Int J Parallel Prog (2008) 36:386–411
DOI 10.1007/s10766-007-0061-2

On the Design and Implementation of a Shared Memory Dispatcher for Partially Clairvoyant Schedulers

K. Subramani · Kiran Yellajyosula

Received: 25 April 2006 / Accepted: 7 December 2007 / Published online: 29 December 2007

Abstract With the onset of distributed computing in hard real-time applications, the problem of assigning to, scheduling in, and executing jobs on processors, has received a lot of attention. Usually, real-time systems are embedded in closed loop reactive environments with uncertain behaviors and such systems take varying times to respond to such stimuli. One of the fundamental features of such systems is the presence of complex timing constraints between pairs of jobs. A secondary feature is the non-constant nature of the execution times of jobs. Real-time operating systems such as MARUTI can measure the interval within which the execution time varies (Mosse et al. In: Second IEEE Workshop on Experimental Distributed System, pp. 29–34., IEEE, 1990; Levi et al. 1989, ACM Special Interest Group Operat Syst 23(3):90–106). Partially clairvoyant scheduling was introduced in (Saksena, Parametric Scheduling in Hard Real-Time Systems. PhD thesis, University of Maryland, College Park, June 1994) to schedule jobs with varying execution times and non-trivial timing constraints. The schedulability of the job set is determined offline and a set of dispatch functions are produced from the given set of constraints if the job set is schedulable. The dispatch functions bind the start time of a job J to an interval that depends on the start and execution times of jobs sequenced before J. The online dispatcher of the system reads these dispatch functions and computes the interval within which a job can start without violating the constraints imposed on the system. In certain situations, the dispatcher fails to dispatch a job as the time to compute the dispatch functions associated with a job is greater than the interval within which the job needs to be dispatched. This

K. Subramani (✉)
LCSEE, West Virginia University, Morgantown, WV, USA
e-mail: ksmani@csee.wvu.edu

K. Yellajyosula
CSE, University of Minnesota, Minneapolis, MN, USA
e-mail: kiran@cs.umn.edu

phenomenon is called *Loss of Dispatchability* (Subramani, Duality in the Parametric Polytope and its Applications to a Scheduling Problem. PhD thesis, University of Maryland, College Park, August 2000). In this paper, we propose and implement a partially clairvoyant dispatching algorithm on a shared memory cluster with Concurrent Read Exclusive Write (CREW) architecture and contrast it with the sequential approach. For a preset number of processors, our approach has $O(1)$ dispatch complexity while using a total of $O(n^2)$ space, while the sequential approach requires $\Omega(n)$ time. The detailed implementation profile obtained clearly demonstrates the superiority of the multiprocessor approach to dispatching. We also address the issue of scalability of the dispatcher for increasing number of processors and show that job sets of different sizes require different number of processors. Finally, we demonstrate the effect of execution time on the dispatchability of schedules.

Keywords Partially clairvoyant dispatcher · Shared-memory · Real-time scheduling · Loss of dispatchability · Safety interval

1 Introduction

Real-time systems are gaining importance in a number of applications ranging from safety-critical systems such as nuclear reactors and automotive controllers, to entertainment systems such as games and animations. In hard real-time systems, the results need to be computed within their deadlines. Additionally, there exist complex timing constraints between pairs of jobs. Traditional strategies schedule jobs using the worst case execution times based on some a priori knowledge of the arrival times or the frequencies of the jobs [12,15,20]. However, assuming that a job will always take the worst case time to complete could lead to catastrophic constraint violations at run time.

Consider the following applications where the systems need to respond within a small interval of time:

(i) Sound and video synchronization are essential for a person playing video games. Video games such as PlayStation, Dreamcast, GameCube, and N64; have multiple embedded processors to record, display and react to the changes made by the player in a very interval of time. These controllers perform multiple tasks such as controlling different components, computing the response to the player and so on. Computing a response to a move requires the controller to evaluate various complex scenarios which are being modeled by the game.

(ii) Mission-oriented robots are equipped with multiple sensors and actuators to achieve a goal in hostile environments; typical goals include surveying the landscape and searching for survivors [17]. The Mars Pathfinder was designed to explore the surface of Mars and transmit information about the surface and topology of the planet. Such robots need to perform multiple tasks concurrently such as monitoring their components and the environment, collecting information, transmitting information and moving from one place to another [4].

(iii) Automobiles such as BMW 7-series and Mercedes S-class contain over sixty microprocessors. All these embedded microprocessors communicate and work

with each other for improving the comfort and safety of passengers. Jaguars and Volvos, use the PowerPC 505 to control the engine and calculate time-angle ratios, which is vital for valve and ignition timing. Some car radios have noise compensation algorithms written into them which can reduce the noise or increase the audio volume according to the road noise.
New cars have adaptive shifting algorithms to modify the shift points based on the road conditions, weather, or the driver's individual habits. The cruise control system varies the acceleration according to the exact speed of the car provided by the anti-braking system (ABS).

