The Design of a Clock Synchronization Sub-system for Parallel Embedded Systems

M. Fleury, A. C. Downton, A. F. Clark and H. P. Sava

Electronic Systems Engineering Department, University of Essex, Wivenhoe Park, Colchester, CO4 4SQ, U.K

tel: +44 - 1206 - 872795

fax: +44 - 1206 - 872900

e-mail fleum@essex.ac.uk

Abstract

Software tools are being developed to support a design methodology specific to parallel real-time continuous-dataflow embedded systems. This paper describes the design of a global clock sub-system which is an essential component of an event trace tool. A new amalgam of algorithms is proposed which attends to the trade-off between clock accuracy and the need to restrict disturbance of the application whilst recording traces. The details of an implementation on a hybrid parallel processor as well as the results of tracing applications in the given problem domain are included.

1 Introduction

Real-time, continuous-dataflow embedded systems are an important class of engineering system for which parallel solutions have much to offer. Examples of such systems can be found in vision [1], radar [2], speech processing [3] and signal compression [4]. The algorithmic complexity and irregularity of such systems means that parallelizing compilers are unlikely to provide much support in the near future [5, pp. 149–155]. Instead, a common design methodology based upon Pipelines of Processor Farms (PPFs) [6] has been proposed which offers a single conceptual design model which is capable of incorporating data parallelism, algorithmic parallelism or temporal parallelism within individual farms. The methodology addresses the requirement to parallelize systems composed of a number of independent algorithms, can handle time, ordering, data dependency and feedback constraints, and produces incrementally scalable solutions. Achievable throughput, latency and upper bound speed-up can be determined by analysis of the original application code.

The PPF design methodology addresses an intermediate level of generality appropriate for a particular domain of applications. It is not limited to a particular machine architecture or a particular processor topology for each farm. A message-passing paradigm is assumed but it is possible and can be desirable to emulate message-passing on a shared-memory machine. The methodology does, however, rely on central farmer processes distributing work and sets of worker processes which perform that work by some means. A high-level application building

block, a farm template, has been designed to speed up prototyping [7]. The template incorporates instrumentation to enable event-tracing. Event traces can be used to give confidence in the correct working of the application and to facilitate performance analysis.

Since the role of the worker process is passive in the PPF methodology it is not appropriate for the worker to initiate tracing when it has reached steady-state. To provide event traces, a centralized clock synchronization system is therefore required.
The synchronization algorithm introduced in this paper is intended *specifically* to support the PPF methodology. The main features of the algorithm are:

- an initialization phase in which the drift of the system from real time is assessed centrally;
- averaging to achieve a system reference time;
- subsequent refresh signals at a predetermined interval so as to prevent drift from the system time;
- local correction of drift between refresh signals.

For the clock mechanism, the essential component of a trace, we were able to provide a central mechanism capable of tightening the precision of the clocks by reason of global knowledge. If a communication harness with integrated trace instrumentation is designed for general use (such as PICL [8]), a client-server clocking system may be preferred, which makes the time-server a passive component. A client-server system is a disadvantage for tracing as it does not allow the pattern of synchronization traffic to be controlled for the purpose of reducing perturbation of the application, though it is more flexible.

2 Design Criteria

If global time is not kept consistently when timestamping a record of events in a multi-computer application, then the ordering of significant events on different nodes (i.e. usually

processors) can easily become awry. In particular, if the ordering of communication events is lost then the phenomenon of 'tachyons' occurs in which a message apparently arrives before it is sent. Solutions exist in which logical clocks [9, 10] are kept, but these involve time-stamping each message. Such systems are applicable to distributed systems or where debugging is a goal [11]. Fault-tolerant solutions [12, 13, 14] typically involve $O(n^2)$ messages for n processes even if all message journeys are confined to a single hop. In [15] an $O(n)$ message solution relies on the presence of an embedded ring topology, which excludes tree-topologies, though these are a natural topology for the data-farming applications of concern for us. Other work [16], though important theoretically, relies on complete graphs. Finally, statistical post-processing of the trace record [17] represents an alternative approach.

A method involving interpolation between start and end timing pulses [18] was initially explored but, though suitable for gauging intra-node process activity, was not found to be sufficiently accurate. A number of convergence algorithms [19, 20] exist but these may not provide accurate timing at the inception of an algorithm. However, unlike distributed or networked methods [21], it is possible to have an initial traffic-free phase; and one can assume that faults are rare, allowing a centralized algorithm.

A criterion for a portable design is that one should aim to avoid a solution involving low-level manipulation of the hardware, as for instance in the synchronized wave solution for transputers of [22].

The requirements of the centralized, portable design can thus be summarized as:

- A minimum of user intervention is needed in regard to the clocking mechanism.
- The design should not be dependent on any feature of the hardware such as would, for example, necessitate the use of assembly language. Solutions involving hardware interrupts [23] are by no means trivial to implement and equally run into the problem of application perturbation.
- The number of synchronization messages should be minimized.

- The mechanism should not rely on a particular architecture.
- It should not be assumed that the system is already in steady-state or can converge slowly to synchronization.
- The times should conform to a suitable real time reference.

In practice, one requires two algorithms: an algorithm to bring the clocks on all nodes within some minimum error range and an algorithm subsequently to maintain time against clock drift.

3 The Processing Model

3.1 Processor Requirements

The method described below is suitable for processor nodes with at least two levels of priority and supporting internal concurrency, as is common for interrupt-driven applications. Only local physical clocks are assumed to be available (most multicomputers can provide high-priority monotonic clocks with resolution of at least 1μs). An intermediary process which acts as a monitor able to intercept communication is assumed in the model; the provision of the monitoring processs would be an implementation-dependent feature.

