A Very High Speed Architecture for Simulated Annealing

David Abramson
Commonwealth Scientific and Industrial Research Organisation

Simulated annealing algorithms for scheduling are often extremely slow, even on workstations or supercomputers. Special-purpose architectures implemented on low-cost PC boards can speed annealing dramatically.

Scheduling is a common requirement in many organizations. The traditional solution — manual scheduling by experts — is time-consuming and expensive. Computer-based intelligent systems have the potential to assist in scheduling operations, but few good robust optimization algorithms exist that can be applied in many different problem domains. Most real-world problems are highly nonlinear, and many problems have an exponential solution time complexity. Thus, simple linear operations-research algorithms or state-space searches are often ineffective.

Kirkpatrik, Gelatt, and Vecchi first suggested simulated annealing as an optimization technique for complex nonlinear optimization problems.[1] This technique has been applied to many different problems with remarkable success.[2] Importantly, annealing prevents searches from becoming trapped in local minima by allowing them to occur in areas of higher energy, according to the substance's temperature. Unlike hand-tailored heuristics, most annealing schedules contain no specialist knowledge about how to solve a particular problem. Thus, the technique can be applied to many different problem domains without substantial changes to the core algorithm.

Here, I present a class of scheduling problems that can be modeled by using very simple cost measures: *counting costs* and *distance-measure costs*. These measures can be easily incorporated into a simulated annealing algorithm, providing a robust system for solving such scheduling problems.

Annealing's major disadvantage is that it can be extremely slow. Solving a complex system may require a very slow cooling rate and thus a solution that uses much processor time. Because of this, simulated annealing has not been widely accepted as an optimization algorithm for low-cost intelligent planning and scheduling systems, and much discussion about using parallel computers to accelerate annealing techniques has occurred. Some parallel systems, in fact, achieve close to ideal speedup on small processor arrays.[3-8]

I describe an alternative technique to speed annealing: a special-purpose computer architecture that supports the simulated annealing of scheduling problems based on counting costs and distance-measure costs. I illustrate an application with

0018-9162/92/0500-0027$03.00 © 1992 IEEE

Theory of simulated annealing

Simulated annealing is a Monte Carlo technique for finding solutions to optimization problems. (Van Laarhoven and Aarts review the theory and practice.[1]) The technique simulates the cooling of a collection of hot vibrating atoms. When the atoms are at a high temperature, they are free to move around and tend to move with random displacements. However, as the mass cools, interparticle bonds force the atoms together. When the mass is cool, no movement is possible and the configuration is frozen. If the mass is cooled quickly, the chance of obtaining a low-cost solution is lower than if it is cooled slowly (or annealed).

At any given temperature, a new configuration of atoms is accepted if the system energy is lowered. However, if the energy is higher, the configuration is accepted only if the probability of such an increase is lower than that expected at the given temperature. This probability is given by $P(\Delta E) = e^{-\Delta E/KT}$, where K is Boltzmann's constant. These acceptance criteria are based on the physics of annealing.[1]

Many optimization problems can be considered as a number of *objects* that must be scheduled to minimize an objective function. Objects replace vibrating atoms, and the value of the objective function replaces the system energy. An initial schedule is created by randomly scheduling the objects, and an initial cost (c_0) and temperature (T_0) are computed. Subsequent permutations are created by randomly choosing a number of objects, rearranging them, and computing a change in cost (Δc). If $\Delta c \le 0$, the change is accepted. However, if $\Delta c > 0$, the change is accepted with probability $P(\Delta c) = e^{-\Delta c/T}$.

If the probability is greater than a randomly selected value in the range 0 to 1, the change is accepted. The most common technique for choosing a random number is to use a pseudorandom uniformly distributed variable on the unit interval. After a number of successful permutations, the temperature is decreased by a cooling rate R such that $T_n = T_{n-1} * R$, where $0 \le R < 1$, and T is a real number. This scheme is called a geometric cooling schedule.[1]

Reference

1. P.J.M. van Laarhoven and E.H.L. Aarts, *Simulated Annealing: Theory and Applications*, Kluwer Academic, Boston, 1987.

a case study based on the school timetable scheduling problem. A high-speed architecture that I constructed runs the algorithm on an IBM PC between 30 and 80 times faster than on a Cray Y-MP, at a fraction of the cost. Computer Techniques P/L has incorporated the architecture into an intelligent system that assists school administrators in planning class timetables.

Counting and distance-measure costs

Scheduling problems usually involve mapping a number of objects into some region of solution space to minimize a global cost measure. For example, scheduling flight crews to aircraft requires mapping crew members onto planes so that

- any crew member is on only one flight at a time,
- crew members can connect to required flights, and
- many other real-world constraints are met.

To measure the quality of a solution, a cost function is devised that measures the seriousness of broken constraints. In many problems, this cost can be composed of two basic measures. First, solution quality concerns the number of times a certain object appears in a particular part of the solution space. For example, a crew member can be in only one place at a time. To measure this type of cost requires only a count of occurrences of that object in each part of solution space. Thus, this cost measure is called a counting cost. Second, solution quality concerns

- the relationship between one object in solution space and another object (or objects), or
- the object's absolute position in solution space.

For example, an aircraft crew member may be unhappy on a given flight if his or her spouse is on another flight, or a crew member may not like early morning flights. Measurement of this type of cost requires the location of other objects in the solution space, together with a cost relationship between points in space. Thus, this cost measure is called a distance-measure cost, because it concerns the distance between objects in space.

Counting and distance-measure costs have an important attribute: We can compute the change in global cost function after a single permutation of the schedule by computing an incremental cost for each cost measure. (The annealing hardware later described uses this feature extensively.)

School timetable problem

The problem of creating a valid timetable involves scheduling classes, teachers, and rooms to the periods of the week so that no teacher, class, or room is used more than once per period.[9,10] A particular combination of a teacher, a subject, a room, and a class is called a *tuple*, and a tuple may be required more than once per week. The problem is further complicated by consecutive periods requests, which demand that certain tuples are mapped onto adjacent periods, but are not separated by a lunch or recess break. Figure 1 shows a sample timetable, with the counting and distance-measure costs illustrated.

