# Delay-Insensitive Synchronization on a Message-Passing Architecture with an Open Collector Bus

Bekker, H.; Dijkstra, E.J.

*Published in:*
EPRINTS-BOOK-TITLE



*Publication date:*
1996

# Delay-Insensitive Synchronization on a Message-Passing Architecture with an Open Collector Bus

H. Bekker

Department of Computing Science,
University of Groningen,
9700 AV Groningen, The Netherlands

E.J. Dijkstra

Department of Computing Science,
University of Groningen,
9700 AV Groningen, The Netherlands

## Abstract

*The performance of some algorithms, running on a message passing computer, is limited by the high latency of global communications. To increase the performance, a simple open collector bus, operated by delay insensitive programs running on each processor can be used. We illustrate this by an example: the constraint algorithm SHAKE as used in Constraint Molecular Dynamics (M.D.) simulation. We present a parallelizable SHAKE algorithm and show how it can be implemented on a ring architecture. On a large ring the use of message passing to synchronize SHAKE iterations may take up to 40% of the total time. We show how the communication time can be reduced by adding a very simple open collector bus, operated by a delay insensitive algorithm. In this way the time spent on the synchronization of SHAKE iterations will be negligible.*

*We want to emphasize that this kind of open collector bus can be used with many delay insensitive algorithms. To show this we will mention other possible applications.*

*Key words: synchronization, delay insensitive algorithm, open collector bus, constraint dynamics.*

## 1 Introduction

Message passing systems consisting of a large number of processors, connected by a sparse interconnection topology (e.g. a ring or a mesh) prove to be a cost effective solution for many practical applications. These systems offer local communication with a high bandwidth and a low latency, but their global communication falls short in two respects: bandwidth and latency. This may give problems for the following classes of algorithms:

(i): Algorithms limited by the sustained bandwidth of the architecture. These algorithms often require local or global communication of large amounts of data followed by calculations which take up less time. This paper is not about this class of algorithms but about:

(ii): Algorithms limited by the latency of the communications. These algorithms often require global communication of very small amounts of data followed by some calculations. An important instance of such an algorithm is the synchronization of a fine grained iterative process.

This paper is about a simple hardware extension, solving the class of problems described in (ii). To be more specific, we will show that a very simple open collector bus, (O.C.-bus) running along all the processors, may be used to improve the performance of the algorithms in (ii).

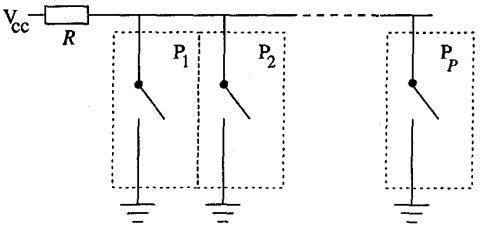


Figure 1: Open collector line, running along $P$ processors. The value read at every processor is the same. It is the logical **and** of the values written at all processors, where True(False) corresponds to an open(closed) gate.

The O.C.-bus consists of a few (4...8) lines, without any further lines for clocking or control. (See figure 1). Each of these lines is memory mapped, which means that by clearing or setting a specific bit in memory every processor can clamp (pull down) respectively unclamp the line. By reading a specific bit in memory, every processor can obtain the logical value of each line. This value is the same on every processor, and is the logical **and** of the values written on that line.

We will show that delay insensitive algorithms are very suitable to operate this bus. By using delay insensitive algorithms a low quality implementation of the O.C.-bus is acceptable: no bus terminators are required, no characteristic impedance, etc. Moreover, the use of delay insensitive algorithms, also called self timed algorithms, makes it possible to operate the O.C.-bus without clock or control lines.

The rest of this paper consists of a worked out example of molecular dynamics simulation on a message passing system. First a general introduction of constraint molecular dynamics is given. Then it is shown that running this algorithm on a ring architecture leads to the type of problem described in (ii). Finally, it is shown how the use of an O.C.-bus, operated by a delay insensitive algorithm solves this problem. Also two similar applications of the O.C.-bus are mentioned.

