[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, Maryland, 1986.

[Pea84] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, 1984.

[PR88] P. Peinl and A. Reuter. High contention in a stock trading database: A case study. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data, Chicago*, pages 260–268, June 1988.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Theory.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[Ram92] K. Ramamritham. Real-time databases. *International Journal on Parallel and Distributed Databases*, 1992. Invited paper.

[SL90] X. Song and J. W. S. Liu. Performance of multiversion concurrency control algorithms in maintaining temporal consistency. Technical report, University of Illinois at Urbana-Champaign, 1990.

[SLKS92] N. R. Soparkar, E. Levy, H. F. Korth, and A. Silberschatz. Adaptive commitment for real-time distributed transactions. Technical Report TR-92-15, The University of Texas at Austin, Computer Sciences Department, April 1992.

[Son88] S. H. Son, editor. *ACM SIGMOD Record: Special Issue on Real-Time Databases.* ACM Press, March 1988.

[Sop93] N. R. Soparkar. Time-constrained transaction management. Ph.D. dissertation. Department of Computer Sciences, University of Texas at Austin. In preparation, May 1993.

[SRL88] L. Sha, R. Rajkumar, and J.P. Lehoczky. Concurrency control for distributed real-time databases. *ACM SIGMOD Record*, 17(1):82–98, March 1988.

[SSK92] N. R. Soparkar, A. Silberschatz, and H.F. Korth. Serializability among autonomous transaction managers. Submitted for publication, December 1992.

[Sta88] J. A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, pages 10–19, October 1988.

[HCL90] J. R. Haritsa, M.J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the nineth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Nashville*, pages 331–343, April 1990.

[HMR+89] R. Holte, A. K.-L. Mok, L. Rosier, I. Tulchinsky, and D. Varvel. The pinwheel: A real-time scheduling problem. In *Proceedings of the 22nd Hawaii International Conference on System Sciences, Kailua-Kona*, pages 693–702, January 1989.

[JLT85] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the Eleventh Real-Time Systems Symposium, Lake Buena Vista, Florida*, December 1985.

[JM86] F. Jahanian and A. K.-L. Mok. Safety anaylsis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[KLS90] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane*, pages 95–106, August 1990.

[KM92] T.-W. Kuo and A. Mok. Concurrency control for real-time database management. In *Proceedings of the Thirteenth Real-Time Systems Symposium, Phoenix, Arizona*, December 1992.

[Kri82] R. Krishnamurthy. Concurrency control and transaction processing in a parallel database machine environment. Ph.D. dissertation. Department of Computer Sciences, University of Texas at Austin, December 1982.

[KS88] H. F. Korth and G. Speegle. Formal model of correctness without serializability. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data, Chicago*, pages 379–388, June 1988.

[KSS90] H. F. Korth, N. R. Soparkar, and A. Silberschatz. Triggered real-time databases with consistency constraints. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane*, August 1990.

[LJJL92] K.-J. Lin, F. Jahanian, A. Jhingran, and C. D. Locke. A model of hard real-time transaction systems. Technical Report RC No. 17515, IBM T.J. Watson Research Center, January 1992.

[Pap79] C. Papadimitriou. Serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

actions requires that several operations be scheduled together. To do so, we have shown how the existing algorithms from scheduling theory may be used in our context. Finally, we have addressed the question of uncertainties in the timing characteristics of transaction executions by describing a probabilistic variant of our model. In several cases, details have been omitted for the sake of brevity.

References

[AAS91] D. Agrawal, A. El Abbadi, and A. Singh. Consistency and orderability: Semantics-based correctness criteria for databases. Technical report, University of California, Santa Barbara, 1991.

[AGM88] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proceedings of the Fourteenth International Conference on Very Large Databases, Los Angeles*, pages 1–12, 1988.

[AGM89] R. Abbott and H. Garcia-Molina. Scheduling real time transactions with disk resident data. In *Proceedings of the Fifteenth International Conference on Very Large Databases, Amsterdam*, pages 385–396, 1989.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[BMHD89] A. P. Buchmann, D. R. McCarthy, M. Hsu, and U. Dayal. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In *Proceedings of the Fifth International Conference on Data Engineering, Los Angeles*, pages 470–480, February 1989.

[C+89] S. Chakravarthy et al. HiPAC: A research project in active, time-constrained database management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, July 1989. Final report.

[Cof76] E. G. Coffman, editor. *Computer & Job/Shop Scheduling Theory*. J. Wiley & Sons, New York, 1976.

[DLK81] M. A. H. Dempster, J. K. Lenstra, and A. H. G. Rinnooy Kan, editors. *Deterministic and Stochastic Scheduling*. D. Reidel Publishing Company, Boston, 1981.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, New York, 1979.

main memory). Hence, the assumption that O_i and O_{i+1} have independence with respect to their execution times does not hold, and an analysis of the type described above will not necessarily yield good schedules. That is, additional information with regard to data placement etc., can be brought to bear, and thereby, provide a more accurate analysis.

The additional information regarding the underlying system and the applications, in general, does provide a better handle for scheduling transactions. The problem is that the analyses becomes even more difficult (i.e., considering the intractability results discussed above, more involved algorithms appear to be counter-productive). Furthermore, the details of data placement, or other underlying factors, are very difficult to obtain — especially as they may change dynamically. Our goal is simply to point out the role played by the estimated values, and the fact that if needed, our approach can handle additional information as well.

We do not detail the description of our approach in this paper (see [Sop93]), except to provide the following brief outline. The key idea is to associate a probability distribution for each estimated weight function time whenever the precise timing is unknown. Note that this implies a *lack* of sufficient information to obtain the exact timing. Thus, in the case that added information is obtained for a particular situation, the probability distribution can be discarded, and the exact values used in its place. That is, the uncertainty may be counteracted by having more information.

The effect of using probability distributions is to create several more alternatives from which to choose the best schedule. That is, a single schedule, as described in this paper, may give rise to several alternatives depending on the distributions. Thus, in many ways, this approach is similar to allowing a larger set of acceptable schedules.

12 Conclusions

We have proposed a model for time-constrained transaction processing based on time associated with the transactions and their schedules. We have demonstrated that, in general, finding strategies for the scheduling problems that arise in this context, is computationally intractable. This negative result does not preclude the practical use of our model. Rather, it indicates that heuristics or other suitable "protocols" are required for time-constrained transaction processing. An analogous situation exists for standard transaction processing, where the set of two-phase locked schedules is usually accepted as a suitable subset of the set of serializable schedules whose recognition problem is NP-complete.

We have suggested some approaches toward the practical usage of our time-constrained transaction model, but many issues remain to be addressed. For example, good heuristics for the correct scheduling of transactions with an acceptable level of performance, are needed. As we have explained in this paper, achieving good results for the dynamic scheduling of time-constrained trans-

scheduling. The inherent trade-offs introduced by such choices, and the formal description of the problem of real-time optimization are not provided in this paper (see [Sop93]).

11 Dealing with the Unknown

We discuss some issues regarding uncertainties with respect to the timing estimates, and the role of additional information in this context. Note that we assume that the estimated values are obtained by some means — experimental, or otherwise. The actual values may vary, and we consider how this may be handled in our model. In the following discussion, we use a schedule that is a sequence (i.e., a total order), and we are concerned with a particular subsequence of n operations. The subsequence, denoted by s, is represented as $O_1, O_2, \ldots, O_n$, where an operation O_i precedes the operation O_{i+1} in the schedule, and we are we are not concerned with the transactions *per se*. We assume that the estimated execution time (i.e., weight function) for each operation O_i is 10 time units, and that no real time elapses between any two consecutive operations.

As an example for discussion, assume that the estimated values are obtained by experiments that measure the realized weight function for operation O_1 to be 1 time unit nine out of ten trials. Also, one out of ten trials measures the time taken by O_1 to be 91 time units. This could happen, for instance, if a disk access were initiated — i.e., the high value being due to the time taken to process a page fault when the data entity being accessed was not in main memory (and with the I/O being very efficient!). This provides the estimated value of 10 time units for O_1 on an average.

However, note that s would have the total makespan of $10(n-1)+1$ about nine times out of ten, and $10n$ one out of ten times — assuming all the other operations take their estimated time for execution. Similarly, in the case that each operation is similar to O_1 in terms of its estimated execution time, it should be clear that assuming that the time taken for each operation is independent of the others, the makespan for s could be n time units 9^n out of 10^n times.

The corresponding values accrued for the same legal schedule would differ with the differing values for the weight functions of each operation. The approach to take then is obviously the selection based on the average expected value accrued for a particular legal schedule for the criterion of maximizing the value accrued.

The above shows how a single legal schedule actually produces several alternatives in terms of values accrued with a detailed examination for the timing. The underlying assumption is that the trials for two operations are independent, which is only an approximation. In reality, this may not be true, as shown below.

Consider the possibility that an operation O_i in s accesses the same data entity as the operation O_{i+1}. In that case, the chances that there will occur a page fault due to the latter operation directly after the execution of the former, are negligible (since the required data would already be in the

the difficulty encountered in proving concrete performance bounds for many sub-optimal algorithms accords them the label of a heuristic.

10.2 Real-Time Optimization

Whether or not the optimization problems that are encountered in scheduling of transactions are tractable, there is an interesting general optimization problem that emerges. It is unlike the case in typical optimization problems where the idea is to find an efficient algorithm that scales-up well with larger instances of the problem, and produces the best, or at least good, solutions as soon as possible. Instead, the new problem is complementary to the typical optimization problem. The goal becomes the finding of as good a solution as possible, for a given instance of the problem, within a particular time bound. The time bound usually arises from the application at hand, and often, there is some flexibility in deciding what instance size may be considered for one run of the algorithm.

Notice that the problem for TCTM is the selection of schedules that are good in terms of performance among the set of legal schedules. In this context, the availability of liberal scheduling criteria (e.g., VSR instead of CSR) provides a larger set of legal schedules, and hence, a higher likelihood of finding schedules with good performance. However, simply having more legal schedules available may not be useful since the time bound for searching out a good schedule may be unavailable, or the algorithms that can produce the candidate schedules may be inefficient.

Consider the determination of a path that a plotter-pen must follow to plot several points of a graph. This is an instance of the *traveling salesman problem* (e.g., see [GJ79]), whose optimal solution is intractable, in that the time taken to complete a plot should be minimized since the plotter is a slow device compared to the computer. However, assuming that a dedicated scheduling processor is available, what is really needed is to ensure that a particular instance of the plotter path determination should be completed by the time the *previous* graph plotting instance is completed by the plotter. Thus, the determination of a good path has a "deadline", and also, the time of invocation of the problem instance has a "release" time (created by the arrival of the graph plotting instance).