This short introduction shows that the school timetable problem exhibits the two basic cost measures described in the previous section: The number of occurrences of any class, teacher, or room in any period must be constrained, and the placement of some tuples relative to other tuples is important.

The problem can be described using a slightly more formal notation. Given a number of tuples in a schedule,

$$TT = \{(T1{:}c_1, t_1, r_1, p_1), (T2{:}c_2, t_2, r_2, p_2), \ldots\}$$

find a mapping such that

$\mathbf{count}(c_c, p) < 2, \{0 \le p \le P\} \{0 \le c \le C\},$
$\mathbf{count}(t_t, p) < 2, \{0 \le p \le P\} \{0 \le t \le T\},$
$\mathbf{count}(r_r, p) < 2, \{0 \le p \le P\} \{0 \le r \le R\}$

and

$\mathbf{adjacent}(x, y) = \mathbf{should_be_adjacent}(x, y), \{1 \le x \le N, 1 \le y \le N\}$

Given that the class, teacher, and room fields of the tuples are fixed, this amounts to finding a period value in each tuple. A global cost that measures the number of broken constraints is defined in Figure 2, where T_n is the tuple name, t_n is teacher number n, C is the number of classes, c_n is class number n, r_n is room number n, T is the number of teachers, p_n is the period, P is the number of periods, N is the number of tuples, C is the number of classes, and R is the number of rooms. Then

adjacent(x, y) returns 1 if tuple x and tuple y are in adjacent periods and y <> 0 else 0
should_be_adjacent(x, y) returns 1 if tuple x is required to be adjacent to tuple y else 0
neighbor(x) returns the number of the tuple that should be adjacent to x; 0 if no tuple
count(x, p) = the number of times x appears in period p
bound(x) = 0 if $x < 2$ else x

A perfect schedule will have a cost of zero. This analysis shows that **count** is a counting cost, and **adjacent** is a distance-measure cost. Further, if we change the schedule TT by altering the period p of an arbitrary tuple (T_n:c_n, t_n, r_n, p) to form a new tuple ($T_{n'}$:c_n, t_n, r_n, p'), we can compute the change in cost incrementally as shown in Figure 3, where

saving(x, p) = 1 if **count**(x, p) > 1 else 0 and
insertion(x, p) = 1 if **count**(x, p) > 0 else 0.

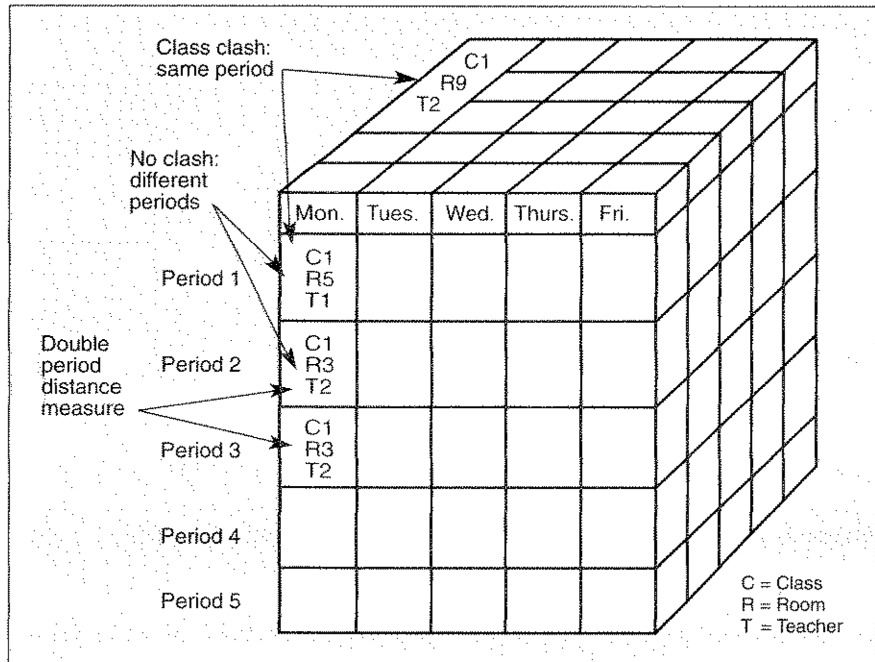

Figure 1. A sample timetable with counting and distance-measure costs.

$$\text{Global_cost} = \sum \mathbf{bound}(\mathbf{count}(c_c, p)) \text{ for all } \{0 \le p \le P\} \{0 \le c \le C\} + \sum \mathbf{bound}(\mathbf{count}(t_t, p)) \text{ for all } \{0 \le p \le P\} \{0 \le t \le T\} + \sum \mathbf{bound}(\mathbf{count}(r_r, p)) \text{ for all } \{0 \le p \le P\} \{0 \le r \le R\} + \sum |\mathbf{adjacent}(x, y) - \mathbf{should_be_adjacent}(x, y)| \text{ for all } \{1 \le x \le N, 1 \le y \le N\}$$

Figure 2. Definition of a global cost that measures the number of broken constraints.

$$\Delta\text{Cost} = \mathbf{saving}(c_n, p) + \mathbf{saving}(t_n, p) + \mathbf{saving}(r_n, p) - \mathbf{insertion}(c_n, p') - \mathbf{insertion}(t_n, p') - \mathbf{insertion}(r_n, p') + \mathbf{adjacent}(n', \mathbf{neighbor}(n')) - \mathbf{adjacent}((n, \mathbf{neighbor}(n))$$

Figure 3. Incremental computation of the change in cost.

