing between these would be useful. In this paper the feasible allocation with the lowest bus utilisation is preferred (since more soft real time messages could meet their deadlines with a lower bus utilisation). Therefore another energy component, E_{bus}, is needed, which returns an energy proportional to the bus utilisation. The complete energy, E, is given by:

$$E = k_1 E_{replica} + k_2 E_{mem} + k_3 E_{sched} + k_4 E_{bus}$$

The k weightings allow the prioritisation of the hard constraint components, and balance out the hard constraint components so that the range of values returned by each are similar.

The E_{bus} component penalises allocations which violate a *soft* constraint, and care should be taken to ensure that there is no tradeoff between hard and soft constraints. For example, a situation might arise where the total energy for an infeasible but low bus utilisation allocation is lower than a feasible allocation with a higher bus utilisation. Normally, the algorithm would return the infeasible allocation as the better, and therefore it must be changed to ensure that this does not occur. The anneal step is changed to include two rules:

(1) A jump from an infeasible to a feasible solution always occurs (even if the feasible solution has a higher energy)

(2) A jump from a feasible to an infeasible solution of lower energy may occur, but with a probability which decreases as the temperature decreases.

IMPLEMENTATION CONSIDERATIONS

During the execution of the simulated annealing algorithm the energy function is invoked many thousands of times, and a considerable reduction in the run time of the algorithm can be made if the energy function is evaluated quickly. With the neighbour function described earlier the energy for a neighbouring point can be obtained quickly from the energy of the current point by computing the differences in energy resulting from the change after applying the neighbour function. To illustrate this consider the energy component E_{mem} (See Equation 6). The neighbour function moves a task T from processor P_{old} to P_{new}. When T moves from P_{old} to P_{new} the memory usage of all other processors in the system remains unaffected, and hence the memory penalty due to these other processors remains the same (and need not be recalculated). The new memory penalty can be calculated by removing the penalty due to P_{old} and P_{new} before T is moved, and adding the penalty due to P_{old} and P_{new} after T has moved.

Hence, and from Equation 6, the components of the memory penalty due to P_{old} and P_{new} in the neighbour point, F_{mem}^{neigh}, can be calculated. The penalty $F_{mem}^{current}$ for the current point due to P_{old} and P_{new} is already known, and so the change in E_{mem} due to the move, ΔE_{mem}, can be determined:

$$\Delta E_{mem} = F_{mem}^{neigh} - F_{mem}^{current}$$

$$E_{mem}^{neigh} = E_{mem}^{current} + \Delta E_{mem}$$

Where:

E_{mem}^{neigh} The value of E_{mem} for the neighbour point

As can be seen from above, the evaluation of ΔE_{mem} has algorithmic complexity $O(1)$, whereas the evaluation of E_{mem} has algorithmic complexity $O(n)$ — the computation of E_{mem}^{neigh} is much

faster using ΔE_{mem}. The energy components $E_{replica}$ and E_{bus} can be evaluated for the neighbour point using appropriate ΔE functions formulated in a similar way to ΔE_{mem}.

However, the formulation of ΔE_{sched} is more complex and the following points must be noted:

- When task T moves from P_{old} to P_{new} the interference for T must be recalculated.
- The interference due to T on lower priority tasks located on P_{old} must be removed.
- The interference due to T on lower priority tasks located on P_{new} must be added.
- The movement of T may affect the utilisation of the bus, changing the TRT. This will affect the deadlines of all tasks which send messages on the bus and may change the priority ordering of tasks in the system, and hence interferences must be recalculated.
- The movement of T away from tasks on P_{old} which communicate with T and to tasks on P_{new} which communicate with T may change the deadlines of such tasks and hence the relative priorities and interferences must be recalculated.

Using ΔE functions has proved to be valuable in the implementation of the task allocation algorithm — in one implementation the algorithm executed about a hundred times faster when ΔE functions were used. Further reductions in running time can be obtained by parallelising the algorithm [18, 1, 20].

5. RESULTS

Evaluating the performance of the simulated annealing algorithm is difficult, since to compare the result produced with the optimal result requires the optimal result to be found! Nevertheless, some measure of the performance of the algorithm can be made (Appendix 1 presents an example allocation problem and shows how the algorithm finds a good solution).

To check the performance of the algorithm a small problem (9 tasks and 5 processors) was used and the optimal allocation found by brute-force evaluation of all possible allocations. The simulated annealing algorithm produced identical optimal results. Of course, finding the optimal allocation for a small problem is no guarantee that an optimal allocation is always obtained for larger problems.