One use of event traces is to provide input to a visualizer, which acts as a data-reduction agent. Current visualizers when faced with applications of varying time duration or communication intensity may be limited by the display rate of the graphics terminal but this does not mean that one should not aim for as high a resolution as possible. Event traces may also be post-processed as input to an execution-driven simulation or an analytic model. In [24] there is an appropriately accurate method of supplying a global time reference, which was aimed at hypercube multicomputers. However, on its own this method only completely resolves communication events. Additionally, it is important for a trace mechanism with wide usage (though within the real-time PPF environment) to provide timestamps that are not biased by possible drift of the master clock. In particular, where interrupts occur real time

should be maintained. If real time is not required then it will be sufficient to synchronize to an uncorrected master clock. (The whole system will drift at the same linear rate.)

3.2 Local Clock Requirements

Experimental evidence [25] shows that crystal clocks drift linearly if the temperature regime is constant and that anyway drift is small ($< 10^{-5}$) so that second order terms are neglected in the following. (For runs longer than a few minutes temperature oscillations may occur for transputer clocks [26], in which case second order corrective terms offer one solution.) It is therefore assumed that for correct clocks the accuracy of clocks should be such that

$$(1-f)(t-t') - \rho \leq H(t) - H(t') \leq (1+f)(t-t') + \rho, \tag{1}$$

where f is the maximum clock frequency drift, t is a notional reference time (real time) such as Universal Time Coordinated (UTC) [21], ρ is the discrete clock precision, $H(t)$ is the reading of a clock at time t and $(t-t') > \rho$.

4 The Synchronization Model

4.1 Normal Behaviour

If the system is assumed to be in steady-state and the refresh interval has been calculated, normal behaviour proceeds as follows. The central synchronization server on node $i = 0$ polls every other node for its time reading by sending a synchronization 'pulse' and immediately receives back the reading from a particular node, as shown in Figure 2. The central node in each case records the time delay for the round-trip, as given by its local clock. The central node corrects its round-trip timing for local clock drift against an estimate of real time. It estimates the difference between its time reading and the other node's time reading by subtracting half the corrected round-trip time from the reading at the other node. By using the mid-point of the corrected round-trip time as an estimate of the reading at the other node, the central node minimizes its error in the long run. In other words, the mid-point is

an unbiased estimator of the arrival time of the pulse (i.e., its mean approaches the mean of the population of round-trip times).[1]

Formally, the offset estimate for node i ($i = 1, 2, \ldots n$) is

$$o_i = (D_i + c)/2 - r, \tag{2}$$

with D_i the total round-trip time recorded on the central node. c is the afore-mentioned correction to the central node's round-trip time, which will be further explained in equation 4. r is the interval between sending the pulse as recorded on the central node and receiving the pulse as recorded on the other node's clock when the pulse arrives. The central node completes its polling of the remaining nodes before sending the offset correction.

The advantage of further polling is that because there is no reason to give prominence to the central node's clock it is necessary to form some view of real time by an averaging procedure. In part, so as to reject outliers, particularly if the algorithm could also be adapted for times close to boot-up time, the median was chosen as an average. Once the median, m, is calculated, it is added to the offset for each of the nodes so that the final offset is

$$O_i = o_i + m. \tag{3}$$

On the central node, the offset is simply m. By recording successive median estimates, the central node is able to estimate its drift:

$$d_0 = (m(t) - m(t + R))/R \tag{4}$$

where $m(t)$ is the median estimate at time t and R is the refresh interval. Given this estimate, $c = D_i d_o$. Node 0 corrects its local clock by d_0 and sends the median-adjusted offsets to the nodes. On receipt of the offsets, each node additionally corrects for the drift in its time since the synchronization pulse was sent. This node drift correction is described in Section 5.

[1] Were *a priori* knowledge available on a skew in the ratio of outward to return-trip journey times then another point other than the mid-point could be taken. However, bear in mind that measurement of a skew implies a hardware timer, which would obviate the need for software timing.

Using the median rejects outliers but does not rely on *a priori* knowledge to reject them. It makes no assumptions about the nature of the aggregate message round-trip time distribution. Other methods that have been employed or suggested to increase the accuracy of readings include: using a recursive estimate of the round-trip time with a linear smoothing function [25]; and rejecting long delay times as these are more likely to include asymmetrical journeying times [27, 28]. These solutions rely on eventual convergence, whereas for a trace application in the PPF methodology one requires bounded accuracy at a time signalled from the central farmer. This contrasts with toolkits that allow a trace to be initiated when a process ascertains it has reached a steady state. Note that for cooperative algorithms using the median may be unsuitable since no new value is introduced [20] and convergence is therefore not guaranteed.

The calculation of the median is not onerous if we can employ [29, pp. 476–479]:

$$\sum_{i=1}^{n} \frac{x_i - x_m}{|x_i - x_m|} = 0 \tag{5}$$

(with n data values x_i having median x_m) to form an iterative and logarithmically convergent equation. (Other algorithms are at best $O(n \log(\log n))$ [30, pp. 216–220].) The situation is better than this: since we only estimate the true median we need only make a few iterations. The possibility of ill-conditioning may anyway make it necessary to curtail the iterations. An advantage of using a median-adjusted corrective term, particularly if the clocks were not to be brought together by the initialization phase, is that the correction jumps on each clock are reduced in aggregate. The possibility of global time appearing to jump forward abruptly or even move negatively is reduced.

4.2 Establishing the Refresh Interval

Since there is uncertainty in the estimate of the clock offset and clock drifts relative to a central reference it is necessary to refresh the clocks by periodic synchronization pulses. Unlike some other systems, the method used here does *not* seek to keep track of when to refresh each individual node with a separate synchronization pulse (or allow each node to request a synchronization from the central server). Instead, the worst-case error is found and

the nodes are refreshed accordingly. As it turns out it is possible to minimize the estimate of the worst-case error. An advantage of the single refresh time is that regular perturbations occur rather than perturbations caused by the particular clock skews and boot-up timings.