Although the above example is not precisely in TCTM, it illustrates the utility of a *real-time* optimization study. It should be noted that while the example is not from a real-time systems domain *per se*, the optimization problem itself is. That is, the best possible solution to a problem instance is desired subject to a particular time bound.

The choice of the problem instance size may be governed in a TCTM environment as follows. Instead of considering all the available operations to be scheduled for a given instantiation of the scheduler, a subset, whose size is decided by the dictates of the time bound, may be chosen for

10.1 Seeking Alternative Strategies

Alternative strategies that may be sought are now discussed (e.g., see [PS82]). Again, note that our discussion is only representative, and is in no way comprehensive. The alternatives that may be used should be applied to the batch-online transactions scenarios that are the most general case as explained above. Providing examples for each instance is not within the scope of this paper (see [Sop93]), but such examples can be easily constructed from published results in scheduling theory (e.g., see [DLK81]).

- Approximation algorithms: These algorithms produce solutions that are a fixed percentage away from the optimal solutions. As such, they are the best for providing guarantees (e.g., for real-time scenarios) as far as the values accrued by the schedules — barring inaccuracies in the timing estimates. The analysis required to prove the lower bounds on the values accrued may often be non-trivial.

- Probabilistic algorithms: Assuming some probabilistic distribution of the problem instances, these attempt to produce solutions that are not "too" bad, and their computation is not "too" inefficient. The problem in transaction processing applications may be that such a distribution of the input parameters is difficult to obtain, particularly in online situations.

- Special cases: If special cases can be isolated that are relevant for particular applications, then the intractability results for the general problem are irrelevant. In fact, most of the available techniques in scheduling theory fall into this category (e.g., see [DLK81]).

- Exponential algorithms: Algorithms that are *pseudo*-P-time may be exponential time algorithms, but they are effectively P-time for practical situations. Furthermore, there may be branch-and-bound, or dynamic programming, approaches that may actually serve well for small-sized problems. If, especially in the batch-online transactions case, the scheduling algorithm is invoked often enough, and executes on a dedicated processor, this alternative may not be too unattractive.

- Local search: Approaches such as neighborhood search techniques (i.e., greedy techniques) are relevant to attack combinatorially difficult problems. A large body of literature, especially from the domain of artificial intelligence (e.g., see [Pea84]), exists on this subject, and may be used in our context.

- Heuristics: A technique which simply produces "good" results without any guarantees with regard to the degree to which it is "good", may be regarded as a heuristic. Most of the empirical work in TCTM falls into this category (e.g., see [Ram92]), and it is the case that

- *meeting deadlines on individual operation in a PRAM machine.*

Proof: The proof is essentially the same as for the case of scheduling a single transaction.□

A large number of similar intractability results can be derived for DRAM machines in the context of batch-online transactions for CSR schedules since the essentially intractable problems in areas such flow-shop and job-shop scheduling involve the assignment of tasks to specific processors.

9.4 Scheduling with Aborts

So far we have assumed that the optimality results are sought without taking recourse to the aborting of transactions. However, except for the cases of minimizing tardy transactions or operations, all the intractability results hold even *with* the option of aborting transactions. We direct attention to the fact that in cases where minimization of transactions, or operations, that will miss their deadlines, is concerned, the additional possibility of aborting the tardy executions exists. Such aborts may increase the chances for improving the overall performance given that the executions that do not contribute sufficiently to the value accrued for the schedule are eliminated. In fact, in the batch-online transactions case, other operations belonging to a transaction that are available to be scheduled at that point. This improves the availability of the shared resources for the purposes of scheduling other executions that *do* contribute to the value accrued. A more detailed discussion of scheduling when aborts are available is not included in this paper (see [Sop93]).

The importance of aborts becomes very evident when the time estimates, heretofore assumed to be accurate, are found to be undependable. In such cases, a scheduled operation may take inordinately long to execute, and then it becomes imperative to abort the rest of the transaction. Otherwise, the entire transaction must be unnecessarily executed without contributing to the value accrued — especially if it is a transaction that has a deadline. Hence, it is necessary in the practical scenarios where precise estimates on execution times are unavailable that the facility to abort transactions internally be available to the scheduler.

10 When the Going Gets Tough

The search for optimality led to intractability results. Thus, the approach to take then is to attempt alternatives, or to examine the problem in a different manner. This paper is not concerned so much with seeking alternatives for particular scheduling problems as much as with adapting existing techniques to the transaction management problem. Again, unfortunately, the state-of-the-art techniques available in scheduling, especially for multiprocessors, are meager. A recourse often taken is to attempt *ad hoc* heuristics, and to attempt to study the problems from an empirical standpoint (e.g., see [JLT85, HCL90]).

Since it is the case that the restricted and the less restricted transactions scenarios are restrictions of the batched-online situation, we note that optimal results for the cases considered above hold for the batched-online case as well. Moreover, note that an instance of a problem in the batched-online case where there is no conflicting accesses between the transactions, is exactly the same as a restricted transactions scenario for the purposes of deriving intractability results.

Let us consider an interesting situation that imposes a fixed partial order $<<$ on the set of n operations as a result of the realized schedule. Note that CSR schedules require that conflicting accesses between transactions must be ordered in the same manner. Consider two conflicting operations, $O_i^{'}$ and $O_j^{'}$, from transactions T_i and T_j, respectively, such that in the realized schedule, $<$, it is the case that $O_i^{'}$ precedes $O_j^{'}$. In that case, a partial order relationship is necessary between two conflicting operations, O_i and O_j, from the transactions T_i and T_j, respectively, created by $<<$, if the expected completions are to be legal CSR schedules. That is, a fixed precedence relationship $<<$ is created for each batch-online problem instance due to the realized schedule. Such precedence requirements can arise for other logical correctness criteria as well.

The above observation is important since it not only allows us to recognize certain problems as being intractable, but more importantly, it allows the use of heuristic approaches developed in the context of precedence constrained scheduling. To utilize tractable algorithms, the $<<$ partial order must also be made to include the $<_i$ partial order for each transaction T_i in the schedule, for reasons that are obvious. Furthermore, we need to assume that there are no conflicting operations within a transaction T_i that are not ordered by $<_i$. With the above explanation, it becomes clear that the problem is similar to the situation for scheduling a single transaction, and thus, we may state the following results.

Theorem 9. *There exists an efficient algorithm to schedule a set of batch-online transactions operations such that the throughput of operations is maximized for CSR schedules.*

Proof: The requisite algorithm is the same as that for the scheduling of a single transaction to maximize throughput.□

Of course, there do remain the large number of intractable problems. The following are a representative set of such results.

Theorem 10. *The following scheduling problems are intractable in the context of batch-online transactions for CSR schedules.*

- *minimizing tardy operations, and weighted tardiness, on a RAM machine, with individual deadlines on the operations.*
- *maximizing the throughput of operations with unit weight functions on PRAM and DRAM multiprocessors.*

Not surprisingly, similar problems for the multiprocessor cases are also intractable. In fact, the uniprocessor cases may be regarded as special cases of the multiprocessor scenarios. Additionally, the multiprocessor scheduling issues yield further results as follows.

Theorem 7. *Scheduling a set of restricted transactions that have an overall deadline on a PRAM machine is intractable.*

Proof: The proof is a simple reduction from the multiprocessor scheduling problem (e.g., see [SS8] in [GJ79]).□

The restricted transactions case is possibly too restrictive, and hence, we now consider the more general case where each transaction T_i is a partial order $<_i$ on its constituent operations. We continue to disregard aborts as explained above, and we refer to this as the less restricted transactions scenario. We consider only the question of maximizing throughput in this model.

Theorem 8. *Scheduling a set of less restricted transactions on a PRAM machine with a finite or infinite number of processors so as to maximize throughput, with CSR or VSR as the logical correctness criteria, is intractable.*

Proof: (From [Kri82]) It is known that minimizing the makespan for a set of less restricted transactions is intractable.□

9.3 Scheduling Sets of Operations Online

Now we consider a more general case wherein a set of transactions must be scheduled, but neither are all the transactions, nor are the operations of each, known *a priori*. Thus, the scheduling problem becomes one where a set of n operations are outstanding, and there is a need to schedule them. We briefly discuss issues regarding the frequency of the scheduling, and the manner in which the n operations are obtained, in Section 10. We have discussed above how the value functions are assigned to the operations, and these are derived from the value functions assigned to the transactions themselves. An interesting observation in such cases is that deadlines may occur in the context of interactive transactions wherein after an operation is invoked, a deadline specific to that operation is created, and such a situation is also represented in the scenario under study.

Clearly, this is a situation where a set of operations is scheduled online. It is realistic in that it lies in-between the optimistic extreme of assuming that all the transactions and operations are known *a priori* (e.g., see [Kri82]), and the pessimistic extreme of assuming that only one operation is outstanding at a time (e.g., see [Pap79]). Hence, we refer to such scenarios as *batched-online* transactions scheduling. The legal schedules are taken to be CSR or VSR, as indicated. Finally, as above, we shall consider that aborts are disregarded.

scheduling of a single transaction. To state these more rigorously, the model in Section 2 needs to be extended to include value functions for individual operations, and hence, we do not explore it further.

9.2 Sets of Transactions

It may be argued that a single transaction, especially with all its operations known *a priori*, does not represent a reasonable scheduling problem. We now make the problem more realistic by considering sets of transactions, each known *a priori*, that need to be scheduled to maximize performance. In this case, note that it is always possible to create legal schedules for the transactions such that each is committed. We assume that a non-zero penalty accrues for an abort, and the value function for committed transactions is always non-negative. This effectively ensures that we do not need to deal with aborted transactions. Examples of situations where such sets of transactions are known *a priori* include well-studied scenarios (e.g., in a manufacturing environment), or for a particular *period* in a periodic problem in real-time systems.

We specify the legal schedules to be those that are CSR and VSR — similar intractability results occur in each case. The size of the problems that are considered is n where n is the total number of operations arising from all the transactions in the set to be scheduled. For the other features in the model considered above, we state that $m = m'$, and $<<$ is empty.