The algorithm was also tested with a contrived larger problem (where the tasks could be perfectly partitioned into tightly coupled clusters) with an optimal energy of zero. The annealing algorithm found an allocation with this optimal energy; each cluster was identified and located on a separate processor.

A good test of an allocation algorithm is to take a soluble problem and *add* resources; the algorithm should return an allocation no worse than the result of the original problem *; this was observed for our simulated annealing approach. The report [19] details these tests and presents the results found.

The simulated annealing algorithm has been extensively applied in many fields, including VLSI placement, routing, and image processing. (See [1] for an extensive survey). The performance results obtained from these applications are encouraging — Aarts and Korst [1] obtain solutions to the travelling salesman problem to within 2% of the optimal solution. We contend that this problem is structurally similar to the task allocation problem. Price and Salama [15] have

* Not quite true. Each processor takes time τ to pass-on the token, so more processors increase the TRT. A version of the program where $\tau = 0$ was used.

compared the use of simulated annealing and heuristics for the allocation of non-real-time tasks, and found that simulated annealing produced better results. They also assert that simulated annealing characteristicly requires computation time that scales as a low order polynomial problem. Bollinger and Midkiff [4] have applied simulated annealing to the problem of allocating tasks and processor links in a point-to-point non-real-time network, and have found that simulated annealing applied to this problem is $O(N^{2.8})$. Kirkpatrick *et al* [10] found that simulated annealing applied to the travelling salesman problem is approximately $O(n \log n)$.

6. CONCLUSIONS

Simulated annealing has proved to be an effective approach to task allocation: the results produced are near-optimal, the algorithm scales well for large problems, and there are several implementation enhancements to increase the speed of the algorithm. The great beauty of the algorithm is that it is sufficient to say *what* makes a good solution without describing *how* to get one. This becomes very important as real time architectures become complex with many global tradeoffs — it may be difficult or impossible to obtain good heuristics for many hard real time problems (Appendix 1 shows how the algorithm was easily changed to balance processor utilisation).

7. ACKNOWLEDGEMENTS

The authors would like to thank John McDermid, Dave Scholefield, and Russell Beale for helpful comments on an earlier version of this paper. The authors would also like to thank Neil Audsley for help with deadline monotonic scheduling, and Andy Hutcheon for comments on the implementation of the algorithm.

8. REFERENCES

1. E. H. L. Aarts and J. Korst, *Simulated Annealing and Boltzmann machines*, Wiley-Interscience (1988).
2. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ‘‘Hard Real-Time Scheduling: The Deadline Monotonic Approach’’, *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, USA (15-17 May 1991).
3. J. A. Bannister and K. S. Trivedi, ‘‘Task Allocation in Fault-Tolerant Distributed Systems’’, *Acta Informatica* **20**, pp. 261-281 (1983).
4. S. Wayne Bollinger and Scott F. Midkiff, ‘‘Heuristic Technique for Processor and Link Assignment in Multicomputers’’, *IEEE Transactions on Computers* **40**, pp. 325-333 (March 1991).
5. A. Burns, ‘‘Scheduling Hard Real-Time Systems: A Review’’, *Software Engineering Journal* **6**(3), pp. 116-128 (1991).
6. G. Chen and J. Yur, ‘‘A Branch-and-Bound-with-Underestimates Algorithm for the Task Assignment Problem with Precedence Constraint’’, *10th International Conference on Distributed Computing Systems*, pp. 494-501 (1990).
7. Chu, W.W. and Lan, L.M., ‘‘Task allocation and precedence relations for distributed real-time systems’’, *IEEE Transactions on Computers* **A02**, pp. 667-79, IEEE Trans. Comput. (USA) (June 1987).
8. A. Damm, J. Reisinger, W. Schwabl and H. Kopetz, ‘‘The Real-Time Operating System of

MARS'', *ACM Operating Systems Review* **23**(3 (Special Issue)), pp. 141-15 (1989).