For simplicity consider the refresh time, R, necessary to ensure communication events are correctly ordered between just two nodes, which is given by

$$R = \frac{T_0(1) - 2e_d}{2e_f}, \tag{6}$$

where $T_0(h)$ is the minimum one-way message time with $O(0)$ data and h is the number of message hops. e_d is the actual error (in units of time) in the offset estimate (caused by the variance of round-trip timings). An upper bound for e_d is given in equation 7. The number e_f is the actual error in the skew-rate estimate. An upper bound for e_f is given in equation 10. R is the minimum time taken for the clocks to drift apart to such an extent that a message between any two processes might result in an ordering error. The factors of two arise because if the uncertainty between the central node and a subsidiary node is e then the uncertainty in timing between any two subsidiary nodes, were they to pass a message, is $2e$. T_0 can be found by measurement on an empty system. A long-tailed distribution of round-trip times is common to LAN networks [31] and multicomputers [24] alike. T_0 will be skewed at one end of this distribution. The clock difference error is bounded for any one node by

$$e_d < (D_i/2 - T_0(h))(1 + f_i) + \rho + f_i T_0(h). \tag{7}$$

$D_i/2 - T_0(h)$ represents the uncertainty in the reading of the remote clock. For the purposes of tracing one can make do with $T_0(1)$ rather than keep a table of minimum journey times indexed by h. f_i is the relative clock drift between node 0 and node i.

$$f_i = \frac{O_i(t + S) - O_i(S)}{S} \tag{8}$$

where $O_i(t)$ is the measured clock offset at time t and S is a suitable sample interval. In equation 7, $f_i T_0(h)$ is the uncertainty in node 0's measurement of the minimum journey time. The maximum error, e_m, that can be made is:

$$e_m = D_i/2(1 + 2f_m) - T_0(1) + \rho \tag{9}$$

where f_m is the maximum frequency drift for one clock, substituted for f_i into equation 7 in order to arrive at equation 9. In fact, this is the same equation for maximum clock reading error as developed in [31] by a different route and for a different purpose. The maximum possible error is determined by the maximum round-trip time which naturally arises as the synchronization method depends on message passing. An upper bound to the error in the frequency drift can also be arrived at:

$$e_f < \frac{2e_d}{S}. \tag{10}$$

Though one might estimate the S needed for a required e_f (by estimating e_d for each subsidiary node through comparing successive values of D_i) this would mean that S might cause an arbitrary delay at the beginning of a test in order to bound e_f sufficiently. A simple expedient of not allowing e_f above a characteristic data-sheet value for f_m ensures that accuracy is preserved at a small cost in efficiency.

Equation 7 relies on a measurement of D_i to bound the reading error. If the same refresh pulse is used for all nodes then one seeks the maximum error that can occur, which will arise from the maximum journey time. However, if successive sampling rounds are made to find the maximum journey time, then one can select the minimum maximum journey time. If the probability of finding a low enough value from any one round of samples is p, then the probability that waiting time will be k rounds before finding that value is $p(1-p)^{k-1}$. The expected number of Bernoulli trials (including the successful trial) to achieve a low enough value is

$$\overline{x_1} = \frac{1}{p} \tag{11}$$

with variance

$$\sigma^2 = \frac{1-p}{p^2}. \tag{12}$$

If one estimates that the probability of not getting a low enough value is 0.5, then $\overline{x}$ is two. We need to include at least σ extra trials to account for the error in the estimate, which is

approximately four trials in all. One can also examine the distribution of round-trip messages to estimate the number of rounds necessary. Certainly, sufficient samples should be taken to bracket any system interrupts.

5 Local Clock Adjustment

The method outlined in Section 4 does *not* do anything to ensure that between synchronization pulses local clocks remain accurate. For generality, a software correction method is described, though on some machines it is possible to correct by hardware means, (for instance, phase-locked loops on Internet machines serviced by the Network Time Protocol [28]). Each local node can estimate its drift relative to the averaged real-time by finding the change in clock differences over a number of synchronization rounds. Since drift is assumed to be linear, two observations are sufficient. A linear corrective function ensures smoothness, avoiding discontinuities, *viz.*

$$L_i(t) = H_i(t) + A_i(H_i(t))(H_i(t) - H_i(s_p)), \tag{13}$$

where L_i returns the local time for node i, $H_i(t)$ is the underlying software-adjusted clock and A_i is the corrective function:

$$A_i(H_i(t)) = (O_i(s_p) - O_i(s_{p-1}))/(H_i(s_p) - H_i(s_{p-1}),\ H_i(t) \geq H_i(s_p) \tag{14}$$

which is the change in offsets over the previous offset reporting times s indexed by p and $p-1$. H_i is re-adjusted at time s_p by an offset calculated centrally for time r. Thus, one should correct for the drift since then:

$$H_i^p(t) = H_i^{p-1}(t) + O_i(s_p) + A(H_i^{p-1}(s_p))(H_i^{p-1}(s_p) - r) \tag{15}$$

where the suffix on H_i indicates what is in effect a change of clocks. This method of local clock correction does not require regularly-spaced synchronization intervals nor does it require the correction term to be supplied immediately after the initial difference timing signals.

6 A Case Study: Implementation on the Paramid Multicomputer

Our system is in a development stage. This section presents a preliminary case study, but further implementations will be necessary to validate our approach. In particular, an implementation for a larger number of processors would be needed.