1066-6192/96 $5.00 © 1996 IEEE
*Proceedings of PDP '96*

## 2 Constraint Molecular Dynamics simulation

Molecular Dynamics (M.D.) Simulation is a method to simulate the behaviour of a many particle (atom) system by numerically integrating Newton's equation of motion. M.D. simulation is performed as follows: the initial system state $S_0$, that is, the position $\mathbf{r}_i$ and velocity $\mathbf{v}_i$ of every particle $i$ in the system at time $t_0$ is given. By integrating Newton's law $\mathbf{F}_i = m \cdot \mathbf{a}_i$ for every particle, subsequent states $S_1, S_2, \ldots, S_n$ are calculated where $S_n \equiv S(t_0 + n\Delta t)$. To calculate $S_{n+1}$ from $S_n$, first the total force $\mathbf{F}_i(t_0 + n\Delta t)$ on every particle $i$ due to all other particles in the system is calculated. Then this force $\mathbf{F}_i(t_0 + n\Delta t)$ is used to calculate for every particle the new velocity $\mathbf{v}_i(t_0 + (n+1)\Delta t)$. Using this velocity, the new position $\mathbf{r}_i(t_0 + (n+1)\Delta t)$ of every particle is calculated. Repeating this procedure gives the time development of the system.

Every timestep, during the force calculations, many types of interaction-forces are evaluated: Coulomb forces, Lennard-Jones forces, covalent forces, etc. Some of these interactions are very rigid. The most rigid interaction in an M.D. simulation is the covalent interaction. This means that two particles having a covalent interaction, have an almost constant distance. Put in another way: covalent interactions have a high eigenfrequency. The maximal allowed timestep used in an M.D. simulation is dictated by the allowed numerical drift of the integration algorithm, so it is dictated by the highest frequency in the system, and should be approximately $1/(40 \times \textit{highest frequency})$. However, the behaviour of covalent interactions is not part of the physics of interest of an M.D. simulation. Leaving out frequencies above 1/4 to 1/2 the highest frequency does not influence the outcome of an M.D. simulation. So, it is a waste of computer time to use a timestep based on covalent eigenfrequencies. For that reason, nowadays in most M.D. programs, the covalent interactions are handled using *constraint dynamics*, which means that the distance between particles with a covalent bond is kept constant. Then the timestep may be as high as 1/20 to 1/10 ($\times$*highest frequency*). In this way an M.D. simulation runs two to four times faster.

Because an atom may have covalent interactions with a number of atoms[1], substituting covalent interactions by length constraints will in general result in a set of connected length constraints with a, possibly cyclic, graph like structure. Covalent interactions are bonded interactions, so, no constraints are created or broken during a simulation.

The introduction of length-constraints has no consequences for the force calculations, except of course that the forces of covalent interactions are not calculated. However, the introduction of length-constraints has severe consequences for the algorithm in which Newton's law is integrated, resulting in a matrix equation. As the rank of the matrix is the number of constraints in the system, for systems with many constraints, solving this equation directly on a parallel computer is complex. There exists however a fast, iterative method called SHAKE [1], to solve the matrix equation. The special thing about SHAKE is that its iterative way of solving the matrix equation is directly reflected in iterative adjustment of pairs of particle positions. This last interpretation of the SHAKE method has become so familiar that it is almost forgotten that it is a matrix solver in disguise. We will adhere to this habit, and in what follows write about the SHAKE algorithm as pairwise adjusting particle positions to constraint conditions in an iterative way.

SHAKE is used as follows. Every timestep, the interaction forces, the new velocities and new positions are calculated as if no constraints exist, except that no covalent interaction forces are evaluated. Clearly, particle positions obtained in this way do not fulfill the distance constraints between particles. Then SHAKE is invoked. In SHAKE, particle positions are corrected in an iterative way, such that finally all length-constraints are fulfilled within a predefined tolerance. So, at the end of every timestep many SHAKE iterations have to be done.

SHAKE is implemented as follows. The particle numbers of every pair of particles between which a distance constraint exists, are kept in a constraint-list (CL). So, in CL, every constraint is represented by two particles. (In this article we assume that the constraint distance is the same for every constrained pair of particles, so we need not store this in CL.) In every iteration, SHAKE goes through CL once; the order in which items of CL are processed does not matter. Processing an item of CL means that the positions of the particles of this pair are adjusted such that their relative distance becomes the required constraint distance.[2] Because a particle may have more than one constraint interaction, repositioning a particle due to one constraint may disturb another, previously adjusted constraint. Therefore, after every iteration of adjusting positions of particle pairs, the constraint conditions are checked. If these conditions are not fulfilled within a predefined tolerance, another iteration is done, in which all constraints in CL are processed once again. Typically, at the end of every timestep SHAKE does 4 . . . 40 iterations, but for large molecules, some hundreds of iterations may be required. On a single processor, SHAKE typically takes 5. . .20% of the total CPU time of an M.D. simulation.