9. Houstis, C.E., ''Module allocation of real-time applications to distributed systems'', *IEEE Transactions on Software Engineering* **16**(7), pp. 699-709, IEEE Trans. Softw. Eng. (USA) (July 1990).
10. S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, ''Optimisation by Simulated Annealing'', *Science*(220), pp. 671-680 (1983).
11. P. J. M. Laarhoven and E. H. L. Aarts, *Simulated Annealing: Theory and Applications*, D. Reidel Publishing (1987).
12. J. Lehoczky, L. Sha and Y. Ding, ''The Rate Monotonic Scheduling Algorithm: Exact Characterisation and Average Case Behaviour'', *Proceedings of the Real-Time Systems Symposium* (1989).
13. J. Y. T. Leung and J. Whitehead, ''On The Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks'', *Performance Evaluation*, pp. 237-250 (Vol. 2, Part 4, Dec 1982).
14. C. L. Liu and J. W. Layland, ''Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment'', *Journal of the ACM* **20**(1), pp. 46-61 (1973).
15. C. C. Price and M. A. Salama, ''Scheduling of Precedence-Constrained Tasks on Multiprocessors'', *The Computer Journal* **33**(3), p. 219 (1990).
16. N. Radcliffe and G. Wilson, ''Natural Solutions Give Their Best'', *New Scientist*, pp. 47-50 (14th April 1990).
17. K. Ramamritham, ''Allocation and Scheduling of Complex Periodic Tasks'', *10th International Conference on Distributed Computing Systems*, pp. 108-115 (1990).
18. Pierre Roussel-Ragot and Gerard Dreyfus, ''A Problem Independent Parallel Implementation of Simulated Annealing: Models and Experiments'', *IEEE Transactions on Computer-Aided Design* **9**(8) (August 1990).
19. K. Tindell, ''Allocating Real-Time Tasks (An NP-Hard Problem made Easy)'', YCS 149, Department of Computer Science, University of York (December 1990).
20. V. Varbosa and M. C. Boeres, ''An occam based evaluation of a parallel version of Simulated Annealing'', *Microprocessing and Microprogramming*, pp. 85-92, Euromicro '90 (1990).

APPENDIX 1: AN EXAMPLE RUN

This section describes an example run of a task allocation program giving a good final solution, and indicates how the result was obtained. The quality of the result is discussed, and an illustration of the flexibility and power of the approach is given.

In order to illustrate the algorithm a specific task allocation problem is required. The table below shows an example task set, consisting of 42 tasks, including five paired replicas.

Task	Period	WCET	Memory	Messages	Location
0	60	4	3000	50→1,150→2	0
1	60	4	1500	60→3,70→4,30→5	
2	60	2	1200	20→3	
3	60	2	1700		1
4	60	2	3000	60→6	
5	60	4	3000	80→6	
6	60	6	1100		2
7	35	2	500	40→8	1
8	35	2	700		1
9	35	8	900	90→11	0
10	35	14	2200	250→11	
11	35	4	1000		1
12	14	2	1000	150→13,150→14	2
13	14	2	1500	50→15	
14	14	2	1600	50→15	
15	14	2	1300		3
16	14	2	1100	50→17	3
17	14	2	1000		2
18	35	1	1000	50→19	1
19	35	1	1600		1
20	14	1	1900	40→21	
21	14	2	2000		3
22	14	1	1000	40→23	
23	14	1	2000	40→24	
24	14	1	1000	20→25	
25	14	1	2000	20→26	
26	14	2	7000	20→27,20→28	
27	14	1	1100	50→29	
28	14	1	900	30→29	
29	14	1	500		6
30	14	1	600	50→31	7
31	14	2	800	70→32	
32	14	2	1300		7
33	20	3	1000	50→35	2,3
34	20	2	1000	50→35	0,1
35	20	2	1000	60→36,60→37	
36	20	2	1000		6,7
37	20	2	1000		
38	20	3	1000	50→40	2,3
39	20	2	1000	50→40	0,1
40	20	2	1000	60→41,60→42	
41	20	2	1000		6,7
42	20	2	1000		6,7

The table shows the periods and WCET (worst-case execution time) for each task, along with the

memory required (in bytes) and the messages sent between tasks. The entry "60→41,60→42" indicates messages of 60 bytes sent to tasks 41 and 42. The final column in the table shows any location constraints — task 33 (for example) must be located on either processor 2 or processor 3.