Gate allocation

Another scheduling problem common in transportation is the mapping of transport units to loading and unloading bays. An example is the assignment of various aircraft to airport terminal gates. The problem is complicated because flights arrive and depart at arbitrary preset times. An aircraft can be held at only one gate at any point in time, and some aircraft can be placed only at specific gates because of physical limits. We can model this type of constraint with a counting cost because we need counts of which aircraft are mapped onto which bays. If an aircraft cannot be housed at a certain gate, the count can be initialized to "pretend" that an aircraft is already present, and thus no more aircraft can be accommodated. Inter-aircraft constraints can prohibit certain plane types from being parked on adjacent bays and can also request that certain linking flights are placed close to each other to facilitate passen-

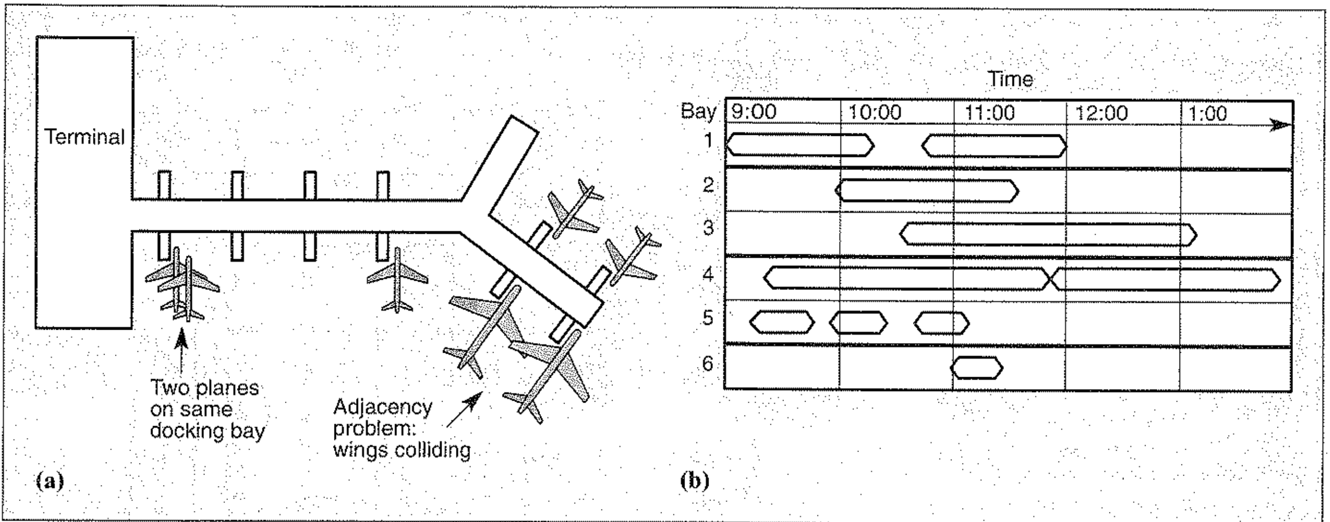

Figure 4. The gate-allocation problem (a) and sample Gantt chart (b).

ger movement. This latter type of constraint can be modeled with a distance-measure cost. Figure 4a shows a sample airport with the constraints described, and Figure 4b shows a sample Gantt chart of the optimized mapping.

We can phrase the problem using the formal description in the previous example. Given a number of flights f, the problem is composed of tuples that arrive and depart at prefixed times a_n and d_n, to which we must allocate gates g_n at the terminal:

$$f = \{(T1{:}a_1, d_1, g_1), (T2{:}a_2, d_2, g_2), \ldots\}$$

Each tuple can be decomposed into a number of smaller tuples containing a time-interval number between the arrival and the departure times, each of which is mapped into the same gate:

$$f = \{(T1.1{:}t_1, g_1), (T1.2{:}t_2, g_1), (T1.3{:}t_3, g_1), (T2.1{:}t_4, g_2), (T2.2{:}t_5, g_2), (T2.3{:}t_6, g_2), \ldots N \text{ tuples}\}$$

Thus, $t_1 \ldots t_3$ is a discrete sequence of times between a_1 and d_1, and $t_4 \ldots t_6$ is a discrete sequence of times between a_2 and d_2.

We can use distance-measure costs to guarantee that the smaller tuples are grouped on the same gate, as well as to guarantee that intraflight constraints are not broken. We find a mapping such that

$\mathbf{count}(t, g) < 2$, $\{0 \le t \le T\}$ $\{0 \le g \le G\}$
$\mathbf{notadjacent}(x, y) = \mathbf{should_not_be_adjacent}(x, y)$, $\{1 \le x \le N, 1 \le y \le N\}$
$\mathbf{samegate}(x, y) = \mathbf{should_be_samegate}(x, y)$, $\{1 \le x \le N, 1 \le y \le N\}$

As in the last example, we can define a global cost that measures the number of gate time slots that contain more than one aircraft, as well as the number of broken adjacency constraints as shown in Figure 5, where a_n is the arrival time of flight n, g_n is the gate number used for flight n, d_n is the departure time of flight n, N is the total number of tuples, and G is the number of gates. A perfect schedule will have a cost of zero.

$$\text{Global_cost} = \sum \mathbf{bound}(\mathbf{count}(t, g)) + \sum|\mathbf{notadjacent}(x, y) - \mathbf{should_not_be_adjacent}(x, y)| + \sum|\mathbf{samegate}(x, y) - \mathbf{should_be_same_gate}(x, y)|$$
$$\text{for all } \{1 \le x \le N, 1 \le y \le N, 0 \le t \le T, 0 \le g \le G\}$$

Figure 5. Definition of a global cost that measures the number of gate time slots.

This analysis shows that **count** is a counting cost, and **notadjacent** and **samegate** are distance-measure costs. Further, if we change the schedule f by altering the gate g of an arbitrary tuple $(T_n{:}\, t_n, g)$ to form a new tuple $(T_{n'}{:}t_n, g')$, the change in cost can be computed incrementally as shown in Figure 6, where

$\mathbf{saving}(t, g) = 1$ if $\mathbf{count}(t, g) > 1$ else 0
$\mathbf{insertion}(t, g) = 1$ if $\mathbf{count}(t, g) > 0$ else 0
$\mathbf{neighbors1}(t)$ returns the identification numbers of the tuples that have adjacency constraints with tuple t.
$\mathbf{neighbors2}(t)$ returns the identification numbers of the tuples in the next and previous time slots for the same gate.

$$\Delta\text{Cost} = \mathbf{saving}(t_n, g) - \mathbf{insertion}(t_n, g') + \mathbf{notadjacent}(n', \mathbf{neighbors1}(n')) - \mathbf{notadjacent}((n, \mathbf{neighbors1}(n)) + \mathbf{samegate}(n', \mathbf{neighbors2}(n')) - \mathbf{samegate}((n, \mathbf{neighbors2}(n))$$

Figure 6. Second incremental computation of the change in cost.