6.1 The System Set-up

The Paramid [32] is a transputer-based machine, each transputer being equipped with a computational accelerator, the i860 [33]. We used an eight-module machine. In our system, the transputers run at a nominal 30 MHz, the superscalar i860 at 50 MHz and the transputer communication links were set at 20MHz. Figure 1 shows the physical arrangement of processors as well as existing system software which includes servicing of run-time I/O. An intermediary process acting as the interface between transputer and i860 by means of shared-memory could conveniently also be used as a monitor process. On other transputer-based machines a monitor is typically provided by reversing the direction of communication channels and triggering traces by the `alt` indeterminate command [22]. The central synchronization server was run at high priority, as were the local time servers, so as to provide a rapid response. The central synchronization server also provided local time for its node. High-priority processes on the transputer interrupt low-priority processes and run until blocked by communication or they deschedule themselves (by being put on a timer queue). As by implication high-priority processes do not interrupt themselves, other high-priority processes should be of short duration. (We did not take special measures to reduce the normal latency of high-priority scheduling which might be considered for real-time applications [22].) The manufacturer's figures give a high-priority latency between 3.9 and 2.9 μs were the transputers to be run at only 20 MHz. The organization of the Paramid was not ideal in this respect because, as the i860 ran more quickly than the transputer, it was necessary to give high priority to any process servicing the i860. For the purposes of the trace, this did not affect message ordering. Because of the

limited numbers of available processors, one of the nodes was placed in one test on the same processor as the synchronization node. This should have distorted the results as the minimum journey time was less than the recorded one hop time, but in practice the message ordering on the visualizer was not affected.

The Paramid usually employs a virtual channel system but the underlying hardware is dependent on store-and-forward communication. Figure 3 is the result of measurements on an empty system when it will be seen that the round-trip message time distribution is indeed long-tailed, though the lower 'hump' may be caused by the low-priority process time-slice interrupt (every 1024μs) which makes some readings appear longer. The initiation algorithm was run before the system settled into normal working, avoiding extraneous calculations in later stages. The clocks were corrected in the initial stage, introducing an element of convergence into the estimate of the refresh time.

6.2 Results

A crude measure of overhead was made on the central node by recording (without clock drift adjustment) the time taken for synchronization, recording the trace and supplying a local clock. The overhead on other nodes will naturally be less. The timing breakdowns are given in Table 1, which is for 1000 round-trip messages of approximately 200 bytes each. Note that the results in the table are consistent since in some cases, discounting the first four pulses at 0.1 second intervals, the synchronization manager was halted just before making a last pulse. The overheads recorded refer to a particular total job size regime, as Figure 4 shows that the fixed overhead becomes a much greater burden for shorter job sizes. The communication overhead, aside from synchronization, was between 6% and 9%. The number of synchronization pulses includes those made mostly over shorter intervals at initiation time. The number of messages is restricted so that the trace file can be held entirely in main memory. Though no work, and a round-robin distribution of work with constant intervals were tried, in this instance truncated and random quantized Gaussian distributions were employed for the workloads (with mean per node work time equal to the variance). Experience of testing showed

that using a Gaussian distribution was more likely to find errors in recording message times, though once the appropriate parameters had been set no errors occurred. We made use of an existing post-mortem visualizer, Paragraph [34, 35], which unfortunately does not include a tool to display tree topologies explicitly. Nevertheless, with the space-time diagram and using our trace collection system we were able to examine differing characteristic behaviour of: a single farm (Figure 5); a handwritten postcode recognition application involving three farms (Figure 6); a hybrid video encoder with three logical farms (Figure 7); and a face identification application involving three farms (Figure 8).

The sampling interval was set to 0.1 s, the clock drift to 10^{-5} and the minimum one hop journey time for 1 byte was found to be 6.0×10^{-5} s. With these parameters the clocks became accurate after four synchronization pulses during an enforced quiescence for the eight processors used. By accurate is meant that Paragraph did not report any errors for the test runs. Fewer initial synchronization pulses always led to ordering errors. The maximum error for varying round-trip times can be calculated from equation 9 using the measured minimum round-trip time, 6.0×10^{-5} s, a clock-drift of 10^{-5} s and a clock resolution of 10^{-6} s. With a measured net bandwidth of approximately 16 MHz on the Paramid and with the diameter of the network only three, the maximum error for our set-up is well within a 0.5 ms boundary. After four timings, the refresh period was set once and for all. It would probably be too disruptive to re-sample at a later time and would add to the complexity of the software. One must certainly not use a recursive method of finding the refresh times (by using the last refresh time as a sample interval) as this would lead to errors spiralling upwards.

7 Conclusion

If parallelization is to be used more commonly in engineering systems design, standard design methodologies will be necessary. For continuous-flow embedded systems this methodology can be based on the Pipeline Processor Farm model. The essential feature of the model is a number of farms each of which will have a central farmer process and subsidiary worker

processes. Provision of the model will depend on integral support of performance evaluation tools for the particular application. A trace tool is one component of the toolkit and this paper has described the design of the clock synchronization sub-system for that tool. The detailed design is a compromise between achieving accuracy and avoiding perturbation of the parallel application. At least three causes of perturbation can be isolated:

- excess number of messages;
- disturbance of the pattern of communication;
- excess calculations to achieve the synchronization.

In this design, by using one centrally-determined refresh time, the disruption to the pattern of communication is limited to one regularly-occurring pulse. The selection of the refresh time is assisted by discarding inaccurate samples by a probabilistic criterion. Specifically, averaging by a median is used to find an estimate of real time. Local adjustments for the estimated local clock drift between refresh pulses reduce the central computation as does the selection of an iterative method of median calculation.

For the class of algorithms of concern to us, targets for throughput and/or latency through the parallel system usually exist. If the targets are met while the clock synchronization is in place then there may be no need to remove the monitoring apparatus from the production model. The visualization would then represent a true reflection of the system giving confidence in its correct working. The case study has shown that for jobs of a suitable duration on the Paramid the time overhead from synchronization is a small percentage in time. In other circumstances, one should consider perturbation analysis [36] after the event trace has been collected. In fact, future work will investigate the post-processing of traces in order to provide input to an analytic (queueing) model.

Acknowledgement

This work was carried out under EPSRC research contract GR/K40277 'Portable software tools for embedded signal processing applications' as part of the EPSRC PSTPA (Portable Software Tools for Parallel Architectures) directed programme.

References

[1] A. Çuhadar and A. C. Downton. Structured parallel design for embedded vision systems: An application case study. In *Proceedings of IPA'95 IEE International Conference on Image Processing and Its Applications*, pages 712–716, July 1995. IEE Conference Publication No. 410.