There are 8 processors in the system, with equal execution speeds, and with the following memory capacities:

Processor	Capacity
0	10000
1	10000
2	10000
3	12000
4	7000
5	7000
6	12000
7	10000

The following tasks must avoid each other (they are replicas):

Tasks 33 and 38
Tasks 34 and 39
Tasks 35 and 40
Tasks 36 and 41
Tasks 37 and 42

The first step in applying the algorithm is to choose the start temperature and the weights on the energy components — these can all be chosen by ‘rules of thumb’. The start temperature chosen is 65536, and the weights chosen are:

$k_1 = 15777.3$
$k_2 = 117.4$
$k_3 = 1.0$
$k_4 = 12.4$

It should be pointed out that the values of the weights are fairly arbitrary (the algorithm is not very sensitive to the value of the weights), and were chosen so that hard constraint components return similar values, and that the soft constraint component returns lower values.

Firstly, a random allocation is chosen. The table below shows the allocation, and adopts the following notation: The first row indicates the processor number, and the numbers underneath each processor are the tasks allocated to that processor, listed in priority order. The ‘*’ symbol by a task indicates that the task is unschedulable.


```
  0   1   2   3   4   5   6   7

14* 13* 12* 16*     35* 22* 30*
20* 39* 26* 24*     40* 29  10*
25*  7  27* 31*     37      32*
28*  8  33* 15       4      36*
34* 11  38* 21              41*
 9* 18  17   1              42*
23* 19   6   5               2*
 0*  3

Processor 0: processing utilisation 82.4%, memory utilisation 133.0%
Processor 1: processing utilisation 56.2%, memory utilisation 90.0%
Processor 2: processing utilisation 90.0%, memory utilisation 132.0%
Processor 3: processing utilisation 77.6%, memory utilisation 89.2%
Processor 4: processing utilisation 0.0%, memory utilisation 0.0%
Processor 5: processing utilisation 33.3%, memory utilisation 47.1%
Processor 6: processing utilisation 14.3%, memory utilisation 12.5%
Processor 7: processing utilisation 94.8%, memory utilisation 83.0%
TRT=23.4ms, bus speed=90 bytes/ms, bus utilisation=96.2%
```

The token rotation time based upon the peak load is 23.4 ms (of course, some tasks could execute more than once in this time, increasing the peak load — we ignore this aspect since for a schedulable system TRT must be less than the periods of tasks using the bus). The bus utilisation is 96%.

There are three replica clashes: 33 and 38, 35 and 40, and 36 and 41, giving a replica penalty of 6 (33 ‘sees’ 38 as a clash, and 38 ‘sees’ 33 as a clash, so each pair is counted twice).

The system is unschedulable (due mostly to a TRT of 23.4 ms), and processors 0 and 2 have memory utilisations of approximately 132%. The total energy for this allocation is 447618. The next step in the annealing algorithm is to apply the neighbour function. The neighbour function needs to decide whether to apply a task move, or to swap two tasks (for this example we set the probability of using swaps to 0.15). Let’s assume that a move is chosen. The neighbour function randomly picks a task: task 22. This task can be placed on any processor, and one is randomly chosen: processor 1. The new allocation looks like this:

```
  0   1   2   3   4   5   6   7

14* 13* 12* 16*     35* 29  30*
20* 22* 26* 24*     40*     10*
25* 39* 27* 31*     37      32*
28*  7  33* 15       4      36*
34*  8  38* 21              41*
 9* 11  17   1              42*
23* 18   6   5               2*
 0* 19
     3
```

```
Processor 0: processing utilisation 82.4%, memory utilisation 133.0%
Processor 1: processing utilisation 63.3%, memory utilisation 100.0%
Processor 2: processing utilisation 90.0%, memory utilisation 132.0%
Processor 3: processing utilisation 77.6%, memory utilisation 89.2%
Processor 4: processing utilisation 0.0%, memory utilisation 0.0%
Processor 5: processing utilisation 33.3%, memory utilisation 47.1%
Processor 6: processing utilisation 7.1%, memory utilisation 4.2%
Processor 7: processing utilisation 94.8%, memory utilisation 83.0%
TRT=23.4ms, bus speed=90 bytes/ms, bus utilisation=96.2%
```