Machine-job allocation

Another classic scheduling problem is machine-job allocation. Many algorithms solve this problem,[11] but it is useful to consider the application of counting and distance-measure costs in this domain. The scheduling task consists of mapping machines to jobs so that each machine performs only one task at a time and the most appropriate machine performs each job. The latter is measured by an arbitrary cost function, which should be minimized in the final allocation.

We can map the problem onto the counting and distance-measure costs. Because any machine can perform only one job at a time, a counting cost is required to limit the number of jobs to any machine. We use a distance-measure cost to measure the cost of any particular assignment. In the previous examples, the distance-measure cost had a value of 0 or 1. In this scheduling problem, we want a higher resolution, so we denote the cost by a small integer (for example, an integer between 0 and 15). The aim of the schedule is to minimize the total assignment cost.

Optimization with simulated annealing

In the general form, a scheduling problem can be viewed as a collection of tuples with each tuple composed of a number of fixed fields and a number m of modifiable fields. Thus, any individual tuple has the form $T(x, y, z, \ldots, P_1, P_2, \ldots, P_m)$. The optimization task is to find values for all modifiable fields that satisfy the counting and distance-measure constraints. Annealing can be applied for optimization in all three problems cited in the previous sections.

Applying simulated annealing. Figure 7 summarizes the basic simulated annealing algorithm. At each temperature, the algorithm applies a number of trials until the substance achieves thermal equilibrium.[2] The simplest scheme is to iterate for some fixed, predetermined number of trials at each temperature. If the tuples have a number of modifiable fields, only one is chosen at any iteration. A new tuple field value is chosen at random, and the change in cost is computed. If the change is accepted, the tuple is altered to contain the new modifiable field value; otherwise, the tuple is left unaltered. The algorithm uses the incremental cost-changing formulas computed in the previous sections.

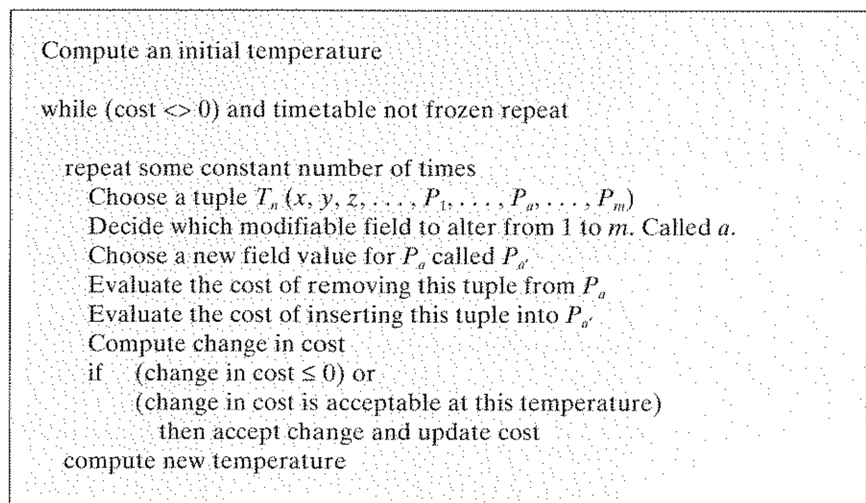

Figure 7. Sequential simulated annealing algorithm.

A parallel algorithm. Using a parallel algorithm[3-8] rather than the sequential one shown in Figure 7 speeds simulated annealing. The serial algorithm performs each permutation of the tuples sequentially and either accepts or rejects the new configuration. It does not generate a new configuration until the previous one is completed. However, a parallel algorithm can perform multiple permutations concurrently if each permutation is independent of the others. Elsewhere, I present results from a parallel execution of the program.[9] On average, the speedup was about 50 percent of optimal for a small number of processors (fewer than 10). Although the parallel algorithm shows reasonable speedup, it still falls short of the times required for simulated annealing in an intelligent system. For example, solving real school problems was taking days of processor time on conventional workstations. In the next section, I present a different approach to accelerating the algorithm.

An annealing machine

When applied to the scheduling problems described here, a software implementation of the simulated annealing algorithm is slow for two reasons:

• The code is executed sequentially. Each time the algorithm computes a

Performance of special-purpose architectures

Research projects around the world are looking for cheaper ways to provide high-speed computing systems, either for the general-purpose computing market or for special-purpose niche applications. For example, in the general-purpose computing market a plethora of parallel computers is now available.

High-performance special-purpose architectures are also an active area of research. Such architectures show promise for solving computationally expensive applications without expensive supercomputers, and range from specially designed computer architectures to multiprocessors specifically configured to match a given problem's structure. Examples include architectures for solving finite-element computations, digital signal processing multiprocessors, artificial intelligence processors, and encryption architectures. Technical advances in areas like workstation bus standardization and field-programmable gate arrays make it possible to add application-specific processors to a range of machines for further improved performance.