[2] M. N. Edward. Radar signal processing on a fault tolerant transputer array. In T.S Durrani, W.A. Sandham, J.J. Soraghan, and S.M. Forbes, editors, *Applications of Transputers 3*. IOS, Amsterdam, 1991.

[3] S. Glinski and D. Roe. Spoken language recognition on a DSP array processor. *IEEE Transactions on Parallel and Distributed Systems*, 5(7):697–703, 1994.

[4] A-M Cheng. High speed video compression testbed. *IEEE Transactions on Consumer Electronics*, 40(3):538–548, 1994.

[5] M. H. Coffin. *Parallel Programming A New Approach.* Silicon Press, Summit, NJ, 1992.

[6] A. C. Downton, R. W. S. Tregidgo, and A. Cuhadar. Top-down structured parallelisation of embedded image processing applications. *IEE Proceedings I (Vision, Image, and Signal Processing)*, 141(6):431–437, December 1994.

[7] M. Fleury, H. P. Sava, A. C. Downton, and A. F. Clark. Designing and implementing a software template for embedded parallel systems. In C. R. Jesshope and A. V. Shafarenko, editors, *UK Parallel'96*, pages 163–180. Springer, London, 1996.

[8] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instrumented communication library. Technical report, Oak Ridge National Laboratory, Oak Ridge, TN, USA, August 1990. Report ORNL/TM-11616.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[10] M. Raynal and M. Singhal. Capturing causality in distributed systems. *IEEE Computer*, pages 49–56, February 1996.

[11] L. J. Levrouw and K. M. R. Audenaert. Minimimizing the log size for execution replay of shared-memory programs. In B. Buchberger and J. Volkert, editors, *CONPAR'94-VAPP VI*, pages 76–87. Springer, Berlin, 1994. Lecture Notes in Computer Science Volume 854.

[12] L. Lamport and Melliar-Smith P. M. Synchronizing clocks in the presence of faults. *Journal of the Association for Computer Machinery*, 32(1):52–78, January 1985.

[13] K. G. Shin and P. Ramanathan. Clock synchronization of a large multiprocessor system in the presence of malicious faults. *IEEE Transactions on Computers*, 36(1):2–12, January 1987.

[14] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[15] U. de Carlini and U. Villano. A simple algorithm for clock synchronization in transputer networks. *Software—Practice and Experience*, 18(4):331–347, April 1988.

[16] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2-3):190–204, 1984.

[17] A. Duda, G. Harrus, Y. Haddad, and G. Bernard. Estimating global time in distributed systems. In $7^{\underline{th}}$ *International Conference on Distributed Computer Systems*, pages 299–306. IEEE, 1987.

[18] W. Obelöer, H. Willeke, and E. Maehle. Performance measurement and visualization of multi-transputer systems with DELTA-T. In G. Haring and G. Kotsis, editors, *Performance Measurement and Visualization of Parallel Systems*, pages 119–143. Elsevier, Amsterdam, 1993.

[19] K. Marzullo and S. Owicki. Maintaining the time in a distributed system. *ACM Operating System Review*, 19(3):44–54, July 1983.

[20] N. W. Rickert. Non byzantine clock synchronization — a programming experiment. *ACM Operating System Review*, 22(1):73–78, January 1988.

[21] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8):933–940, August 1987.

[22] U. Villano. Monitoring parallel programs running in transputer networks. In G. Haring and G. Kotsis, editors, *Performance Measurement and Visualization of Parallel Systems*, pages 67–96. Elsevier, Amsterdam, 1993.

[23] J. Jiang, A. Wagner, and S. Chanson. Tmon: A real-time performance monitor for transputer-based multicomputers. In D. L. Fielding, editor, *Transputer Research and Applications 4*, pages 36–45. IOS, Amsterdam, 1990.

[24] T. H. Dunigan. Hypercube clock synchronization. *Concurrency: Practice and Experience*, 4(3):257–268, May 1992.

[25] R. Cole and C. Foxcroft. An experiment in clock synchronization. *The Computer Journal*, 31(6):496–502, 1988.

[26] E. Maillet and C. Tron. On efficiently implementing global time for performance evaluation on multiprocessor systems. *Journal of Parallel and Distributed Computing*, 28(1):84–93, 1995.

[27] D. L. Mills. On the accuracy and stability of clocks synchronized by the Network Time Protocol in the Internet system. *ACM Computer Communication Review*, 20(1):65–75, January 1980.

[28] D. L. Mills. Internet time synchronization: the Network Time Protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, October 1991.

[29] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. C.U.P., Cambridge, UK, $1^{\underline{st}}$ edition, 1988.

[30] D. E. Knuth. *The Art of Computer Programming*, volume 3/Sorting and Searching. Addison-Wesley, Reading, MA, 1973.

[31] F. Cristian. A probabilistic approach to distributed clock synchronization. In *Proceedings of the Ninth IEEE International Conference on Distributed Computer Systems*, pages 288–296, June 1989.

[32] Transtech Parallel Systems Corporation, 20, Thornwood Drive, Ithaca, NY, USA. *The Paramid User Guide*, 1993.

[33] M. Atkins. Performance and the i860 microprocessor. *IEEE Micro*, pages 24–78, October 1991.

[34] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.

[35] I. Glendinning, V. S. Getov, S. A. Hellberg, and R. W. Hockney. Performance visualisation in a portable parallel programming environment. In G. Haring and G. Kotsis, editors, *Performance Measurement and Visualization of Parallel Systems*, pages 251–275. Elsevier, Amsterdam, 1993.