In the SHAKE algorithm as we presented it, two particle *positions* are adjusted when processing an item from CL. When processing the next item from CL, a possibly previously adjusted particle position is adjusted further. So, items from CL cannot be processed simultaneously. Fortunately, the SHAKE algorithm can be restated without these dependencies. When processing an item from CL, instead of immediately adjusting the two particle positions, the resulting particle *displacements* are accumulated.[3] After processing the whole CL, particles are displaced over their accumulated displacements. In pseudo code, the parallelizable SHAKE algorithm looks like:

[1] In a typical M.D. system the number of constraints is of the same order as the number of particles.

[2] Although not relevant for this paper, adjusting positions goes as follows. The particles of the constrained pair $a, b$, with positions $\mathbf{r}_a$ and $\mathbf{r}_b$, are reset in the direction of $\mathbf{r}_a(t - \Delta t) - \mathbf{r}_b(t - \Delta t)$, such that the center of mass of this pair does not change, and their distance becomes the constraint distance.

[3] This cannot be derived by transformations of the algorithm, but is a matter of numerical mathematics.

```
procedure SHAKE;
type rvec= array[1..3] of real;
   partId=1..N; { number of particles is N }
var
   r, displ: array [partId] of rvec;
   CL: array [1..nr_constr] of
                record a,b: partId end;
begin
repeat
   clear(displ);
   for i:=1 to nr_constr do begin
      displ[CL[i].a] += .... ; { See footnote[2] }
      displ[CL[i].b] += .... ; { See footnote[2] }
   end;
   for i:=1 to N do r[i] += displ[i];
until all_constraints_within_tolerance;
end;
```

## 3 SHAKE on a ring architecture

At the departments of physical chemistry, and computer science in Groningen, the M.D. simulation package GROMACS [2,3] has been implemented on a custom-built ring architecture, consisting of 32 i860 processors. Each i860 board plugs into a collective PC bus, and has two eight bits wide parallel interfaces (2 Mb/sec) to connect the board in the ring. Also on the PC bus is an i486, running UNIX, which serves as a host. This host uses the PC bus to load code and initial data on the i860 processors, and for I/O purposes.

In the GROMACS ring implementation, particles are statically allocated on processors. An M.D. system of $N$ particles, numbered from 1 to $N$, is mapped on $P$ (in our case $P = 32$) processors by allocating the first $N/P$ particles on processor 1, the second $N/P$ on processor 2, etc. The processor $H_i$ on which particle $i$ is allocated is called its *home processor*. The home processor of particle $i$ calculates the final position of $i$ after a timestep (constrained and unconstrained). As will be clear, the particle numbering determines the home processor of every particle.

For the force calculations it does not matter how particles are allocated on the processors. That is because every particle potentially interacts with every other particle. Therefore, at the beginning of every timestep, the position of every particle $i$ is, starting from its home processor $H_i$, distributed over half the ring, in say the positive direction. Distribution over half the ring is sufficient because in this way every position pair $\mathbf{r}_i, \mathbf{r}_j$ is present on at least one processor. After this distribution stage, interaction forces are calculated. Then the interaction forces on every particle $i$ are communicated [4] in the negative direction to $H_i$ where they are summed to the net force on particle $i$. Finally, on the home processor of each particle its new, unconstrained velocity and position is calculated. Now SHAKE is invoked. Because between any two particles there may be a constraint, in principle for every SHAKE iteration, as in the force calculations, particle positions would have to be distributed over half the ring. However, this would take far too much time. In [4] we proposed a method, to minimize communication during SHAKE calculations. It is based on the bandwidth reduction algorithm of Gibbs, Poole and Stockmeyer. The essence of the method is that particles are numbered in such a way that particles between which a length constraint exists, get close numbers, so, are mapped on close processors. In fact, with this method, even for rather complex molecules, particles between which a constraint exists, are mapped on the same or on directly adjacent processors.[5] So, during SHAKE only communication between very near processors is required.