First, we consider the simple case where each transaction T_i has exactly one operation O_i other than the commit or abort operation. The weight function estimates are provided as discussed for the results discussed above. Also, we assume that the commit operation has a weight function of 0 (e.g., imagine a main-memory architecture that is battery backed-up so that no explicit commitment is necessary). In such a situation, any partial order of the operations O_i constitutes a legal schedule obeying CSR. We refer to this situation as the scheduling of *restricted* transactions. It should be noted that this situation is exactly one of an independent set of tasks for the purposes of obtaining intractability results.

Theorem 6. *the problem of scheduling a set of restricted transactions on a RAM machine to:*

- *meeting transaction release times and deadlines,*
- *maximize the value accrued for transactions with deadlines, and values before the deadlines differ among the transactions,*
- *minimize weighted tardiness of transactions with deadlines,*

is intractable for each of the above.

Proof: Simple reductions from essentially the same problems in sequencing theory (e.g., see problems [SS1,SS3,SS7] in [GJ79]) exist.□

execution time limit. The task lengths map to the weight function on the operations, and the overall deadline maps to an overall execution time limit. Since there is no precedence constraint on the tasks for the multiprocessor scheduling problem, the partial order $<_i$ is regarded to be empty.□

In the presence of a non-empty partial order $<_i$, the situation is even more grim.

Theorem 3. *The scheduling of a single transaction with unit weight functions on the operations on a PRAM machine so as to maximize throughput is intractable.*

Proof: An instance of the precedence constrained scheduling problem (e.g., see [SS9] in [GJ79]) can be reduced to our problem. The precedence constraint on the tasks maps to the partial order $<_i$.□

However, the above problems may be solved tractably for some special cases (e.g., see [Cof76, GJ79]).

The situation for a single transaction scheduled on a DRAM is also bad.

Theorem 4. *The scheduling of a single transaction with unit weight functions on the operations on a DRAM machine so as to maximize throughput is intractable.*

Proof: An instance of the precedence constrained scheduling problem (e.g., see [SS9] in [GJ79]) with each task mapped to a specific processor, can be reduced to our problem. To create a situation that such a mapping is required in our problem, let each operation access a different data entity. Hence, if a task t is mapped to a processor p, then the corresponding situation in our problem is to assume that the data entity accessed by the operation corresponding to t resides in the processor-memory pair corresponding to p.□

Essentially the same results hold for the question of scheduling a single transaction with a deadline constraint (i.e., a value function that is constant, say 1, up to the deadline point in real time, and 0 thereafter), and with a non-zero penalty associated with the abort (i.e., aborting is not an option given that late commitment does not incur a penalty). Let us simply call this a "deadline-constrained" scheduling problem.

Theorem 5. *The deadline-constrained scheduling of a single transaction on PRAM and DRAM multiprocessors is intractable.*

Proof: Simple reductions similar to the proofs for Theorems x and y above from the same intractable problems in multiprocessor scheduling.□

Consider a situation where there are individual deadlines on the operations of a single transaction which may occur when transactions are used in the design of real-time systems. This is because a transaction simply captures a logically coherent set of tasks to be executed, and these tasks, which can also be viewed as operations, may have individual deadlines. They may have a partial order of execution as well. In such cases, there exist similar intractability results for the

use operations that span more than one processor-memory pair. Again, we regard this architecture to be a backend platform.

We shall refer to the three target architectures as RAM, PRAM, and DRAM machines. The time estimates for particular operations or transactions etc., are assumed to be independent of the workload on the systems. In other words, we assume that the systems have normal workloads for the weight function estimates, and that these estimates remain stable.

As mentioned earlier, we encounter, not unexpectedly, intractability results in the attempt to obtain optimal results. We consider the different flavors of intractable results, and consider their significance. Note that the following discussion is not meant to be comprehensive in any sense of the word; instead, it represents the difficulties encountered in attempting to obtain optimality. We do not use value functions *per se* in the discussions; instead, we use the traditional nomenclature of "deadlines" etc., where it should be understood that corresponding value functions could have also been used.

9.1 Single Transactions

Consider the scheduling questions faced when a single transaction, T_i, is known, and it is the only transaction that needs to be scheduled. In such a situation, the partial order $<<$ may be regarded as the partial order $<_i$ that represents the precedence order on n operations of T_i. Note that $<_i$ does not necessarily relate two operations within T_i that may access the same data entity in a conflicting manner. Thus, when a P-time algorithm is provided to schedule the operations in T_i on a multiprocessor, additional relationships must be defined in the $<<$ partial order. Also, assume that the weight functions for the operations in T_i are provided by estimation.

Consider the maximization of throughput in the the above scenario. Clearly, it is the minimization of the makespan of the transaction execution.

Theorem 1. *A single transaction can be efficiently scheduled to maximize throughput on a RAM machine.*

Proof: The requisite algorithm is to topologically sort the operations of a single transaction T_i, and to execute it on the uniprocessor machine in the resulting order. The makespan of the execution is the minimum possible since it is the sum of the weight functions of the operations.□

Unfortunately, as may be expected, even in this simple case, the multiprocessor scheduling problem is intractable.

Theorem 2. *The scheduling of a single transaction on a PRAM machine so as to maximize throughput is intractable.*

Proof: An instance of the multiprocessor scheduling problem (e.g., see [SS8] in [GJ79]) can be reduced to the problem of executing a single transaction on a PRAM machine with an overall

scheduling the n operations, it is not unreasonable to assume that the value function for T_i may be applied to O_i after "shifting" it by t_i in the real time domain. That is, the value function to be considered for an operation O_i is $vf(T_i)$ with the difference that the value mapped to for O_i from a real time point t is the same as the value mapped to for T_i from the real time point $t + t_i$.

It is possible to seek optimal solutions to the restricted maximization problem that is sketched above. Unfortunately, extant results from scheduling theory preclude the efficient handling of the problem. Thus, to obtain the best results, the underlying system may have to wait inordinately long periods of time as examined below.

9 The Best comes to those who Wait

We consider the issue of obtaining the optimal results in time-constrained transaction scheduling in this section, and we show that it is an intractable problem, in general. To do so, it is necessary to briefly examine the target architectures on which these results are based, and also, to explain the meaning of the optimality criteria.

We consider three major target hardware architectures in a centralized environment. As of now, the secondary storage considerations are ignored, and hence, the databases may be regarded as being entirely resident in main-memory. To account for stable storage, when necessary, the main memories may be regarded as being battery backed-up. Note that in all cases, a common clock is assumed to be available.

First, consider a uniprocessor architecture whose relevant abstract counterpart to it should be regarded as a *random access memory* (RAM) machine. Thus, the operations, or tasks, assigned to the processor are executed by accessing the main memory. There could be further enhancements to the basic architecture to allow separate processors to execute the transaction programs and the scheduling disciplines. These are not not explicitly considered, and thus, the architecture represents a backend database server platform.

Second, consider a multiprocessor architecture with shared memory. The abstractions used to study such architectures are called *parallel* RAM (PRAM) machines with variations as to their abilities to access the memory in true parallelism. We assume that there is no difference in the ease of accessing any data between the processors. As for the case of the uniprocessor, this architecture is regarded as a backend database server platform.

Third, consider a distributed memory multiprocessor architecture. The counterpart abstract machines are called *distributed* RAM (DRAM) machines. The constraint in such machines is that the data is distributed, and the dictates of efficiency require that operations, or tasks, be scheduled on the particular processor-memory pair that house the data being accessed. Also, it is inefficient to

queued at the input of a scheduler, and there is the need to schedule these in a logically correct manner while also considering the enhancement of performance criteria.

The model that we use has the following general features for a particular instance when a scheduler is invoked. There are m *active* transactions, where an active transaction is one whose invocation is known to the scheduler, and whose final commit or abort operation is not included in the realized schedule. The total number of transactions in the schedule is m'. Also, there are n operations that need to be scheduled, and all these operations belong to the active transactions. In some instances, we require that a partial order, $<<$, on the operations is imposed such that the partial order $<$ produced by the scheduler must subsume $<<$. Usually, the partial order $<<$ arises from certain logical correctness criteria or from within the partial order $<_i$ defined for an active transaction T_i, and this is discussed below. For those instances where we allow the scheduler to abort transactions at will, we also consider an *internal* abort operation — where the term "internal" is used to distinguish it from a transaction-supplied, or *external*, abort — for each active transaction which has no external abort represented in the n operations. Note that if a commit operation for an active transaction is included in the n operations, then any schedule includes only one of the abort or commit operations. Usually, there is also a need to consider the commit operations of the active transactions to study the expected completion of a schedule.

Consider the expected completion of an incomplete schedule. By providing estimates for the weight functions in the portion of the expected completion that is not part of the realized schedule, a time-annotated schedule is obtained. The value accrued for the expected completion may be regarded as the *expected* value accrued for a schedule with the completed transactions in the realized schedule, and with expected completions for the active transactions. Thus, we are in a position to restate the goal of the scheduler in TCTM environments, which is *"to maximize the value accrued for the expected completions."* That is, the goal of ensuring good performance for the transaction schedules generated, is made explicit. Note that since the value accrued for a logically incorrect schedule is infinitely negative, the preservation of consistency is subsumed in the above goal.

The scheduler is in a position to deal with the scheduling of the n operations available to it. Assume that none of the active transactions are to be internally aborted. The question then arises as to how to provide value functions for the restricted optimization problem of scheduling the n operations. To do so, let the expected value of the weight function (i.e., estimated time) from the point that a particular operation O_i for an active transaction T_i is acknowledged to have been executed, up to the point that the final operation for the transaction T_i is acknowledged to have completed, is t_i. Assume that the value t_i is the same for every expected completion — we relax this simplifying assumption in Section 11. Furthermore, assume that there is at most one outstanding operation O_i for each active transaction T_i. In the case of the restricted maximization problem in

of scheduling operations, and this benefit does not seem to be adversely affected by the intractability results.

- The important question of aborted transactions is not handled by the characterization. Indeed, the model in [Pap79] does not consider the commit and abort operations. As such, the question of the degree of concurrency with aborts possible is not addressed. Thus, any scheduler based on certification does support the maximum possible concurrency in a model since it potentially permits every possible schedule. In our model, we potentially take into account the impact of aborted transactions since the value functions may be defined for aborted transactions as well.