[36] A. D. Maholy, D. A. Reed, and H. A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

List of Tables

List of Figures

Wall-clock Time	Sync. Interval	No. of sync. timings	Overhead from Syncs.	Overhead from Serving time	Overhead from Making trace	Total % overhead	Mean work time
16.641	3.197	9	0.013	0.311	0.505	4.987	0.1
32.125	3.197	13	0.063	0.316	0.503	2.745	0.2
47.452	3.197	18	0.145	0.316	0.504	2.034	0.3
109.040	3.197	37	0.097	0.294	0.505	0.822	0.7
155.096	3.197	52	0.183	0.308	0.502	0.640	1.0
309.957	3.197	100	0.321	0.306	0.504	0.365	2.0

Table 1: Timings (s) Recorded on the Central Node Using the Local Clock

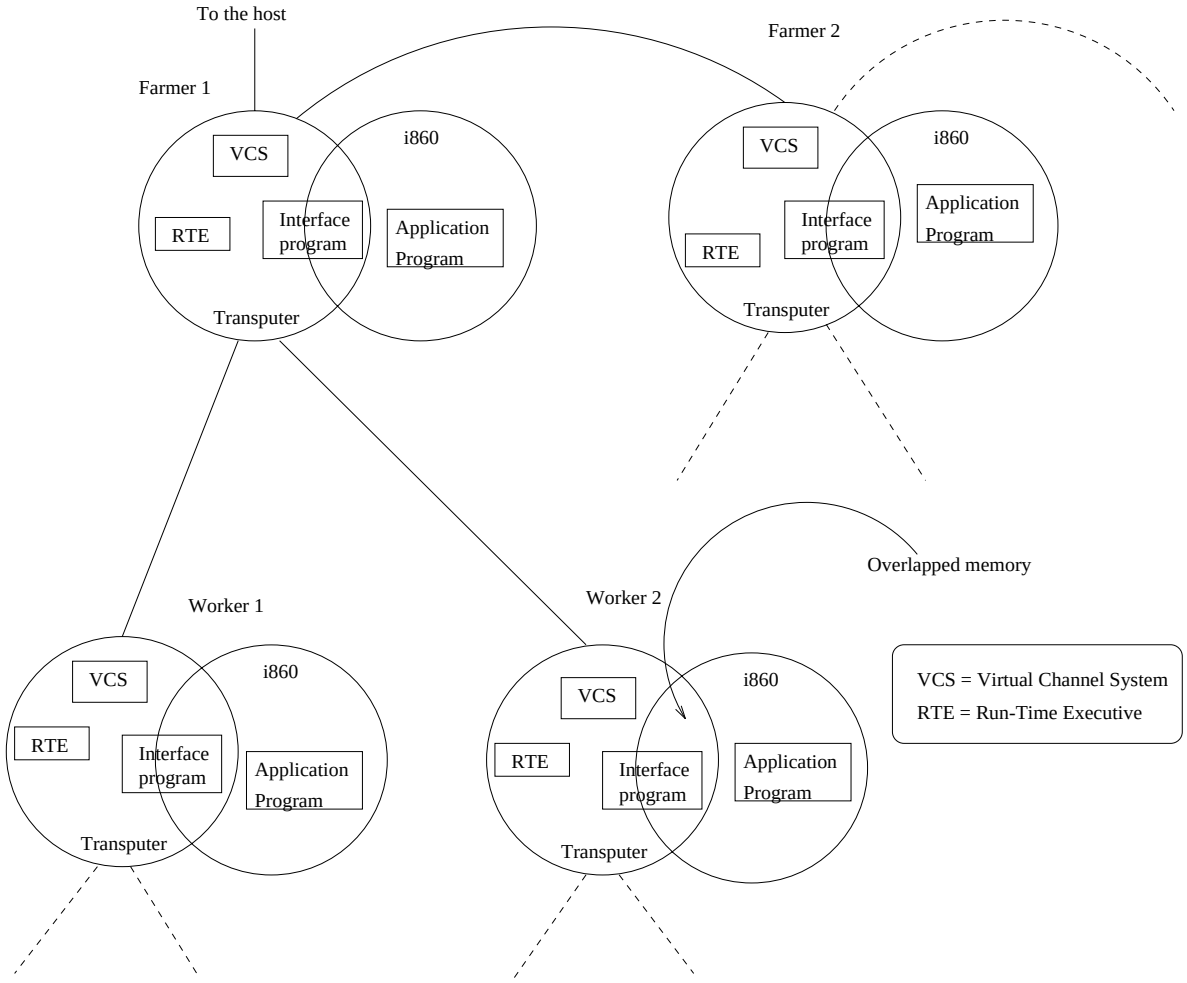

Figure 1: System Set-up on the Paramid with a PPF Arrangement

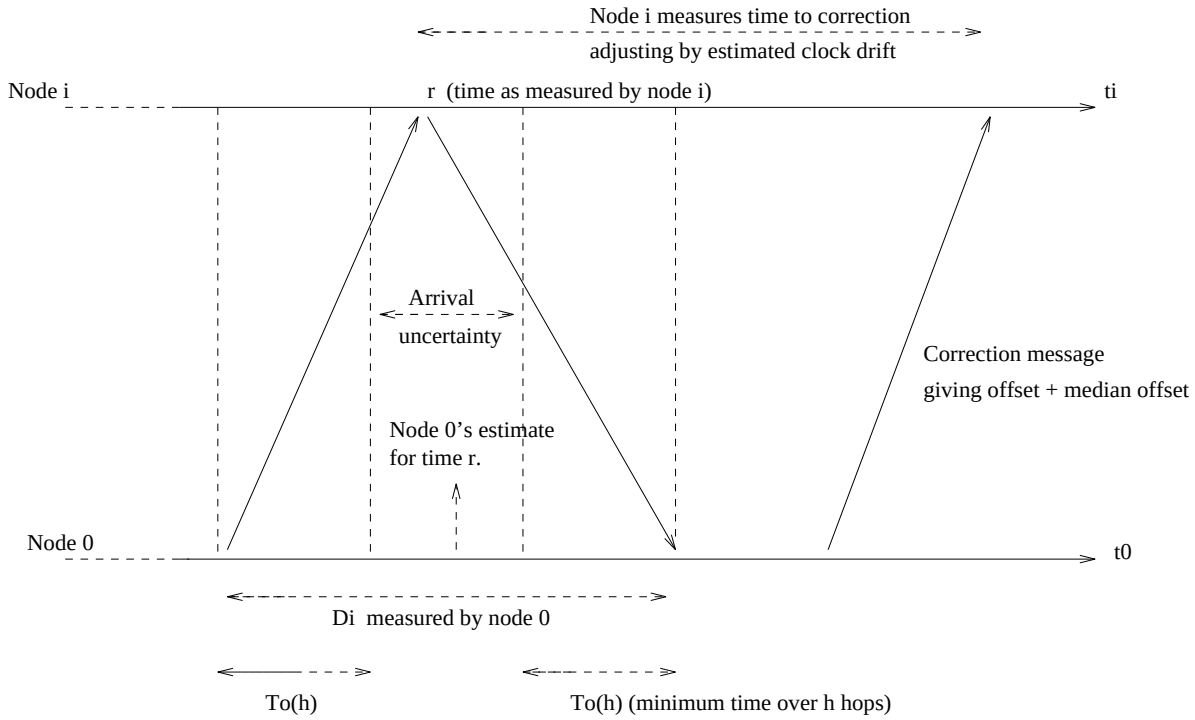

Figure 2: Synchronization Signal Pattern

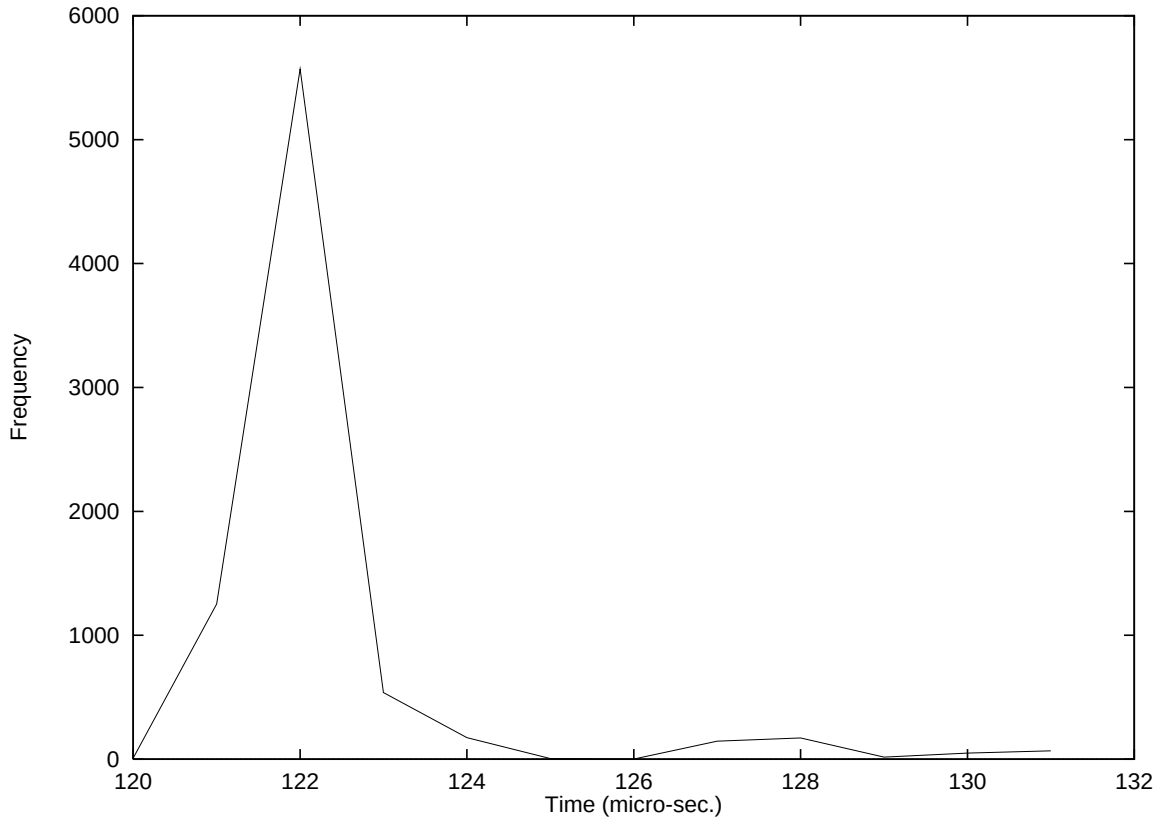

Figure 3: Frequency of Timings (μs) for One Hop (Round-Trip) on the Paramid