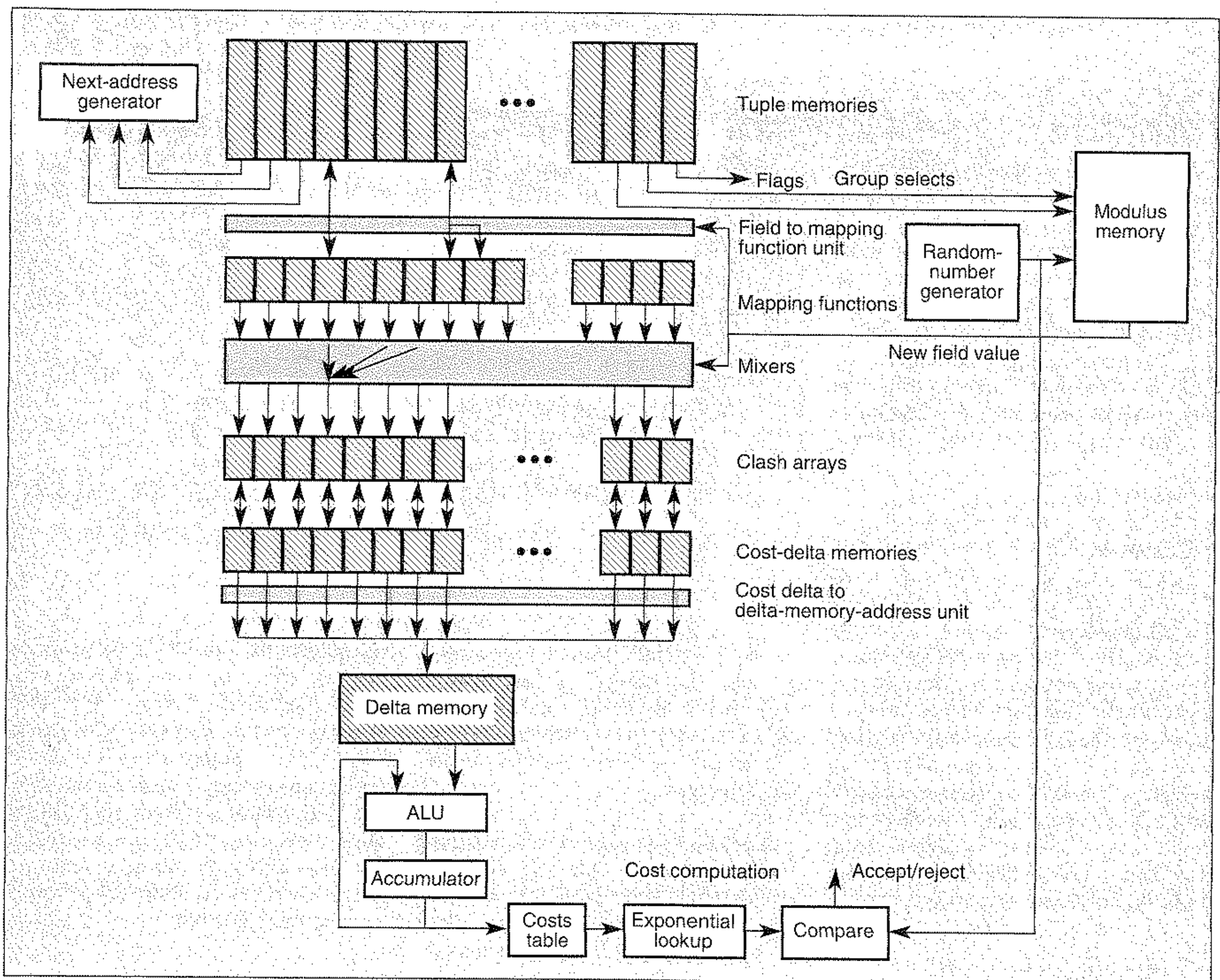

Figure 8. Schematic diagram of an annealing machine.

change in cost, it must evaluate the various saving and insertion costs one after another, even though they can be computed concurrently.

• The data and control structures do not ideally match a von Neumann architecture. For example, to read a count value, the algorithm must index into a two-dimensional data structure, which requires many instructions.

As shown in the previous section, a parallel implementation of the algorithm achieves only a small speedup, at considerable expense. The aims of a special-purpose annealing machine are to use the low-level concurrency inherent in the computation and map the data structures directly onto the machine architecture. Achieving both aims has a massive effect on the algorithm's execution speed.

Figure 8 shows a schematic diagram for a special-purpose scheduling machine. The machine is composed of a number of sections:

- a next-address generator,
- tuple memories,
- a field to mapping-function unit,
- mapping functions,
- mixers,
- clash arrays,
- cost-delta memories,
- a cost-delta to delta-memory-address unit,
- a delta memory,
- an arithmetic logic unit and accumulator,
- a random-number generator, and
- a modulus memory.

The tuples are stored in the tuple memories, in which they are arranged as sets of very long data words. The simulated annealing algorithm calls for the random selection of a tuple. However, it can read and process the tuples sequentially if the random number used to calculate new field values is statistically independent of the tuple address. Thus, the tuple address is produced by a sequential counter, which can be loaded from a link chain field in the tuple. Since only some field values can be modified, only some memory fields require update hardware.

The simulated annealing algorithm uses the field values to index the counter tables and the distance-measure tables, which are denoted as clash arrays in the diagram. Two clash-array memories are dedicated to each of the counting and distance-measure costs. Using two memories allows simultaneous access to the counting or distance-measure cost for the current modifiable field value (for example, the current period number or gate number) and to the stored values for the new field value. Because the computations are independent, they can be evaluated concurrently, thus speeding up the new cost computation. The two memories normally have identical

contents, except for some intermediate stages of the cost evaluation (described later).

A three-way mapping process extracts the addresses for the clash arrays. First, the process splits the field values to index various mapping functions. The outputs of the mapping functions are then combined to form clash-array addresses. Although this appears complicated, the scheme allows any arbitrary combination and modification of field values before the clash arrays are accessed. For example, if the tuple contains fixed fields for class, teacher, and room number, but period number is a modifiable field, then the machine can index the clash arrays using various combinations of these basic fields. Thus, one counting cost may require the combination of some function of class number that computes the year-level from the class number, with some function of room number that computes the section of the school where the room is located. For example, the combined counting cost field value may allow a constraint on the number of times a certain year level uses the northern section of the school.

The output of each clash array is a count value, which must be updated to reflect the change in field value. For example, if there are 13 occurrences of aircraft 1 on gate 3 at time period 10, and we change the gate from 3 to 5, the algorithm must decrement the count value of 13 and increment the corresponding count value for gate 5. The cost-delta hardware manages this function. The hardware also emits an incremental change in cost value according to the current cost. Thus, it can update all the counting costs and concurrently produce the change of cost values.

We can also use clash arrays to implement the distance-measure costs. In the simplest case, we associate each tuple with *one* neighbor. If we use the extra fields in the tuple to hold the modifiable fields of the neighbor as well as the tuple itself, the clash arrays can operate as predefined arbitrary functions that compute the change in cost caused by moving the tuple in solution space. For example, if a tuple contains class number and period number fields, adding the period value of the tuple's neighbor lets an arbitrary function compute the cost of moving the tuple from one period to another. Of course, the neighbor tuple must also contain its own period value as well as its neighbor's to allow symmetrical calculations to be performed. Further, linkage field values are required for a period value to be updated in the owning tuple as well as the neighbor tuple when the value is changed. (The specific implementation of the school timetable annealing hardware described later uses a slightly different representation to simplify the update operations.)