Task 22 has increased the load on processor 1 which makes tasks of lower priority less schedulable (in particular, the deadline of task 39 is missed by a larger amount). Task 22 still needs to send messages on the bus, so there is no change in the peak load and hence TRT. The total energy for this allocation is 450620 (leading to an increase in energy of 3002. At the current temperature of 65536 the expression $e^{\frac{E_{old}-E_{new}}{T}}$ evaluates to 0.96, giving a 0.96 chance of the jump being accepted. Assume that the jump is taken (no further action is required). Now assume the neighbour function decides to swap two tasks, and picks tasks 10 and 23. The following allocation is produced:


```
  0   1   2   3   4   5   6   7

14* 13* 12* 16*     35* 29  23*
20* 22* 26* 24*     40*     30*
25* 39* 27* 31*     37      32
28*  7  33* 15       4      36
34*  8  38* 21              41
 9* 11  17   1              42
10* 18   6   5               2
 0* 19
     3
```

```
Processor 0: processing utilisation 115.2%, memory utilisation 135.0%
Processor 1: processing utilisation 63.3%, memory utilisation 100.0%
Processor 2: processing utilisation 90.0%, memory utilisation 132.0%
Processor 3: processing utilisation 77.6%, memory utilisation 89.2%
Processor 4: processing utilisation 0.0%, memory utilisation 0.0%
Processor 5: processing utilisation 33.3%, memory utilisation 47.1%
Processor 6: processing utilisation 7.1%, memory utilisation 4.2%
Processor 7: processing utilisation 61.9%, memory utilisation 81.0%
TRT=24.2ms, bus speed=90 bytes/ms, bus utilisation=101.2%
```

The energy for this point is 490606, resulting in an increase in energy of 39986, and the chance of taking this jump is 0.54.

A number of these jumps are proposed at each temperature. The temperature is reduced when a fixed number of downward jumps have been made, or when an upper limit on the number of proposed jumps is reached. In this example, the temperature is reduced after 50 jumps have been made, or a total of 1000 jumps have been proposed. The following shows the pattern of jumps for the current temperature:

```
^=vv===v=^==^^==-====v=^vv===v=v^==v^^==---^====v==v=vv====-=^==
--=-^====^====^v==========v===^v==========^^====^=v===v-vv===^v
=v-v-==^v===vvv====^-v^=^==v====v^=-^=v===v=^=====^=v====v=v====
vv=v==v-^==^-===v==v^^v====v===^^=-==v=-=^-v^-====^==v======^===
====v^^^==^=====-===-==vv=-==^=v
```

The ‘v’ symbol indicates that a downward (energy reducing) jump was taken. The ‘^’ symbol indicates that an upward jump was taken. The ‘=’ symbol indicates that a jump to an allocation with an equal energy was taken. The ‘—’ symbol indicates that a jump was proposed but refused.

As can be seen from above, few jumps are refused at this temperature, and the algorithm examines many allocations. As the temperature is lowered this ability is reduced. The value of α in this example is 0.95, and the temperature reduces slowly. The following shows the pattern of jumps for the temperature 9332:

```
^======-v-======-==-=^-==-=v=--=====-^====-=-=-===-=v=^==-vv==-=
v==^==^=========^=====--^==v====v-=^==v-==--^v-===^-==-=v===^^==
=v^=====^====v==-=vv-===-=v===--===^==-==-v==-v==-===-^=v^-===^=
--=^===^===v====-======v^-=^==v=-==^=^===-=-==v-v-===-==^-===-v
^=v=-====-==-=--=-^==-=-==vv=v-=====-==v=--=v-=-=-=--=^--===-==
=-===^-=^=^--=^=-v==-===v=v-=^=^v-==-=--^=---===-====v^=====-===
--====v=v====v-==-==^====v==vv-==-^-^v-===---====-==v^--===^=-==
=--^^=-=-===--^===-=-=========-^=====v^-v=^=-====-=-==-=-^v===v=
=v^=====-=-=v
```

As can be seen, many more proposed jumps are being rejected, and a much larger number of trials are required before 50 downward jumps are made. A typical allocation at this temperature is given below:

```
   0   1   2   3   4   5   6   7

 39  34  13* 16* 24  22  29  20
  9   7  14* 26*  4  28* 40  25*
  1   8  27* 38   5  35  41  23
  2  10  33* 15      37  42  30
  0  11  12  21              31
     18  17                  32
     19   6                  36
      3
```

```
Processor 0: processing utilisation 49.5%, memory utilisation 76.0%
Processor 1: processing utilisation 81.9%, memory utilisation 97.0%
Processor 2: processing utilisation 89.3%, memory utilisation 83.0%
Processor 3: processing utilisation 72.1%, memory utilisation 103.3%
Processor 4: processing utilisation 17.1%, memory utilisation 61.4%
Processor 5: processing utilisation 34.3%, memory utilisation 55.7%
Processor 6: processing utilisation 37.1%, memory utilisation 29.2%
Processor 7: processing utilisation 67.1%, memory utilisation 96.0%
TRT=12.9ms, bus speed=90 bytes/ms, bus utilisation=51.6%
```

The energy for this allocation is 25435. As can be seen, only one processor now has a memory utilisation of greater than 100%, and more tasks are schedulable. Also, replicas are placed on different processors.