- In our context, the most important shortcoming of the no-delay schedulers is that they do not attempt to provide better performance explicitly despite the fact that it is supposed to be a primary goal. Thus, a no-delay scheduler simply depends on the input sequence to be one that will actually provide good performance, and it outputs the schedule unchanged. A closer examination reveals that the problem is not so much with the scheduler as it is with the schedules themselves; the problem lies with the assumption that all logically correct schedules are equally desirable. This is fallacious for the purposes of enhancing performance criteria, and examples provided above exhibit this shortcoming. In the case of time-constrained transaction scheduling, this problem becomes more acute since the correct schedules cannot be regarded as being equally important.

8.2 Enhancing Performance Explicitly

To output schedules that are better for some given measure of performance, it becomes necessary to schedule operations in terms of sets of operations rather than the approach used for no-delay schedulers. The reason is that a no-delay scheduler simply has two choices with regard to an input operation — either to output it, or to delay it and to check the next one. Since there is no performance benefit possible in the model by delaying an operation if its output does produce a legal prefix, the operation is output and this leads to the shortcomings in no-delay scheduling discussed above.

A consideration of the options that a scheduler has to enhance performance shows that it can affect the ordering of the operations in the output, and abort certain transactions. Given this limited power, the option available to enhance performance is to schedule a set of operations together, and possibly abort some of the active transactions. The objection to this approach is that a set of operations may not be available, and in that case, the approach reduces to the no-delay scheduler approach. However, in general, it is likely that a set of n outstanding operations are

However, if the scheduler did not guarantee that a schedules in its output class would be output unchanged if it were provided as an input, then this argument does not hold. It is not clear that such schedulers that may change legal input sequences admit of efficient implementations.

The no-delay characterization of schedulers has been long used to compare the utility of schedulers. Its success lies in its simplicity, and elegant mathematical characterization. Thus, it is used to make comparisons between the concurrency level supported by different schedulers (e.g., see [Pap79, KS88]). For example, a scheduler F_1 is said to allow more concurrency than a scheduler F_2 if, and only if, the class of schedules handled by F_2 is a proper subclass of the class handled by F_1. Furthermore, the no-delay stipulations permit the outright rejection of classes of schedules that cannot be efficiently recognized (e.g., see [KS88, AAS91]). Conversely, the existence of algorithms that efficiently recognize the legal prefixes of a class of schedules is considered as an indication that requisite schedulers may be constructed for that class. Note that the characterization includes all commonly used schedulers such as 2PL, TO, etc. In the case of certification, the only difference is that the check to ensure that a legal prefix has been generated is done at the end of a transaction rather than at each operation (e.g., see [BHG87]).

Unfortunately, there are several drawbacks to this approach to scheduling, even considered in a theoretical framework. Furthermore, for time-constrained transaction scheduling, the deficiencies are more acute.

- In practice, delays may be encountered for operations that arrive in an input sequence that is legal for a scheduler. For example, in a 2PL scheduler, an operation O_i that arrives while a conflicting operation O_j from another transaction is holding a lock, must await the release of the lock. In such cases, the scheduler delays the operation O_i, and instead, outputs some other operation O_k, thereby effecting a change in the input sequence. This is possible even though there may be a schedule in which O_j immediately precedes O_i. The problem is that the mathematical characterization does not capture such delays since the model used does not explicitly account for the passage of real time. The manner in which this is resolved in the characterization described above is that an output sequence that schedules O_i directly after O_j is created, and the underlying system is expected to effect the necessary delays between the two operations. It is a moot point whether the scheduler has indeed scheduled the operations with no delay involved.

- Intractability results exhibited for some liberal classes of schedules renders the classes unusable in practice for no-delay schedulers. This seems to indicate that those classes are entirely unusable in practice for any schedulers. However, as discussed above, that is not necessarily justified. One benefit of using more liberal classes is the greater choice they afford in terms

on the number of operations of transactions that are incomplete (i.e., transactions that are active). Thus, the scheduler behaves as an *on-line* scheduler since it does not need to examine the entire s_i sequence to produce s_o. Furthermore, the scheduler behaves in an *optimistic* manner in that a prefix of the output sequence is produced whenever it is possible to extend the prefix to a schedule that is legal within the correctness class that is handled by F. Thus, the function F is defined also for prefixes of schedules, and the output prefix of s_o must be legal. The latter requirement obviously subsumes the requirement that the s_o should be legal.

There is an implicit assumption that a scheduler must not delay any operation unless it is necessary to do so. The manner in which this is expressed mathematically is to require that any input sequence belonging to the logically correct class of schedules handled by the scheduler must be output unchanged. That is, scheduler F is said to handle a correctness class C if, and only if, for any input sequence $s_i \in C$, F produces s_i itself as the output (i.e., for any $s_i \in C$, it is the case that $s_o = s_i$). From the above discussion, this implies that the prefix of s_o that is output must be the same as the prefix of s_i that is input, for any prefix that is legal. Note that if there exists an algorithm that can efficiently recognize all legal prefixes for a correctness class C, then that algorithm can be used to construct a scheduler F that handles schedules in C. In doing so, the scheduler outputs operations from the input sequence one at a time by computing whether or not the output prefix generated thus far will be legal. We refer to such efficient, optimistic, on-line schedulers that do not delay any operations unnecessarily for a class C, as discussed in this paragraph, as *no-delay* schedulers for the class C.

It is not difficult to construct examples for situations where a sequence s_i may be within a class C that is efficiently recognizable, but whose prefix may belong to a class C' that is *not* efficiently recognizable (e.g., see [Pap79]). Hence, the efficient recognition of all legal prefixes of a class C is also necessary for the existence of a no-delay scheduler for that class.

Note that the above discussion indicates why the existence of no-delay schedulers for the class VSR is precluded since the class is not efficiently recognizable [Pap79]. However, it does *not* preclude the existence of schedulers that do not have all the characteristics of no-delay schedulers. In particular, if it is not necessary that an input sequence s_i that is legal with regard to a class C be output unchanged, then a scheduler may exist that outputs all schedules in C despite the possibility that legal prefixes of C are not efficiently recognizable. This may be understood by considering how the intractability of recognizing a legal schedule hinders the existence of no-delay schedulers. For, if it were the case that an efficient no-delay scheduler did exist for such a class, then the recognition problem for legal schedules could be tractably resolved as follows. The schedule in question, s_i, would be provided as input to the scheduler, and the output schedule, s_o, would be compared to the s_i. The schedule s_i would be recognized as being legal if, and only if, $s_i = s_o$.

It is outside the scope of this paper to examine the meaning and formulation of the various value functions. Indeed, we examine but a small subset of the large number of such possible formulations. Furthermore, in many cases it may happen that a value function is not available. In such situations, we exercise our discretion in suggesting that *default* value functions be used — keeping in mind that such functions are employed implicitly in any scheduling technique.

Several other issues that arise for our model may be easily incorporated within the framework. These include the questions of deadlocks, livelocks, and priority inversions (e.g., see [Son88] for a limited discussion relevant to TCTM). Suffice to say that their presence degrades performance, and their characteristics may be observed in our model in which they are reflected in schedules with low accrued values.

8 Revisiting Traditions

The traditional approach to transaction scheduling regards every logically correct schedule as being equally desirable. As large a number of schedules as possible are certified as being logically correct with the expectation that the concurrency control module will impact the performance as little as possible. In a nutshell, as stated in [Pap86], the goal of a scheduler is *"to preserve consistency while maintaining a high level of parallelism or performance."* Note that the preserving of consistency refers to the ensuring of logically correct schedules, and the parallelism refers to the number of such permitted schedules. In this section, we examine this goal in some detail in our context, and we use the description in the literature (e.g., see [Pap79, Pap86]) to study this established goal of transaction processing protocols.

8.1 No-delay Scheduling

We restrict our attention here to schedules that are sequences (i.e., total orders). Similar considerations for partial orders have been examined with essentially the same approach in the research literature (e.g., see [Kri82]).

A *scheduler*, or the concurrency control module, is regarded as a function F that transforms an input sequence of operations, s_i, into an output sequence of operations, s_o. The two sequences contain exactly the same operations, say n, in number. There are several requirements on a scheduler F as described below.

First, the output sequences that it produces must belong to an acceptable logically correct class of schedules (i.e., the output should be "legal") — the largest such class is taken to be VSR. Second, the function F must be efficiently computable in that its computational complexity must be P-time in n. Third, each operation in s_i should be output in P-time in the length of the prefix of s_i examined till that point. In practice, the efficiency requirement is changed to a P-time bound

those that are invoked in that period, those that complete, or both, etc.), must be specified. Since it is a *rate* measure, it is supposed to be defined for each point in real time; however, it is unclear as to which period is intended to be used to for counting the transactions. In UNIX, for example, the past few minutes are regarded as the period for the purposes of accounting the throughput. However, for scheduling, that does not serve the purpose since the obtained values are for the realized schedule rather than for the expected completion of an incomplete schedule.

Let us assume that the throughput is measured as the number of transactions that complete in some pre-specified real time interval (i.e., the transactions need not necessarily have been invoked during that period). Since we are not dealing with transactions that have any particular time constraint *per se*, it is not unreasonable to assume that the associated value functions are constant at a value, say 1. Assume that the penalty due to aborts may be disregarded. Hence, the throughput for the pre-specified period is simply the sum of the values accrued by the transactions during that period.

The problem with the above definition is that it is not entirely clear how the average throughput may be obtained. Thus, some *ad hoc* approach must be taken to define and maximize throughput. One approach that appears to be reasonable is to minimize the *makespan* of the schedule (i.e., the total length in real time for a schedule). We shall use this alternative approach to study throughput maximization.

7.4 Complicated Value Functions

The value functions examined so far are comparatively simple in that they have essentially only one parameter. However, it is possible that some performance criteria may be best represented by comparatively more complex value functions as the next example shows.

Consider the description of the 100 transactions per second benchmark for transaction processing systems. It is defined as the committing of 100 transactions each second, and additionally, with 90% having a response time of less than one second. To represent such a performance criterion, two value functions can be defined for each transaction — one to be used for the throughput, and the other to be used for the response time. In this case, the value function related to the throughput may be taken to be a constant function, and the one for the response time may be taken as a block-shaped function.