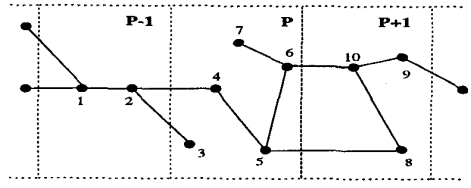


Figure 2: The lists NCCI, LCI and PCCI on processor $p$ for the part of some constraint graph mapped near processor $p$. NCCI= (3,2); (4,2). LCI= (4,5); (5,6); (6,7). PCCI= (6,10); (5,8). On processor $p$ constraint interactions in NCCI and LCI are evaluated.

Most parallel implementations of the M.D. algorithm do not include constraint dynamics. In those cases where it is included [5,6] no use is made of accumulated displacements to parallelize SHAKE, nor the bandwidth reduction method to minimize communication during SHAKE iterations.

We are now almost ready to write down the SHAKE algorithm as it runs on every processor of the ring, but first we will explain how the global constraint list CL, as introduced in section 1, is distributed over processors, and how on every processor this partial list is partitioned into even smaller parts.

The contents of CL do not change during a simulation, so an almost perfect load balance for parallel SHAKE calculations can be accomplished by assigning the same number of items of CL to every processor. On every processor the constraint interactions assigned to that processor are stored in LCL (Local Constraint List). The list LCL is subdivided still further into three sublists (see figure 2): NCCI, LCI, and PCCI (Negative-Crossing, Local-, and Positive-Crossing Constraint Interactions). On processor $p$, NCCI (PCCI) contains those constraint interactions of which particle $b$ is home on processor $p-1$ ($p+1$), and particle $a$ is home on $p$. LCI contains interactions of which both particles are home on $p$. So on $p$, the list PCCI contains the same number-pairs, but in reverse order, as the list NCCI on $p+1$. The data structures NCCI, LCI and PCCI can be used to define which constraint interactions are evaluated by which processor: *processor $p$ evaluates the constraints in its NCCI and LCI list.* Then the SHAKE algorithm as it runs on every processor is as follows:

[4] On the GROMACS ring architecture this communication, together with the foregoing communication to distribute particle positions takes about 5% to 10% of the total time.

[5] If this is not the case, the particles are still mapped on very close processors. Such a case can be handled by a straightforward extension of the method we propose, but we did not encounter molecules with constraint structures of this complexity.

```
procedure SHAKE;
{ parallel SHAKE on a ring architecture,
    see also figure 2 }
type rvec= array[1..3] of real;
partId=1..N; { N is the number of particles }
var
r, displ: array [partId] of rvec;
NCCI: array [1..nr_NCCI] of record a,b: partId end;
      { home(a)= p, home(b)=p−1 }
PCCI: array [1..nr_PCCI] of record a,b: partId end;
      { home(a)=p, home(b)=p+1 };
LCI: array [1..nr_LCI] of record a,b: partId end;
      { home(a)=p, home(b)=p };
begin
send PCCI_b_positions to posdir;
receive NCCI_b_positions from negdir;
repeat
  clear(displ);
  calculate displacements;
      { due to constraints in NCCI and LCI}
  send NCCI_b_displacements to negdir;
  receive PCCI_b_displacements from posdir;
  sum displacements and add to r;
      { for particles home on this processor }
  send PCCI_b_positions to posdir;
  receive NCCI_b_positions from negdir;
  LCWT:=check_local_constraints;
      {Local_Constraints_Within_Tolerance}
until ACWT;
    {All_Constraints_Within_Tolerance}
end;
```

With this, the parallel SHAKE algorithm is completely specified, except for the last few statements with LCWT and ACWT (Local- and All Constraints Within Tolerance), which concern the evaluation of the global stop criterion of SHAKE iterations during the current timestep. This will be discussed in the next section.

## 4 The function ACWT implemented with the open collector bus

SHAKE iterations should be stopped when on every processor the constraints in the lists NCCI and LCI are within tolerance, i.e. when on every processor the boolean variable LCWT is true. Representing LCWT on processor $p$ by $\mathrm{LCWT}_p$, the function ACWT can be specified as ACWT:=$\mathrm{LCWT}_1$ **and** $\mathrm{LCWT}_2$ **and** ... **and** $\mathrm{LCWT}_P$.