At a temperature of 1717 few jumps are being taken:

```
==^====---=-=-====-=^v--=====-==-==--==v=-==v=-===-=-====^==-==-
-=^-==-=-===^vv=======^-=-====-==---==-==-==v--==^=-====v=--=-==
=====-====--=-=-===--=-==-=--=---==-=-=-=v=--===-==-=--==--=====
=--=-=--==---===-====-==-======-====--==-===-=-=----===^-=--=^v==
=-=-===---===-==--=-===-==-=v=-=.....
```

At a final temperature of 28 no upward or downward jumps have been made for 4000 proposed jumps and the final allocation, with an energy of 3403, is shown below:


```
   0   1   2   3   4   5   6   7

 35  39  13  16  25      26  30
 34   7  14  38  22      27  31
 37   8  33  15  23      28  32
  9  10  12  20  24      29  40
  1  11  17  21          36  41
  2  18   6   5              42
  4  19
  0   3
```

```
Processor 0: processing utilisation 72.9%, memory utilisation 99.0%
Processor 1: processing utilisation 81.9%, memory utilisation 97.0%
Processor 2: processing utilisation 82.1%, memory utilisation 72.0%
Processor 3: processing utilisation 71.7%, memory utilisation 85.8%
Processor 4: processing utilisation 28.6%, memory utilisation 85.7%
Processor 5: processing utilisation 0.0%, memory utilisation 0.0%
Processor 6: processing utilisation 45.7%, memory utilisation 87.5%
Processor 7: processing utilisation 65.7%, memory utilisation 57.0%
TRT=8.7ms, bus speed=90 bytes/ms, bus utilisation=29.4%
```

As can be seen, all the hard constraints have been met, and the bus utilisation is low. Interestingly, the solution has a structure to it: communicating subsystems are clustered together to minimise bus traffic and to increase schedulability. For example, tasks 22, 23, 24, 25, 27, 27, 28, and 29 are all related. Tasks 26, 27, 28, and 29 are clustered on processor 6 (task 29 is forced to reside on processor 6), and tasks 22, 23, 24, and 25 are clustered on processor 4. Only one message in the subsystem is sent on the bus: task 25 sends a message to task 26; all other messages are local. The whole subsystem would not fit on processor 6, and the best place to split the subsystem is between tasks 25 and 26 (other splits would require more communication on the bus).

As has been mentioned before, the annealing algorithm is very flexible and changes can be accommodated easily. A small change to the annealing program (just a few lines) was made to change the soft constraint to balance the processor utilisation between the used processors instead of minismising bus traffic (it could be argued that a ‘balanced’ allocation gives better tolerance to execution time over-runs). At a bus speed of 90 bytes/msec many of the well-balanced allocations result in infeasible solutions (since a high bus utilisation gives a high TRT, resulting in less schedulable allocations). To illustrate the potential of the algorithm the program was run on a problem with a faster bus. The following allocation was produced:

```
   0   1   2   3   4   5   6   7

22  24  12  16  40  13  26  25
34  28  38  33  10  14  31  30
 9  39  17  15   5  20  27  23
 0   7   4  21      35  29  32
 1   8   6          37  41  36
    11                   2  42
    18
    19
     3
```

```
Processor 0: processing utilisation 53.3%, memory utilisation 74.0%
Processor 1: processing utilisation 56.2%, memory utilisation 94.0%
Processor 2: processing utilisation 56.9%, memory utilisation 44.0%
Processor 3: processing utilisation 57.9%, memory utilisation 45.0%
Processor 4: processing utilisation 56.7%, memory utilisation 88.6%
Processor 5: processing utilisation 55.7%, memory utilisation 100.0%
Processor 6: processing utilisation 56.2%, memory utilisation 96.7%
Processor 7: processing utilisation 55.7%, memory utilisation 79.0%
TRT=7.9ms, bus speed=250 bytes/ms, bus utilisation=91.3%
```

This allocation is feasible — all tasks are schedulable, all processors have memory utilisations ≤ 100%, and replicas are on different processors. The average processor utilisation is 56%.