Thus, the value function for the schedule would need to be maximized along two dimensions, and the benchmark would be met for schedules that have a throughput value of 100τ, and a deadline value of 90τ, for each time period of τ seconds that may be examined. The precise meaning of an average value of 100 transactions per second is not very well defined.

value function. Such a value function is constant at, say, 1, up to the deadline point in real time, and after that, it reduces to a lower value. Note that there are several ways in which this reduction in value occurs, and consequently, a deadline can be interpreted in several ways. For example, *soft* deadlines permit the value to fall-off gradually — implying that the transaction in question accrues a non-negative value even if it does not commit by the deadline. In *firm* deadlines, the value falls abruptly to 0 — meaning that there are no gains nor penalties associated with the commitment of the concerned transaction after the deadline. Sometimes, a large negative value accrues after the deadline is passed, and consequently, it is preferable not to commit the concerned transaction after the deadline.

The above discussion suggests that a transaction should be aborted in some instances when the deadline passes or cannot be met. In such cases, it is necessary to consider the penalty (also provided by the value function) associated with the abort. In traditional real-time systems, this penalty is usually disregarded. Thus, in using extant results, for many cases, we disregard the penalty as well. Disregarding a penalty implies that the value function is taken to be 0.

Ensuring that our choice of schedules, guided by the maximization of the value accrued by them, does reflect, for example, meeting as many deadlines as possible, is not difficult. By disregarding the penalty of aborted transactions, the value function for a schedule is defined simply as the sum of the values accrued by the constituent transactions.

This example demonstrates how the sum of accrued values of transactions may be used to define the value functions for schedules.

7.2 Response Time

To achieve a minimization of the response times, consider a response time graph that plots the response time against the completion point in real time. This is a straight line that begins at the invocation point in time with a response time being 0, and has a slope of 45°. Thus, minimizing the average response time for a schedule is equivalent to finding a schedule such that the sum of the response times, divided by the number of transactions, is minimized. To state this in terms of the maximization of the sum of values accrued by transactions, the value function of each transaction should be regarded as the negative of the response time graph described above. Note that the value function formulation permits the expression of minimizing response times in a manner that favors the "more important" transactions as well.

7.3 Throughput

Throughput is a measure of the rate at which transactions are processed. Thus, the period of time over which the throughput is measured, and which transactions to include within that period (i.e.,

operating conditions (i.e., both "normal" and "crisis" situations). Thus, we do not address the question of whether or not a particular scheduling strategy produces schedules that commit the transactions "in time", or whether some transactions need to be aborted; such considerations are subsumed in the maximization described.

We do not deal with the various possible failures or their solutions specifically with regard to time-constrained situations in any detailed manner. Instead, it should be understood that the extant methods used in transaction management technology are to be used. In particular, logging on stable storage etc., are assumed to be required as in extant technology. However, it should be noted that in many instances, the full-fledged use of failure-resilience techniques may be unnecessary for time-constrained transactions (e.g., see [LJJL92]). For example, data that has significance for only a short period of time (e.g., the position of a target aircraft) need not be saved onto stable storage media. In general, this may be true only for a part of the data.

The above creates the problem that the access to the stable storage media makes for uncertainty in the timing estimates. With the extant technology, there is essentially little that can be done to alleviate this problem. In Section 11, we discuss the uncertainties in timing estimates.

7 The Performing Arts

We consider the performance metrics for time-constrained transactions in this section. It is assumed that the necessary triggering associated with the invocation of the transactions is achieved by other processes. Thus, compensating transactions, for example, are assumed to be triggered by other processes, and are treated simply like any other transaction. The emphasis in this paper is on centralized scheduling (i.e., there is exactly one concurrency control module).

It should be clear at this point that not all schedules that are logically correct are attractive in terms of performance. The performance that a schedule provides is defined by the value accrued by that schedule. We examine how several well-understood performance measures may be represented in the value function framework. This allows us to use value functions, in general, and to use extant techniques developed for the performance metrics we consider, in particular. The whole approach is to devise value functions that are, as far as possible, independent of the underlying system, the environment, and other concurrently executing processes. Note again that it is not within the scope of this research to provide studies on how the value functions are obtained.

7.1 Deadlines

Real-time applications are often associated with situations that have stringent timing constraints within which each process in the system must complete execution. These time constraints are in the form of *deadlines*. Deadlines, as used in *real-time* systems, may be represented by a block-shaped

Note that if the deadlines for each were changed to 4 instead of 5, the corresponding values accrued for the schedules a, b, c, d, and e, would be 2, 1, 1, 2, and $-\infty$, respectively.

Concurrency control modules are in a position to handle only a restricted set of value functions within their repertoire. We do not deal with the value functions directly, in general. Thus, the discussion may refer to value functions only implicitly by dealing with the notions of a deadline etc., instead of the corresponding value functions explicitly.

6 Time and Tide Wait for No Man

Given that some logically correct schedules are better than the others in terms of performance, the question arises as to how these may be selected. We address this question in some detail in Section 9. At this point, we consider the important issue of meeting the various temporal constraints in terms of the invocation times of the transactions, and the values accrued from the schedules. Also, we state our position with respect to what is meant when by the meeting of temporal constraints, at least in our context.

Note that except for entirely idealized situations, it is never possible to meet all the logical and temporal correctness criteria with complete certainty. Given that unforeseen failures and delays (which may be regarded as a kind of failure for time-constrained environments) may always occur, such guarantees are impossible. Instead, the recourse taken traditionally is to *assume* that certain types of failures do not occur, and to design systems based on these assumptions. The failures that are assumed not to occur are those that have a very low probability of actually occurring. Hence, systems that are based on failure-free assumptions do have a small, non-zero probability that they may fail.

In time-constrained environments, the question arises as to whether the probabilities that an estimated time for some execution will actually deviate from the actual time taken, is small. With the current technology, the answer is negative, and hence, it is futile to attempt to provide any certain guarantees with regard to the execution times (e.g., see [BMHD89]). However, that does not resolve the issues that are faced by a time-constrained scenario.

Further, note that meeting the time constraints on transactions is comparatively simple as regards the invocation times – which we state without further explanation — and research efforts are in progress to address these issues (e.g., see [BMHD89, C$^+$89]). We regard the problem of meeting the invocation times accurately as being resolved — except for crisis situations that are discussed in [KSS90]. However, the constraints on the execution times with regard to the value functions accrued, are quite difficult to meet. In our context, the meeting of time constraints is "equivalent" to finding those schedules for the invoked transactions that maximize the value accrued. Note that this provides a common framework, and common objective, for all types of

$vf(T)$ parameterized by the value of *time* (i.e., the clock) at the point in the schedule where the acknowledgment for the final commit or abort operation for T is received by the concurrency control module. This provides a means to determine the value accrued by each transaction in a schedule.

We are interested in the *value accrued* by a schedule. This may be represented as a value function for a schedule where the function is parameterized by the values accrued by the transactions in the schedule. Thus, the *value function* for a schedule s, denoted by $vf(s)$, is a mapping from the values accrued by its constituent transactions, among other parameters, to the set of real numbers.

Usually, the value function for a schedule is the sum of the values accrued by each transaction in the schedule. However, the value function for a schedule may depend on several other parameters as well. Note that the value function for a logically incorrect schedule is assumed to be infinitely negative. We provide an example to clarify the concept for the value of a schedule.

Consider two time-constrained transactions, T_1 and T_2, with associated value functions representing deadlines (both are assumed to commit), as follows.

- T_1: $R_1[y]R_1[x]W_1[x]W_1[y]$; $vf(T_1) = 2$ if clock≤ 5, else 0.
- T_2: $R_2[x]W_2[x]$; $vf(T_2) = 1$ if clock≤ 5, else 0.

Thus, T_1 and T_2 have the same deadline of 5 time units, and if they commit by that deadline, their corresponding values accrued are 2 and 1, respectively. Assume that the value function for the schedule is the sum of the values accrued by the constituent transactions, and that the weight functions (i.e., the time taken) for each operation is 1. The following schedules start at the clock value 0.

- **Schedule** a. $vf(a) = 3$; a is CSR.

$R_1[y]$	$R_1[x]$	$W_1[x]$	$W_1[y]$
			$R_2[x]$	$W_2[x]$

- **Schedule** b. $vf(b) = 3$; b is CSR.

	$R_1[y]$	$R_1[x]$	$W_1[x]$	$W_1[y]$
$R_2[x]$	$W_2[x]$			

- **Schedule** c. $vf(c) = 1$; c is a serial schedule.

	$R_1[y]$	$R_1[x]$	$W_1[x]$	$W_1[y]$
$R_2[x]$	$W_2[x]$			

- **Schedule** d. $vf(d) = 2$; d is another serial schedule.

$R_1[y]$	$R_1[x]$	$W_1[x]$	$W_1[y]$	
				$R_2[x]$	$W_2[x]$

- **Schedule** e. $vf(e) = -\infty$; e is logically incorrect.

$R_1[y]$	$R_1[x]$	$W_1[x]$	$W_1[y]$
$R_2[x]$			$W_2[x]$

the weight function for a particular operation O may differ in different schedules for the *same* set of transactions.

In a similar manner, the real time elapsed between the acknowledgment for an operation O_i, and the subsequent submission of an operation O_j, where O_i precedes O_j in a schedule, is measured on the clock, and is also represented by a weight function. This weight function, denoted by $w(O_i, O_j)$ for a pair of operations O_i and O_j such that $O_i < O_j$ in a partial order $<$ that represents a schedule, is a mapping from the set of such pairs of operations in a schedule to the set of positive real numbers. Again, this weight function may represent the actual or expected time elapsed. In general, the weights represent the actual elapsed real time for those portions of a schedule that are already realized, and estimated elapsed times for the portions that have not yet been executed.

Extrapolating from these definitions, it is possible to represent the real time elapsed between the submission (or acknowledgment) of an operation O_i up to the the same for another operation O_j in a schedule, where $O_i < O_j$. Without formally defining a function *time* that maps particular points in a schedule — where points are also to be understood intuitively — to the clock value (if the clock is accessed at that point in the schedule), we use it to associate the clock with real time.[1] Intuitively, *time* represents the real time at any particular point in the schedule (i.e., its value is exactly the same as the value for the clock at that point in the schedule). Clearly, the weight functions represent the difference in the values of *time* for the elapsed time. Of course, the value for *time* at a point that precedes another point in a schedule, is at most its value at the latter point. Finally, we assume the existence of a *big bang event* at which the clock value, and hence the value for *time*, happens to be 0. It is assumed that every operation in a schedule is preceded by the big bang event. Note that providing all the weight functions (including the ones from the big bang event to the submission of an operation) is equivalent to providing the *time* values for each point in a schedule.