If the problem formation uses the distance measure to compute the cost of an absolute placement in solution space, the hardware feeds the clash array the appropriate tuple-modifiable field value as well as the new field value, and computes a change in cost value. Thus, the clash arrays can implement the counting costs as well as relative and absolute distance-measure costs.

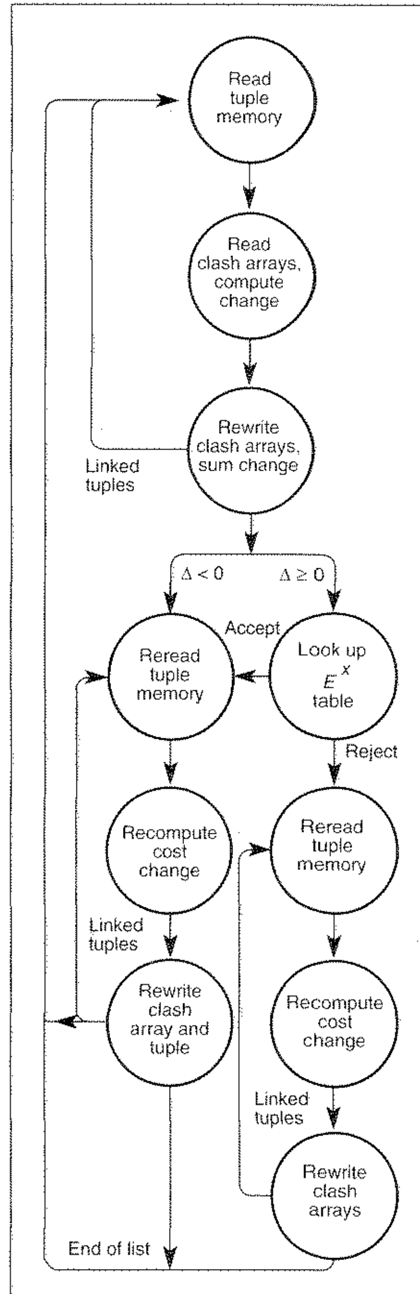

Figure 9. State-transition chart for control of the annealing machine.

The outputs of the cost deltas produce insertion costs and removal savings, which must then be summed together to form a total change in cost. Because we want to scale the various cost components before addition to reflect their importance, we use a delta table memory in preference to a fixed adder. We can preprogram into the table arbitrary scaling constants for very fast evaluation of the summed cost change. The new cost value is loaded into an accumulator via an arithmetic logic unit. Once the value is in the accumulator, the control hardware can determine whether the change in cost is acceptable at the current temperature.

The addition of an ALU circuit allows the hardware to link tuples into one basic unit and evaluate the combined cost. Linked tuples can be moved at the same time rather than individually. This feature can reduce the number of distance-measure constraints significantly. For example, in the gate-scheduling problem, using a **samegate** function is not necessary to guarantee that the time sections of an aircraft are held on the same gate. Rather, the time sections can be linked and moved in one unit. The hardware can evaluate a combined cost for the total move by reading each tuple from the linked unit and adding all cost changes.

A random-number generator such as a linear-feedback shift register produces new field values. A lookup table called a modulus table translates an arbitrary random number into a legal modifiable field value. This table stores the precomputed values of the random number **mod** the maximum field value. Since each modifiable field value has a different maximum value, different tables are stored for each modifiable field. Group select values from the tuple are combined with the random number to form a modulus memory address. The hardware uses the new field value in the clash-array address formation, but must also rewrite it to the tuple memories if the change is accepted.

The finite-state machine shown in Figure 9 controls the annealing algorithm. The program begins by reading