On a message passing ring architecture such as GROMACS, the function ACWT can be implemented in three ways.
(i) By sending a single message around the whole ring twice. First the message accumulates the logical **and** of all LCWT, and in a second round this result is passed to all processors as ACWT.
(ii) As $P$ messages, all moving around the whole ring once. At every processor one message is released which returns to that same processor. While moving around the ring, the message evaluates the logical **and** of all LCWT. When arriving at the processor from which it was released, this processor inspects the contents of the message to see if another iteration of SHAKE is required.
(iii) The third implementation uses the PC bus and the host computer. Every processor sends its LCWT to the host. There, the logical **and** is evaluated and transmitted back to the individual processors.

As will be clear, each of these methods takes at least $P$ communications. On our GROMACS ring implementation, we measured that sending a minimal message (1 byte) from a processor to an adjacent processor, or from a processor to the host, takes $\sim 150\,\mu$sec, mainly due to startup overhead. So, for $P = 32$, evaluating ACWT takes at least $32 \times 15 \times 10^{-5} = 4.8 \times 10^{-3}$ sec. We also measured that the *calculations* of one SHAKE iteration take the same amount of time. (20 SHAKE iterations, without communication can take 10% of the total time. One timestep takes 0.1...1 sec, let us say 1 sec, then one SHAKE iteration takes about $5 \times 10^{-3}$ sec.) On the present architecture the one-to-one ratio of the time spent in SHAKE calculations and its synchronization is however no problem because SHAKE typically takes only 10% of the total time, so spending an additional 10% in ACWT is no major problem. However, when the same type of simulation is done on a ring consisting of twice as many processors, the total time spent on calculations is halved while the time spent in ACWT doubles. So, about 40% of the total time will be spent in ACWT.

To solve this problem we will equip our next architecture with an eight line O.C.-bus as described in the introduction. The function ACWT can be implemented using four of these lines. We will call these four lines "valid", "accepted", "next", and "data". We name the values written on these lines: l_valid, l_accepted, l_next, l_data; and the values read: g_valid, g_accepted, g_next, g_data. The prefixes l and g stand for local and global. The local variables are write-only and the global variables are read-only. The signals "valid", "accepted" and "next" will serve as global control signals. In [7] it is explained why at least three control signals are required. A delay insensitive implementation of ACWT which runs on each processor, is straightforward:

```
function ACWT;
begin
  l_data:= LCWT;
  l_accepted:=False; l_valid:=True;
  repeat until g_valid;
  ACWT:=g_data;
    {ACWT:= LCWT_1 and ... and LCWT_P}
  l_next:=False l_accepted:=True;
  repeat until g_accepted;
  l_valid:=False; l_next:=True;
  repeat until g_next;
end;
```

After the first repeat, "data" is valid. After the second repeat, "data" has been accepted by all processors. The third repeat is necessary to return to the neutral state. Initially, l_valid (g_valid) must be False. It can be seen that after g_valid becomes true, i.e. after the slowest process has evaluated its LCWT and assigned it to l_data, the evaluation of the function ACWT proceeds without delay. On every processor, immediately after finishing the function ACWT the next SHAKE iteration is started, or SHAKE iterations are stopped.

## 5 Discussion

In our particular case, that is, constraint molecular dynamics on a ring architecture, adding a small O.C.-bus is a sensible investment because the price of this feature is low (a few hundred dollars) compared to the price of the whole computer ($\approx$ \$100,000), while the speed increase is much higher than this ratio. Moreover, the hardware risk that goes with this feature, that is, the risk of destabilizing an otherwise well functioning architecture, is very small.

In our opinion, there are many other useful applications of an O.C.-bus in a message passing computer. Especially the combination of an O.C.-bus with delay insensitive algorithms looks promising. We will briefly mention two other applications.

The first example we want to mention is process arbitration. On every processor a number is generated. Process arbitration means that on every processor it is decided whether the highest number is on this processor. A delay insensitive arbitration algorithm, using four wired **or** lines has been designed (unpublished) by C.E. Molnar. This way of process arbitration will be much faster than using the message passing mechanism.

The second example is the TRIMOSBUS [7]. The TRIMOSBUS is a general purpose bus, operated with a delay insensitive algorithm. It consists of at least four open collector lines, three of which are used for sequencing, and the other ones as data lines. It may be used for arbitrary point to point communications, and for broadcasting. In this way, on a parallel computer, small amounts of data may be exchanged between processors much faster than with the usual message passing mechanism.