Since a schedule is represented by a partial order, we need to proscribe certain temporally inconsistent schedules as follows. Since the value of the clock should be unique at any point in a schedule, we only deal with schedules where the value of *time* is consistent among any of the "paths" taken in the schedule from the big bang event, up to the point in question. Thus we only consider such *plausible* schedules, and we disregard all others. We refer to schedules with their weight functions or *time* values provided as being *temporally annotated* schedules.

5.2 All Schedules are not Created Equal

Now we are in a position to consider the values assimilated by the execution of transactions in a schedule. We say that the *value accrued* by a transaction T in a schedule is obtained by evaluating

[1] While it is possible to define these ideas formally, it has been avoided since they can be understood intuitively.

defer its execution under certain situations so long as the overall execution appears to have been one where the transaction to be aborted did not execute. We have exploited this technique further in [SLKS92], but we do not discuss the implementation issues that arise with respect to handling immediate aborts and compensations in this paper (see [Sop93]).

It should be noted that the role of abort operations is much greater in time-constrained transaction systems. Even if the underlying system can be guaranteed to be failure-free, and all the transactions guaranteed not to have errors, as long as it is not possible to guarantee execution times, there may be a need to abort for reasons explained above. The lack of a guarantee on the execution times is reflected in the uncontrolled passage of time represented by the clock. This requirement for the availability of aborting transactions implies that considerations such as *recoverable*, *cascadeless*, and *strict* executions, are an inherent part of time-constrained transaction management. We have examined some issues regarding failures in [SLKS92], and in Sections 6 and 9, we discuss some issues relevant to the abort operations.

The complementary issue that arises for a transaction execution when its precise execution time is unknown is that it may complete much earlier than expected. While this may not appear to be problematic in general, it is quite possible that the early completion creates a negative value in the associated value function. To handle this problem, the concurrency control module can simply insert an artificial delay in reporting the effects of the transaction, and disallowing other operations from examining the effects of the transaction in question. In other words, early completion is indeed an easily handled issue.

5 O Tempora! O Mores!

In this section, we consider several important issues concerning time-constrained transaction management. We first describe how the time elapsed during an execution, as measured by the clock, is represented in our model. Subsequently, we differentiate the schedules in terms of the values that they assimilate due to the transactions that occur in them.

5.1 Time Annotations for Executions

We first consider the real time elapsed between the submission of an operation to the underlying system up to the point where it is acknowledged, as measured on the clock by the concurrency control module. This finite passage in real time for the execution of an operation is denoted by a *weight function* associated with an operation O in a schedule s, is a mapping from the set of operations in s to the set of positive real numbers.

Note that the weight function may either be the actual elapsed time for an operation within the realized schedule, or it may be the estimated execution time for the operation. Furthermore,

could be regarded as being equivalent to a logically incorrect execution of a transaction — i.e., one that cannot be permitted under any circumstances. Finally, a constant value function suggests that the completion time for the corresponding transaction has no significance in terms of its completion time.

It was mentioned that a value function $vf(T_i)$ for a transaction T_i may have parameters other than the clock value. Thus, the operations of the transaction T_i may themselves be regarded as a set of parameters provided to $vf(T_i)$. In particular, $vf(T_i)$ evaluates differently for the case where a commit is part of T_i as compared to the case where an abort is part of T_i. Thus, we represent the value assimilated by the committed execution of a transaction in the same way that we represent a penalty for the aborted execution of the transaction.

4.2 Transaction Commitment

The completion time of a transaction is the value of the clock at the point in real time when the last operation for the transaction has been acknowledged to have been executed by the underlying system. In circumstances where the completion is by commitment, and it is delayed (since the clock has advanced more than was expected), the value function may evaluate to an inordinately large negative value. In that case, the effects of a transaction may need to be undone as though the execution of that transaction did not occur. Prior to the commitment operation having been submitted by the concurrency control module, this is generally accomplished by the abort operation (e.g., see [BHG87]). However, an abort *after* a commit has been executed creates the anomalous situation wherein a committed transaction (i.e., the acknowledgment for the commit operation may have been received by the concurrency control module at that point) needs to be aborted.

The above situation requires an *immediate* abort to be effected as follows. Prior to the submission of any further operations of the schedule, it is determined whether or not the transaction needs to be aborted. If the transaction need not be undone, things proceed as usual. Otherwise, a *compensating* transaction is invoked which essentially simply undoes the operations just as a simple abort would do. The difference is that the compensating transaction needs to access the committed values for the data entities that it must change. This also implies that all the accesses to the database must be effected through the concurrency control module alone since some of the database entities with *committed* values may actually be undone subsequently. In terms of representation, the commit of a transaction, followed by an immediate compensation, is simply represented as an abort operation (e.g., see [KLS90]). This is acceptable since the abort operation itself may be regarded as a small process which accesses the shadow pages etc. to undo the effects of the transaction. The difference here is that the commit operation, followed by the compensation, *together* constitute the abort operation. Note that the use of a compensating transaction means that we can actually

are investigated in other research (e.g., see [C+89]). However, for the important consideration of crises resolution, we examined some relevant issues in [KSS90].

4 Value Systems and Last Rites

In this section, we consider how transaction executions are concluded, and we examine specific issues that arise in that regard. First, note that while the clock is regarded as a special data entity, we do not permit the output predicate of a transaction to mention it. This is to prevent the possibility of an isolated execution of a transaction resulting in a state that does not satisfy the output predicate. Such a possibility exists as a consequence of the uncontrollable passage of time. However, it is necessary in many situations to ensure that a transaction is allowed to complete at a point in real time that satisfies some relationship to the clock value. For example, a transaction that activates an external physical device that should begin some task at a particular point in real time, has such constraints.

As a second consideration, certain transactions need to be regarded as being more important or crucial as compared to the others. For example, the transactions resolving a contingency criterion may be regarded as being more important than the others. In order to schedule such transactions with consideration given to their importance, there is a need to represent such relationships among the transactions explicitly.

4.1 Value Functions

To address both issues that were raised above, we describe the notion of a value function for a transaction in a manner similar to that examined in the research literature (e.g., see [JLT85, AGM88]). Such a function is associated with the real time at which a transaction completes its execution. While it may have several other parameters, one that concerns us is the temporal parameter, and the value obtained from the execution of a transaction is the value of the function parameterized by the value of the clock at the time that the execution is completed.

The *value function* for a transaction T_i, denoted by $vf(T_i)$, is a mapping from real time (i.e., the clock), among other possible parameters, to the set of real numbers. The values assimilated by the execution of different transactions may differ, and that permits the delineation of transactions in the order of their importance in a quantitative manner. Value functions are derived from the applications at hand, and several typical functions have been suggested (e.g., see [JLT85]).

It should be clear that several well-known temporal constraints on the completion times of a process may be represented as suitable value functions. For example, a deadline constraint may be represented as a suitably designed value function in the form of a constant function that abruptly drops (possibly to a negative value) at the deadline. Moreover, an infinitely negative value function

Suppose that the temperature in a nuclear reactor is measured and written to a data entity d in the database. A contingency constraint may be stated as $d \leq 1000$ to represent a constraint that the temperature should never exceed 1000 degrees. Furthermore, a deadline value of 100 time units may be associated with the constraint to represent that the "dangerous" temperature must be dealt with in a period not exceeding 100 time units from the time that the crisis was detected. Hence, a transaction invoked as a result of a violation of the constraint must complete execution within 100 time units.

Note that contingency constraints are trigger constraints which do not mention the clock, and furthermore, have an additional associated real time *deadline* constraint. This permits their use in situations where certain transactions need to be triggered and executed within a particular interval of time. Consider the example of airspace traffic control adapted from [LJJL92] in which an airplane moves from an airspace A to an adjacent airspace B. Since different air traffic controllers are responsible for each airspace, it is required that the aircraft be outside the control of both the controllers for no more than 500 milliseconds. Moreover, at most a single controller at a time is permitted to have control over the aircraft. The data entity O_a indicates whether the control of the aircraft is with airspace A or not depending on whether the value of O_a is 1 or 0 — and similarly for O_b and airspace B. Clearly, a contingency constraint such as $O_a + O_b = 1$ may be defined with the associated temporal deadline of 500 milliseconds.

The example suggests three transactions as follows. Transaction T_1 updates the values of O_a and O_b to indicate the changeover of control between the airspaces. Transactions T_2 and T_3 simply read the data entity O_a and update the displays for their respective controllers in case the aircraft happens to be within the jurisdiction of that airspace.

T_1: $W_1(O_a)\ W_1(O_b)$
T_2: $R_2(O_a)\ W_2(display_a)$
T_3: $R_3(O_b)\ W_3(display_b)$

Note that while any interleaved execution of the above transactions is serializable, unless the temporal constraint is also obeyed, the safety requirement of uncontrolled flight for less than 500 milliseconds would be violated.

Assuming that the contingency constraint violation imposes the 500 milliseconds deadline on transaction T_1, and that the deadline is obeyed, it happens to be the case that any serializable execution where the two operations of T_1 are separated in real time by 500 milliseconds, is safe. Note that in this example, we happen to have a situation where the transaction that reinstates the constraint is the same as the one that violates it — something that need not be true in general.

In general, we deal with transactions after they have been invoked. That is, we are not concerned with the actual mechanisms or issues involving the triggering of the transactions; triggering issues

so that the uncontrolled passage of time does not cause the database to become inconsistent. Thus, in time-constrained environments, we have an additional set of constraints that are similar to the database integrity constraints — however, these constraints are useful for triggering certain transactions, and they are liable to become invalid from time-to-time as explained below. Also, in most instances, these constraints are stated explicitly at least at the design stage for an application.

Assuming that a trigger constraint is stated in a form such that its violation results in the invocation of a transaction, the resulting transaction execution is expected to restore the database state to one that satisfies the trigger constraint. Any transaction that may affect a trigger constraint may be regarded as checking the validity of the constraint as well. However, since the clock is not affected by any transaction, triggering mechanisms for constraints that mention the clock have to be different. Note that such constraints are likely to result in situations where the associated transactions are periodically triggered.

Consider the relationship $timestamp(d) + c \leq clock$ to be a trigger constraint. It may be regarded as representing the requirement that a transaction that updates the value of the entity d every c units of real time, be invoked periodically with a period of c. Thus, in a situation where the temperature of a furnace is being monitored every 5 seconds, c may be regarded as 5 seconds, d may be regarded as the entity that represents the temperature, and the invoked transaction would be the one that makes a recording of the transducer reading into the database.