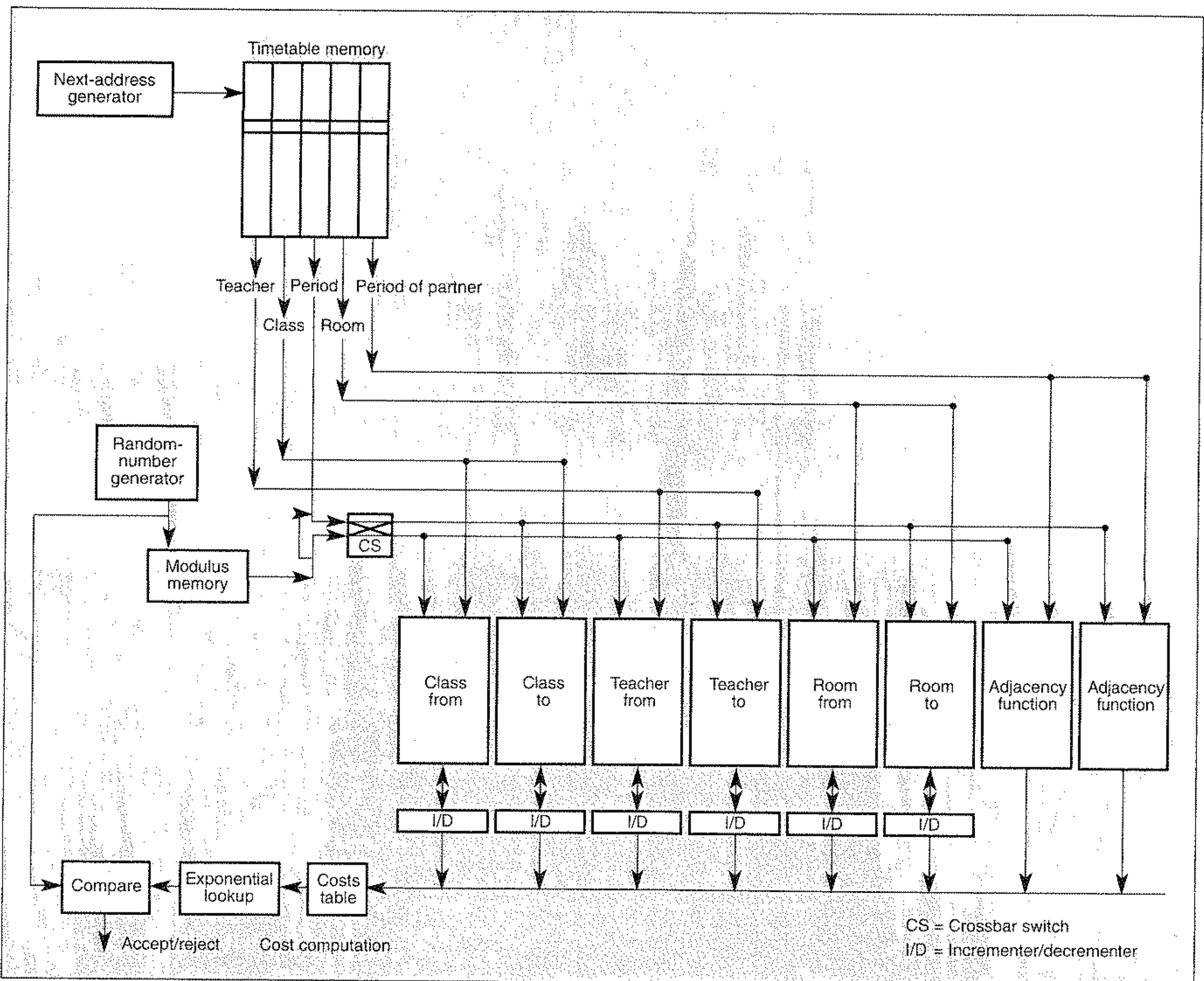

Figure 10. Schematic diagram of a timetable scheduling machine.

the tuple memory fields, which are then passed through the various mapping functions and mixers. The contents of the clash arrays are read, and a change in cost computed. The clash-array values are updated to reflect the changes. This operation leaves the two memories assigned to each counting cost inconsistent. Later in the cycle, the memories are made equivalent again.

If a tuple chain exists, the machine iterates until it computes a total cost change. If the cost change is negative, then the accept logic is entered. However, if the cost is greater than or equal to zero, the negative exponential function must be evaluated to determine whether the change will be accepted. To avoid computation of a complex function, the control software initializes a table with the values of e^x. By comparing a random number with the output of the e^x function, the hardware can determine whether the change is accepted. If the change is rejected, the back-off logic is entered, and the clash arrays are restored to their former values. If the accept logic is entered, the two clash arrays are made consistent, and the modifiable field of the tuple memory is updated. Finally, the global cost is updated by the changed amount.

Case study

The general machine described in the previous section can execute the simulated annealing algorithm with the data structures required to solve the three scheduling problems cited here. In this section, I describe a more specific implementation of the architecture tailored to solve the school timetable problem.

An intelligent assistant. To use the intelligent assistant for creating timetables, school staff members enter their requirements in a special data-description language,[5] which is compiled and validated before being loaded into the system. The data structures are mapped into the tuple representation and loaded onto the annealing accelerator for processing. After the accelerator has terminated, the system uploads the data into the analysis program for printing

or modification. Users can generate a number of reports. The machine can enforce the following constraints:

- class, teacher, and room usage limited to once per period, and
- tuples linked via an arbitrary distance measure, which can be configured to allow multiple periods and any predefined function.

An architecture to solve the problem. The specific architecture implemented for the timetable problem is much simpler than the general structure described earlier, as Figure 10 shows. Direct hardwired connections plus a crossbar switch replace the general mapping functions between the tuple memories and the clash arrays. Circuits that directly implement the insertion and saving functions replace the general cost-delta functions, as well as act as incrementers and decrementers for the clash-array values. The cost values are all 1-bit values (an insertion cost or removal saving is either 0 or 1). These are combined in a fixed way to form the address of the cost table, which is preloaded with a scaled cost function. Two adjacency memories implement the distance measure. One memory computes the cost of the current period placement relative to the partner tuple, and the other computes the cost of the new configuration. The cost lookup table calculates the difference.

All requirements are stored in the special timetable memory, a form of the generalized tuple memory shown in Figure 8. Because a sequential counter addresses this memory, requirements are chosen sequentially rather than randomly from the tuple space.

The algorithm reads each tuple from the timetable memory and presents the tuple attributes to the clash array. The clash array stores counts of occurrences of each attribute in each period of the timetable. Two copies of each count are stored to allow simultaneous computation of the cost of inserting the attribute in the new period and removing the attribute from the current period. The new and current periods are routed via a crossbar switch (a simple form of the generalized mixers in Figure 8) to each clash array. An incrementer/decrementer (a simple form of the cost-delta circuit) processes the count in each memory to produce a new count and the cost of inserting and deleting that attribute. The costs are collated into an address word and sent to a cost table, which stores the results for all possible combinations of cost change.

The crossbar switch lets each pair of memories address the current class, teacher, and room with both current and new period values. While the cost is accumulated, the first memory must read the current period count, and the second must read the new period count. However, during update the memories must be made consistent, and the first memory must read the new period count and the second memory the current one. Thus, during update the crossbar switch is reversed.

If the net change is negative, the new period is written to the timetable memory, and the clash arrays are updated accordingly. If the change is positive, it is used as the key to an exponential table, which stores the probability of the cost being accepted given the current temperature, rescaled as a 16-bit integer. The probability is compared with a random number, which determines whether the change is accepted.

Each time the hardware processes a requirement, it chooses a new random period. A simple linear-feedback shift register generates a 32-bit random number, from which 15 bits are extracted. Because the period must be chosen in a range that may not be a power of two, a random number is reached by computing the modulus of the number with the number of periods per week. This computation yields a number in the correct range. Division is avoided by using a precomputed table of **mod** operations.

Implementation and performance. The current hardware occupies two IBM PC AT standard multiwire boards. The interface to the PC uses only an 8-bit bus. The cards contain about 200 standard ICs, with special-purpose control logic rather than any specific microprocessor. The machine is complex: It contains approximately 120 74Fxx and 74LSxx circuits, 37 memory chips, and 36 programmable logic arrays. Application-specific gate arrays could reduce this chip count significantly. The current hardware cost is about $700.