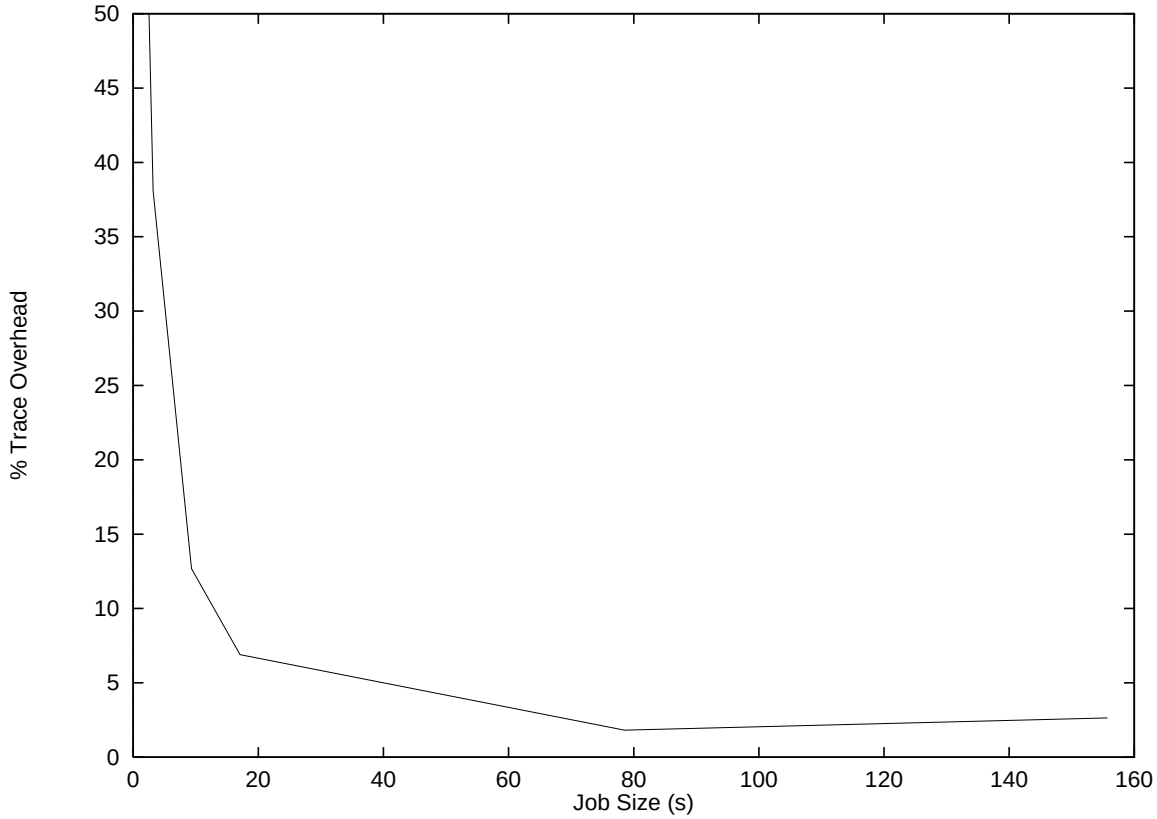

Figure 4: Overhead from Tracing on the Paramid

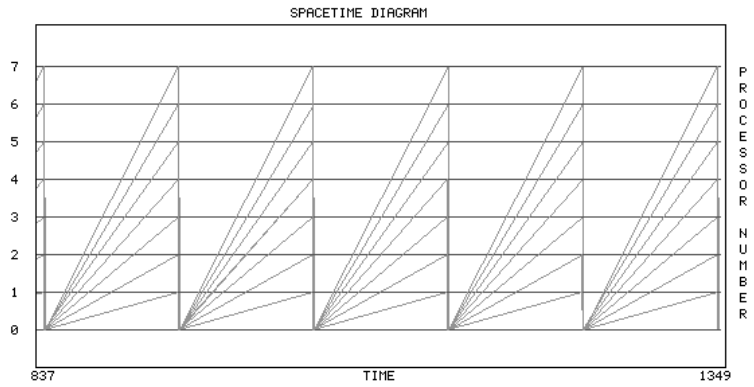

Figure 5: Display for a Seven Worker Farm

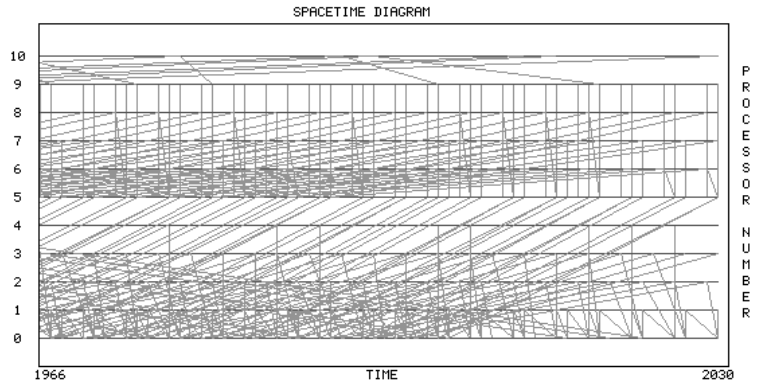

Figure 6: Display of the Postcode Application

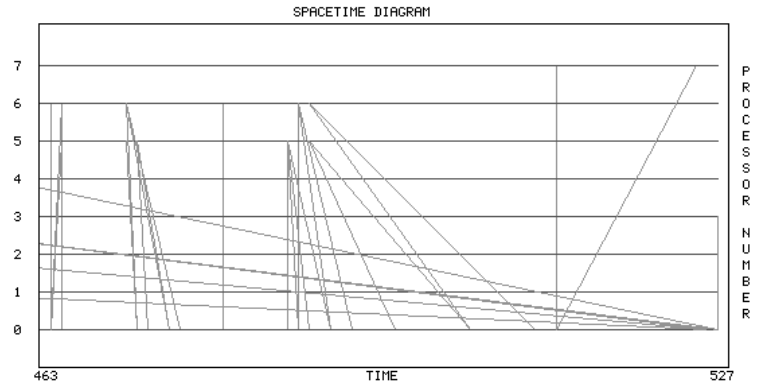

Figure 7: Display for H.263

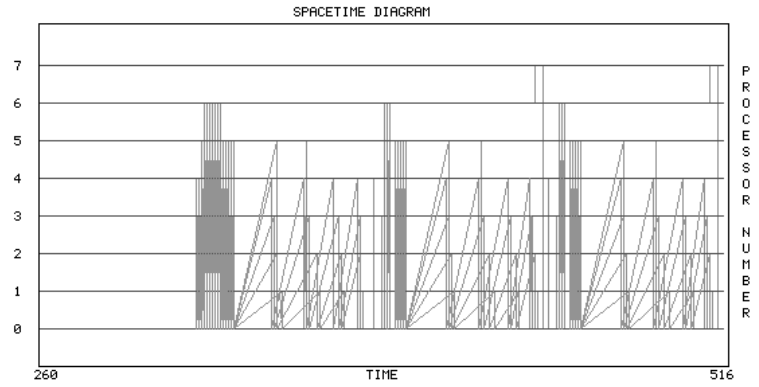

Figure 8: Display for Face Identification Application