The research field of "delay insensitive" algorithms and hardware is thriving nowadays. Many delay insensitive algorithms are conceived, and experimental, delay insensitive hardware is designed. Of both, the correctness can be proved by delay insensitive algebra [8,9]. How delay insensitive algorithms and hardware will develop is not clear at this moment. We do feel however, that a simple O.C.-bus connecting the processors of a parallel message passing architecture, combined with delay insensitive algorithms, is a simple and fast general purpose feature, which may be used to increase the performance of algorithms which cannot be implemented efficiently or elegantly with the message passing mechanism. Obviously, an O.C.-bus cannot replace the usual communication and routing hardware of message passing systems, but for a number of applications it can increase the performance in a straightforward way. Because the price/performance ratio of the O.C.-bus is low, and because it is a simple and robust piece of hardware, it is worth considering to add this hardware feature to sparsely connected parallel computers.

A reviewer remarked that the synchronization mechanism described in this article strongly resembles the barrier synchronization mechanism of the CRAY T3D architecture.

## 6 Conclusions

A small O.C.-bus, operated by delay insensitive algorithms, is a fast and simple mechanism, which on message passing systems can be used to increase the performance of many applications.

An example of such an application is constraint M.D. implemented with the SHAKE algorithm. A SHAKE iteration can be parallelized by accumulating the *displacements* of every particle and adding the total displacement to the particle position at the end of the iteration.

Synchronizing iterations and the termination of SHAKE on a message passing ring architecture, proves to be relatively time-consuming due to the latency of message passing. SHAKE becomes less time-consuming by extending the hardware with a small O.C.-bus. Synchronization can then be done by a simple delay insensitive algorithm.


## Acknowledgments

We want to thank M.K.R. Renardus for carefully reading and commenting this text, J.T. Udding for his expertise in the field of delay insensitive algorithms, and the reviewer for making useful remarks.


## Literature


**[1]** J.P. Ryckaert, G. Ciccotti, H.J.C. Berendsen, Numerical integration of the Cartesian equations of motion of a system with constraints: molecular dynamics of n-alkanes. *Journal of Comp. Phys.* **23**, 327-341, 1977.

**[2]** H. Bekker, H.J.C. Berendsen, E.J. Dijkstra, S. Achterop, R. v. Drunen, D. v.d. Spoel, A. Sijbers, H. Keegstra, B. Reitsma and M.K.R. Renardus, GROMACS: a parallel computer for molecular dynamics simulation. *Conf. Proc. Physics Computing '92*, pages 252-256, World Scientific Publishing Co. Singapore, New York, London, 1993.

**[3]** H. Bekker, E.J. Dijkstra, H.J.C. Berendsen. Molecular Dynamics simulation on an i860 based ring architecture. *Supercomputer* 54, X-2, 4–10, 1993.

**[4]** H. Bekker, E.J. Dijkstra, H.J.C. Berendsen. Mapping molecular dynamics simulation calculations on a ring architecture. In *Parallel Computing: From Theory to Sound Practice*, ed. W. Joosen and E. Milgrom, pages 268–279, IOS Press, Amsterdam, 1992.

**[5]** A.R.C. Raine. Systolic loop methods for molecular dynamics simulation, generalized for macromolecules. *Molecular Simulation*, Vol. 7, pages 59-69, 1991.

**[6]** S.E. DeBolt, P. Kollman. AMBERCUBE MD, Parallelization of AMBER's Molecular Dynamics Module for Distributed-Memory Hypercube Computers. *Journal of Comp. Chem.*, Vol. 14, No. 3, 312-329, 1993.

**[7]** I.E. Sutherland, C.E. Molnar, C.E. Sproull, J.C. Mudge. The TRIMOSBUS. *Proc. of the Caltech Conf. on VLSI*, January 1979.

**[8]** L. Lavagro and A. Sargiovanni-Vincentelli. Algorithms for Synthesis and Testing of Asynchronous circuits, Kluwer Academic Publishers, 1993.

**[9]** M.B. Josephs, J.T. Udding. An overview of Delay Insensitive Algebra. In *Proc. of the* $26^{th}$ *Annual Hawaii Int. Conf. on System Sciences*, ed. T.N. Mudge, V. Milutinovic, L. Hunter, 329-338, IEEE Computer Society Press, 1993.