To compare the performance of the annealing hardware, my colleagues and I wrote a number of different versions and ran them on various machines. We wrote a very fast, simple version in the C programming language and ran it on a Sun SparcStation 1+, an Encore Multimax fitted with NS32332 processors, and a Cray Y/MP. We also wrote a full version that enforced all the constraints. However, it was written in Pascal and could not be run on the Cray.

The hardware solution ran about 33 times faster than the Cray, 77 times faster than the SparcStation, and about 250 times faster than the Multimax, all running C. Interestingly, the Cray version ran only about two times faster than the SparcStation version. Analysis indicates that the Cray version performed poorly because the code has very few floating-point operations and almost no vector operations. Given the algorithm's Monte Carlo nature, it is unlikely that a vector form could be devised.

The hardware ran about 200 times faster than the full Pascal version running on the SparcStation and nearly 1,000 times faster than Pascal running on the Multimax.

Using simulated annealing, a special-purpose architecture can solve important scheduling problems at speeds greatly surpassing those of conventional workstations and supercomputers using the same algorithm. The specific implementation for solving the school timetable problem has also been used for solving the airline gate-scheduling problem. A commercial product, First Class (produced by Computer Techniques P/L), incorporates the architecture described here to provide an intelligent assistant for creating school timetables.

Many algorithms for next-generation intelligent systems will require more performance than can be provided by conventional workstations. Special-purpose architectures may be appropriate for achieving such performance. For example, attached processors for high-speed execution of neural networks could allow such networks to be incorporated into conventional intelligent systems. The work reported here demonstrates that such an approach is feasible.

This article shows how two simple cost measures can be used to solve complex scheduling problems. Development of more generic cost measures will allow the architecture to be applied to other more varied scheduling problems. ■

Acknowledgments

The High-Performance Computation Program is a joint project of the Commonwealth Scientific and Industrial Research Organisation (CSIRO), Division of Information Technology, and the Royal Melbourne Institute of Technology (RMIT). Thanks go to Ian Albrey and Paul Guignard for editing a draft of this article, and to the referees for their helpful comments.

References

1. S. Kirkpatrik, C.D. Gelatt Jr., and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, Vol. 220, No. 4598, May 13, 1983, pp. 671-680.

2. P.J.M. van Laarhoven and E.H.L. Aarts, *Simulated Annealing: Theory and Applications*, Kluwer Academic, Boston, 1987.

3. E.H.L. Aarts et al., "A Parallel Statistical Cooling Algorithm," *Proc. STACS 86*, Springer Lecture Notes in Computer Science, Vol. 210, Springer-Verlag, Berlin, 1986, pp. 87-97.

4. E.H.L. Aarts et al., "Parallel Implementations of the Statistical Cooling Algorithm," *Integration*, Vol. 4, 1986, pp. 209-238.

5. A. Casotta, F. Romeo, and A.L. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells," *Proc. IEEE Int'l Conf. Computer-Aided Design*, IEEE CS Press, Los Alamitos, Calif., Order No. M744 (microfiche), 1986, pp. 30-33.

6. V. Cerny, "Multiprocessor Systems as a Statistical Ensemble: A Way Towards General-Purpose Parallel Processing and MIMD Computers?" unpublished manuscript, Comenius Univ., Bratislava, Czechoslovakia, 1983.

7. D.J. Chyan and M.A. Breuer, "A Placement Algorithm for Array Processors," *Proc. 20th Design Automation Conf.*, IEEE CS Press, Los Alamitos, Calif., Order No. M447 (microfiche), 1983, pp. 182-188.

8. D. Greening, "A Taxonomy of Parallel Simulated Annealing Techniques," Tech. Report CSD-890050, Computer Science Dept., Univ. of California, Los Angeles, 1989.

9. D. Abramson, "Constructing School Timetables Using Simulated Annealing: Sequential and Parallel Algorithms," *Management Science*, Vol. 37, No. 1, Jan. 1991, pp. 98-113.

10. D. de Werra, "An Introduction to Timetabling," *European J. Operations Research*, Vol. 19 1985, pp. 151-162.

11. N. Wu and R. Coppins, *Linear Programming and Extensions*, McGraw-Hill, Maidenhead, UK.

Are You at Level 3 Yet?

A mature software process must include a software cost estimation methodology. **Costar** is based on the COCOMO model described by Barry Boehm in *Software Engineering Economics*.

COCOMO is used by thousands of software managers to estimate the cost, staffing levels, and schedule required to complete a project — it's reliable, repeatable, and accurate.

Estimates are based on 15 factors that determine the effort required for a project, including:
- The Capability and Experience of your Programmers & Analysts
- The Complexity of your product
- The Required Reliability of your product

Costar is a complete implementation of the COCOMO Detailed Model, so it calculates estimates for all phases of your project, from Requirements through Coding, Integration, and Maintenance. **Costar** includes the calibration tools you need to tune COCOMO to your development environment. User definable cost drivers and a wide variety of reports makes **Costar** flexible and powerful.

Costar supports Ada COCOMO & Function Points.

Costar runs on the VAX and MS-DOS PCs.

Softstar Systems
(603) 672-0987
28 Ponemah Road
Amherst, NH 03031

Call for a free demo disk.

SOFTST★R

Reader Service Number 2

David Abramson is the program leader for the High-Performance Computation Program in the Commonwealth Scientific and Industrial Research Organisation's Division of Information Technology. He is also interim director of the Royal Melbourne Institute of Technology Centre for Concurrent Computing, group leader for the High-Performance Computation program within the Collaborative Information Technology Research Institute between the Royal Melbourne Institute of Technology and the University of Melbourne, and the program director of the High-Performance Computation Program within the Centre for Intelligent Decision Systems Cooperative Research Center. He is the general cochair for the 1992 International Symposium on Computer Architecture. His work has spanned most aspects of computer architecture and parallel programming, including algorithm design, instruction, hardware and architecture design, and programming languages.

Abramson holds the BSc and PhD degrees in computer science from Monash University in Melbourne, Australia. He is a member of the IEEE and the ACM.

Readers can contact the author at Division of Information Technology, Commonwealth Scientific and Industrial Research Organisation, 723 Swanston St., Carlton, Vic 3053, Australia.