Aside from noting that it is possible to state trigger constraints, we shall not direct any further attention to them except as noted in the next subsection.

3.4 Contingency Constraints

Many time-constrained applications that need the use of transactions are often faced with crisis situations that need computations that will resolve the crisis quickly (e.g., see [KSS90]). These crises must be anticipated in advance, and the transactions designed to deal with them should be triggered automatically. However, the time at which they may occur is not known *a priori*. We refer to these special trigger constraints as contingency constraints. The clock is not mentioned in these constraints, and instead, we associate a temporal *deadline* within which the constraint needs to be satisfied in case it does get violated. The reason for disallowing the clock, in general, is that unlike typical trigger constraints, contingencies occur with no known periodicity, and no known relationship to real time. A transaction that is invoked due to a contingency constraint violation is required to complete execution, and thereby restore the validity of the constraint, within the time specified by the deadline. We examined such situations in greater detail in [KSS90], and for the present, we simply provide the following example.

currency of data may be accessed either in the input predicate, or at some points in the transaction programs. A typical example for a temporal currency relationship is as follows.

$$(timestamp(d_1) + c_1 \geq clock) \wedge (timestamp(d_2) + c_2 \geq clock)$$

Such a relationship may be used to ensure that the accessed values of the entities d_1 and d_2 (which may represent the values of stocks, or the temperature in a furnace, etc. in requisite applications), are not very stale (since c_1 and c_2 are regarded to be suitable constants). In fact, it should be noted that if such constraints are satisfied, and they occur within an input predicate, then the associated transactions happen to have been invoked as required in real time.

3.2 Temporal Coherence

Besides the currency of the accessed data, a transaction may require that the "spread" in the *timestamp*() values among the accessed data entities obeys some relationship. This relationship is often one that ensures that the values accessed represent those that have been generated close together in real time. This is especially important in situations where multiple alternatives are accessed. Such a temporal coherence relationship among the data entities is also expressed simply as a relationship among the *timestamp*() values. An example for a temporal coherence relationship is as follows.

$$|timestamp(d_1) - timestamp(d_2)| \leq c$$

Such a relationship may be used to ensure that the accessed values of the entities d_1 and d_2 (which may represent the values of two different stocks, or the temperature and the pressure in a furnace, etc. in requisite applications), are within an application-dependent spread of values.

Notice that, in general, the logical constraints, and the two types of temporal constraints described above, may not be easily delineated. In fact, they need not be since we regard the clock as simply another data entity.

3.3 Trigger Constraints

The transactions in a time-constrained environment are often constrained to be invoked at particular points in real time depending on the state of the database. That is, particular states of the database, including the state of the clock, may *trigger* the invocation of a transaction. The inclusion of the value of the clock in the database state suggests that relationships on the database states may be defined whose violation or satisfaction (depending upon how they are defined) should trigger some transaction. We refer such relationships as trigger constraints. It should be noted that we do not regard these to be part of the database integrity constraints (e.g., unlike the description in [Ram92])

are maintained. Similarly, if T_3 were made to access the value for γ_i, the equivalent serial schedule $< T_2T_3T_1 >$ would result. However, a shortcoming of the latter case would be that T_3 would use old values (which we are not considering at present).

It is not our intention in this paper to study the expressibility of various application-specific requirements, and nor do we detail the requirements of applications aside from the available research in the literature listed in the references. Instead, we suggest that a general model, such as the NT-PV model, be used, and our research is useful for in cases where the transactions for the applications are suitably captured the model.

3 A Brief History of Time

The notion of time plays an important role in our research. For a centralized environment, we assume the existence of a device available that measures the passage of *real* time. Furthermore, we assume that the processes that execute in the system may access the value of time by referring to a special data entity called the *clock*. Thus, in a centralized system, the clock is a read-only, positive, real-valued counter that increases monotonically and atomically, and is accessible to all processes. Note that since the clock is read-only, it is implied that some process, transparent to the other processes in the system, increments it. We capture the uncontrollable passage of real time in this manner.

Since it is a data entity, the clock may be referred to in the input predicate for any transaction. Similarly, it is permissible to access the clock by reading its value within a transaction as well. The heretofore uninterpreted attribute, $timestamp(d)$, for an entity d, is to be regarded as the *real timestamp* for the entity, and it is regarded as representing the point in real time when the value of d is valid. That is, $timestamp(d)$ is the value of the clock at which the data entity d has its "actual" value as regards the environment that the database represents. Note that this is simply a means to associate the value of the clock with the validity of a data entity, and that validity is an application-specific criterion. Finally, the value of the clock may be used to design the means to *trigger* the execution of some transactions that need to be invoked at a particular point in real time, and under a pre-specified state of the database. Below, we describe several of these important features.

3.1 Temporal Currency

The data values accessed by a transaction should be "recent" (e.g., see [KM92, SL90, LJJL92]). To allow us to state this more rigorously, we consider a relationship between the $timestamp(d)$ attributes of an entity accessed by a transaction, and the value of the clock. We term such a relationship as one that represents the temporal currency of the accessed data. In general, temporal

the *consistency constraints* of the database (e.g., see [Pap86]). Such a condition is consistent with the requirement that a transaction executes correctly in isolation. Furthermore, it should be clear that a transaction may be regarded as a function that maps consistent states of the database to other consistent states (e.g., see [Pap86, KS88, KLS90]). We use this notion of a transaction in [SLKS92].

Note that the input predicates are usually derived from the application at hand. Since we use the criterion of serializability for logical correctness, the particulars of the output predicate are not crucial since the execution of a transaction in isolation, by definition, will ensure it. A condition that the predicates are required to meet is that they admit tractable evaluation. That is, there should be an evaluation algorithm that is a polynomial time, in the number of entities mentioned, to test their satisfiability with the values present in the database.

To illustrate the use of the input predicates, along with alternatives, consider an example adapted from [LJJL92] for an inertial navigation system in an aircraft. Suppose that the system provides the aircraft latitude and longitude, λ and γ), respectively, every 59 milliseconds, and these are written into the database by a transaction T_1. Also, a radar produces the range and bearing of a target, ρ and θ, respectively, every 6 seconds, and these are written to the database by a transaction T_2. Finally, a sporadically invoked transaction, T_3, uses these values to compute the position of the target, *pos*. Thus, we have the following transactions and their operations.

T_1: $W_1[\lambda]W_1[\gamma]$
T_2: $W_2[\rho]W_2[\theta]$
T_3: $R_3[\lambda]R_3[\gamma]R_3[\rho]R_3[\theta]W_3[pos]$

Assume that the recorded values are timestamped, and this timestamp attribute, *timestamp*(), is represented by a subscript on the entities when the attribute is significant. Also, let that the superscripts on the operations represent the time that the corresponding periodic transaction is invoked. The application is such that a current measurement of the aircraft position is equally good for the calculation of the target position as a measurement that is 59 milliseconds old. Finally, assume that the last values written for the target position are good for the calculation. Clearly, the following schedule would be considered logically incorrect since it is not a serializable schedule.

$$W_1^i[\lambda_i]W_1^i[\gamma_i]R_3[\lambda_i]W_1^{i+59}[\gamma_{i+59}]W_1^{i+59}[\gamma_{i+59}]R_3[\gamma_{i+59}]R_3[\rho]R_3[\theta]W_3[pos]$$

However, this situation is alleviated if alternatives are maintained, and an input predicate is defined for transaction T_3 that states a relationship between the accessed entities and alternatives such as $|timestamp(\lambda) - timestamp(\gamma)| \leq 60$. Hence, the above schedule may be serializable by suitably assigning the alternatives that are read. In particular, the above schedule is equivalent to the serial schedule $< T_2T_1T_3 >$ where different alternatives for separate values of *timestamp*()

system. The same module is also responsible for temporal considerations which will be introduced below. Also, if the invocation of an operation by the concurrency control module occurs after the reception of an acknowledgment for a previously submitted operation from the underlying system, then the newly invoked operation is assumed to be executed after the acknowledged operation.

In many cases, all the operations for a transaction T_i may not be known to the concurrency control module during the execution of T_i, and thus, the associated partial order for T_i that represents the execution thus far is incomplete in that it does not include all the operations of T_i. Let us refer to such partially executed transactions as being *incomplete*. We find it useful to define the *expected completion* of an incomplete transaction T_i by including one of the abort or the commit operations, and defining an *expected* partial order that agrees with the associated partial order for the incomplete transaction for all the other operations. An expected completion is said to be an *aborted* or a *committed* expected completion according as the abort or the commit operation is employed in constructing the completion. Note that the expected completion of an incomplete transaction effectively creates a (complete) transaction.

Since a schedule consists of transaction operations, all the operations in a schedule may not be known to the concurrency control module at execution time. Thus, we may define an *incomplete* schedule as the prefix of a schedule (i.e., thereby containing an incomplete transaction). Furthermore, an *expected completion* of an incomplete schedule consists of the the incomplete schedule augmented by expected completions for each incomplete transaction represented in it. The prefix of a schedule that represents operations that are already executed (i.e., their execution is acknowledged by the underlying system at that point in real time) is called a *realized* schedule. Note that a realized schedule is, in general, an incomplete schedule.

The NT-PV model [KS88] is adopted to permit the examination of serializable as well as more liberal correctness criteria within a common framework. The model also helps to explicitly express certain relationships among the entities that are accessed by the transactions. Such features, from the NT-PV model, are adopted for some special requirements in a time-constrained environment to facilitate either more restrictive, or more liberal, classes of correct schedules. Specifically, *input* and *output* predicates are defined for a transaction that refer to data items within the database (e.g., see [KS88]). The input predicate defines the conditions under which its associated transaction may proceed with its execution. If it is not provided, the input predicate is regarded as evaluating to *true*. Note that the input predicate may be regarded as being part of the transaction program itself; whether its evaluation is effected by the program or by the modules that control the transaction executions, is immaterial. The output predicate describes the state in which an isolated execution of the associated transaction is guaranteed to leave the database if the input predicate of the transaction were also satisfied. The input and output predicates of each transaction must satisfy

is left unspecified except to state that a transaction preserves the consistency of a database when executed in isolation — i.e., in the absence of any other concurrently executing transactions (e.g., see [Pap86]). Such an execution is regarded as a logically correct execution of a transaction. The more complex models of transactions (e.g., see [KS88]) may be studied for TCTM in a similar manner, but they are outside the scope of this paper.

A *schedule* of transactions, s, is a partial order, $<$, on the set of operations from the transactions, and s subsumes each partial order $<_i$ for a transaction T_i that is represented in it (e.g., see [Pap86, BHG87, KS88]). Very often, when a schedule happens to be a total order, it is represented as a sequence of operations.

We regard the criterion of *view serializability* (VSR) as the logical correctness criterion for a schedule for concurrently executing transactions (e.g., see [BHG87, Pap86]). For this correctness criterion, the effect of transaction aborts on a schedule is to restrict the schedule to include only the unaborted transaction operations (e.g., see [BHG87]). Note that the view serializability criterion is regarded to be correct primarily for pragmatic reasons; more liberal criteria are certainly possible, but they are generally more difficult to state, implement and analyze. An examination of the benefits, if any, of regarding a larger set of schedules as being correct, is deferred to Section 8. Note that even in situations where non-serializable executions may appear to be acceptable, they may not be. Consider a Doppler shift based navigational system in an aircraft. Two entities, *velocity* and *control*, are recorded in the database by a transaction T_1. The latter variable indicates whether or not the value of the former is accurate (since, under certain situations such as a change in the direction of the aircraft movement, the recorded value may not be accurate). Suppose that transaction T_2 reads the values of the two variables, and displays the value of *velocity* if it is accurate. Assume that the transactions T_1 and T_2 are invoked periodically to update a display. This example shows that a serializable schedule of the two transactions, with T_1 preceding T_2 in the equivalent serial schedule, captures the required behavior. On the other hand, a non-serializable execution may lead to the incorrect display of an inaccurate value for the *velocity*.

Exceptions to view serializability may be made in certain carefully considered situations where serializable schedules are too restrictive, and its requirements are then relaxed (e.g., see [Son88, KSS90]). However, we focus attention for most part to *conflict serializability* (CSR) as the logical correctness criterion (e.g., see [BHG87, Pap86]), and such schedules are a proper subset of VSR schedules. The reasons include the ease of ensuring CSR schedules, the properties of CSR schedules that make them amenable for use in distributed environments (e.g., see [SSK92]), and the research results accrued are applicable to more general environments as well.

We assume a model for the software architecture as presented in [BHG87] in that a concurrency control module responsible for ensuring logically correct schedules exists above the underlying

explicitly, the last written alternative will be regarded as being the one to which a reference is being made. Note that in the standard models (e.g., see [BHG87, Pap86]), only one alternative of an entity (i.e., the *standard* alternative — see [KS88, Pap86]) is accessible.

Each entity is permitted to have a set of application-dependent attributes. For example, an attribute $timestamp(d)$ for an entity d may be regarded as a temporal attribute for the entity d. Note that the semantics associated with $timestamp(d)$ are entirely application-dependent in that it may be regarded as the time at which the entity was last updated, or the time at which the process that created it was initiated, etc. (e.g., see [SL90, LJJL92]). Allowing attributes for the entities, along with the definition of alternatives, facilitates the expression of ideas that are relevant to TCTM (e.g., it is possible to express a *trajectory* of the values assigned over time to an entity).

The access of database entities is effected by an *operation* defined to be an *atomic* access of the database. An operation may be regarded as a function from one database state to another. We are primarily concerned with the operations *read*, *write*, *delete*, and *create* of the database entities, and their semantics are assumed to be well-understood (e.g., see [BHG87, Pap86]). We often abbreviate read and write to $R[\ldots]$ and $W[\ldots]$, respectively, where the entities accessed are mentioned in the square brackets. Since a *method* used on an object in object-oriented systems is also an atomic execution, the term "method" is subsumed in the definition of an operation. Also, the special operations for *commit* and *abort* are of interest in our discussions, and their semantics are assumed to be known to the reader (e.g., see [KLS90, BHG87, Pap86]). Thus, while the meaning of an abort operation may vary with the particular application or model at hand (e.g., see [KLS90, LJJL92]), its essential characteristic is to undo the effects of a partially ordered set of operations that is expected to execute atomically. Similarly, the commit operation is used to confirm the atomic execution of a partially ordered set of operations.

A *transaction*, T_i, is a partial order, $<_i$, on a finite set of operations that is guaranteed to execute in a logically correct manner, as described below, when executed in isolation (e.g., see [Pap86, BHG87, KS88]). The partial order is generated by the *program* from which these operations emanate, and we are not concerned with these transaction programs *per se* — unless otherwise indicated. Exactly one of the abort or the commit operation is part of a transaction T_i, and all the other operations in T_i precede it in $<_i$. Note that we do not include the abort or the commit operation for a transaction in some discussions in the paper because we are often interested in the transaction executions without regard to their final outcome. The atomic execution of a transaction must be provided by the concurrency control with which we are primarily concerned in the following discussions (e.g., see [BHG87]).

A transaction is also the unit for the consistent access of the database, and a units of recovery in case of failure (e.g., see [BHG87, KS88, Pap86]). The precise meaning of a *consistent* access

these techniques. The research in [SRL88] proposes concurrency control techniques for distributed real-time systems based on a partitioning of data, while the studies in [PR88] discuss the specific time-dependent application of stock-market trading.

Thus, most previous work on TCTM assumes a set of transactions and associated temporal deadlines, and hence, *ad hoc* heuristics are studied from a performance evaluation standpoint. These studies are useful to the extent that with realistic workloads, they do provide a measure of the utility of the heuristics adopted. However, they do not provide a basis for TCTM.

Our model for transaction-processing is based on extensions to established models, and we briefly outline how logical and temporal constraints may be expressed in it. For scheduling the transactions, we study how legal schedules differ from one another in terms of meeting the temporal constraints. Existing scheduling mechanisms do not differentiate among legal schedules, and are thereby deficient with regard to meeting temporal constraints. This provides the basis for seeking scheduling strategies that attempt to meet the temporal constraints while continuing to produce legal schedules.

For the centralized scheduling of transactions, we examine the feasibility of achieving optimal results in terms of meeting the logical and temporal constraints. In most cases, this is shown to be intractable using results from extant studies in scheduling. Hence, we describe the means to achieve good results based on existing heuristics. Also, since the execution time estimates that are used in such studies are prone to be undependable, the issue of dealing with timing uncertainties in the context of scheduling is addressed.

2 A Natural Selection

In this section, we describe a framework which serves as a basis for presenting our results for time-constrained transaction scheduling in a centralized environment. Note that we have chosen to adapt well-established models for our needs.

The *database* is a set of data *entities*. The database is *persistent* in that the data may have lifetimes greater than the processes that access them. Entities within the database may be created, read, modified, and deleted, with suitable access mechanisms described below. An entity may be very simple, or quite complex; its characteristic feature is that it is always accessed atomically (i.e., as a whole).

Each entity is identified by a *name*, and has associated with it a *value* from some domain. The *state* of a database is a mapping of the entity names to their corresponding values. While each entity is permitted to have more than one *version*, we simply regard the different versions of an entity as separate entities in their own right, and hence, we refer to a version as an *alternative*. Thus, the alternatives are expected to be managed by the applications themselves. When not specified

1 Introduction

Transaction processing is an established technique for the concurrent and fault-tolerant access of persistent data. While this technique has been successful in standard database systems, factors such as time-critical applications, emerging technologies, and a re-examination of existing systems suggest that the performance, functionality and applicability of transactions may be substantially enhanced if temporal considerations are taken into account. That is, transactions should not only execute in a "legal" (i.e., logically correct) manner, but they should meet certain constraints with regard to their invocation and completion times. These logical and temporal constraints, typically, are application-dependent, and our research addresses some fundamental issues for the management of transactions in the presence of such constraints. We refer to this area of study as *time-constrained transaction management* (TCTM).

Transaction processing in database systems does not explicitly address the question of meeting any time constraints. However, performance enhancement techniques, such as query optimization and concurrency control, are implicitly geared to reducing the time taken by transactions that access the database. In contrast, the area of real-time systems, which is closely related to TCTM, has the handling of temporal considerations as a primary goal. Such systems find finds applications in process-control, and often require a large database of information. Hence, recent efforts have aimed at integrating real-time systems with transaction processing to facilitate the efficient and correct management of the resulting TCTM systems (e.g., see [Son88]). Also, there are several examples where explicit temporal considerations are necessary on the transactions that access a database system (e.g., see [Son88, KSS90]). In this paper, we describe several situations from TCTM environments that serve as specific examples.

There are difficulties in accomplishing the integration of transaction processing and time-constrained scheduling techniques. First, it is not clear how the scheduling of transactions, whose aim is to produce correct schedules, may be integrated with time-constrained scheduling, which aims to meet temporal constraints. Second, the execution of a transaction operation (read or write) takes a highly variable amount of time depending on whether disk I/O, logging, etc. are required — which is not the case for tasks in a time-constrained scheduling situation. Also, if concurrent transactions are allowed, the concurrency control, to ensure correctness, may cause aborts or delays of indeterminate length.

There has been extensive study of real-time systems [Sta88]. Formal aspects of such systems have been examined from the standpoints of scheduling (e.g., [HMR+89]) and verification [JM86]. In the context of real-time databases, [AGM88, AGM89] consider alternative queuing disciplines with lock-based concurrency control of real-time transactions, and use simulation results to compare

Time-Constrained Transaction Scheduling

Nandit Soparkar[1*]
Henry F. Korth[2]
Avi Silberschatz[1†]

[1]Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188 USA

[2]Matsushita Information Technology Laboratory
182 Nassau Street, third floor
Princeton, NJ 08542-7072

Abstract

Time-constrained transaction management incorporates *temporal* considerations into the transaction and scheduling model. In such a model, transactions are expected to meet certain time constraints with respect to their invocation and execution times. Most existing research deals with the performance evaluation aspect of deadline-based transaction executions using *ad hoc* heuristics. This paper presents a model on which to base the scheduling and correctness criteria for time-constrained transaction management in a centralized computing environment. Our model permits a rigorous study of the scheduling problems that arise when temporal factors are explicitly taken into account. We present some of our research results for scheduling, and discuss several relevant issues regarding the domain of time-constrained transaction management.

*Work supported by an IBM Graduate Fellowship.

†Work partially supported by NSF grants IRI-9003341 and IRI-9106450, by the Texas Advanced Technology Program under Grant No. ATP-024, and by grants from the IBM corporation and the H-P corporation.

TIME-CONSTRAINED TRANSACTION SCHEDULING

Nandit Soparkar
Henry F. Korth
Avi Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-92-46 December 1992

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUSTIN, TEXAS 78712