(iv) The internet has grown to be the center of various business applications for marketing and auctioning products. This increasing demand requires the web servers to provide reliable service in a dynamic environment. Abdelraker et al. [2] discusses how an Apache web server is modeled using real-time scheduling. Auctioning on the internet is an example where the server requires heavy computing power. The server needs to handle new bids over time, send the current bid amount to the bidders and also decide whether to close an auction or continue with new bids [6,9]. In the above applications, a centralized server reduces the complexity of implementing the system but may become a bottleneck in case of increasing load.

The execution time of a job can vary due to different factors such as input dependent loops, caching and compiler-architecture mapping of the machine [24]. Non-constant execution times cause difficulties in scheduling and dispatching, especially in hard real-time environments. Secondly, modeling the execution time of a job as a number decreases reliability; on the other hand, modeling the execution time as an interval enhances reliability. Thirdly, some job-sets will be declared unschedulable, if worst-case assumptions are made regarding the execution times of jobs [25].

Example (1): Consider the two job system $J = \{J_1, J_2\}$, with start times $\{s_1, s_2\}$, execution times in the set $\{(e_1 \in)[2, 4] \times (e_2 \in)[4, 5]\}$ and the following set of constraints:

- Job J_1 must finish before job J_2 commences; i.e. $s_1 + e_1 \leq s_2$;
- Job J_2 must commence within 1 unit of J_1 finishing; i.e. $s_2 \leq s_1 + e_1 + 1$.

Assuming that $e_1 = 2$ or that $e_1 = 4$ results in infeasible constraint sets. However, $s_1 = 0$, $s_2 = s_1 + e_1$ is a valid schedule; such a schedule is called a partially clairvoyant schedule, since the start time of job J_2 is dependent upon the execution time of job J_1.

Another factor affecting the execution time is the variable processor speed of power aware processors. Power aware processors are being used to prolong the battery life of various embedded applications. Transmeta's LongRun, AMD's PowerNow and Intel's SpeedStep technologies vary the processor voltage or clock frequency to decrease the power consumed by the processors. In these embedded machines, the optimal processor speed is computed and adjusted according to the system load. Building or interfacing real-time systems with such processors further complicates the situation. [1] proposes to decrease energy consumption for real-time systems by readjusting

processor speed and reusing the unused processor cycles mustered when a job finishes before the worst case execution time.

It is important to note that determining the schedulability of a constrained job-set is only part of the problem. The task of the dispatcher is to ensure that the schedulable jobs are commenced at the appropriate point on the time line. If the start times of jobs are constants, then the task of dispatching them is trivial [23]. However, in case of partially clairvoyant schedules, the start time of a job is a function of the execution times of jobs that are scheduled before it and hence dispatching is a non-trivial task. Gerber et al. [5] and Choi et al. [3] present dispatching strategies for partially clairvoyant schedules based on evaluating functions. These dispatch strategies have a complexity of $\Omega(n)$ and may result in *Loss of Dispatchability* (See Sect. 2). Subramani [22] proposes a parallel algorithm with $O(1)$ dispatch time to eliminate *Loss of Dispatchability*. For a job set of size n, the algorithm requires n processors and uses $O(n)$ space on each processor. This strategy provides a tradeoff between the computing time and resources required, i.e., the constraints are met by increasing the resources to compute the interval during which the job can be dispatched. In this paper, we extend the algorithm proposed in [22] to parallel machines with shared memory architecture and a *fixed number of processors*; the number of processors is much smaller than the number of jobs. This approach powers up controllers in real-time systems and ensures that various deadlines are met, this is vital in hard real-time systems require reliability of the system at any cost [19,21]. We explore the dispatchability of job sets with different timing constraints and show that for certain job sets, the dispatcher requires multiple processors to successfully dispatch the job set.

The principal contributions in this paper are as follows:

(a) Evaluating partially clairvoyant dispatchers in CREW-shared memory environments.
(b) Evaluating the scalability of the algorithm with respect to the number of processors.
(c) Studying the effect of execution time on the dispatcher.
(d) Studying the effect of spacing time on the dispatcher.

The rest of this paper is organized as follows: Section 2 describes the partially clairvoyant scheduling and dispatching problems; it is important to note that the thrust of this paper is in the dispatching problem and not in the scheduling problem. Section 3 describes the motivation for our work and the related work in the literature. Section 4 presents the architecture and our dispatching algorithm. Section 5 describes the results of the experiments performed. We summarize our contributions in Sect. 6, providing pointers for future research.

2 Problem Statement

Consider a set of hard real-time jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$. These jobs are ordered, non-preemptive and occur once in each scheduling window; the scheduling window refers to the time interval between the commencement of J_1 and the conclusion of J_n. We use $\vec{\mathbf{s}} = [s_1, s_2, \ldots, s_n]^T$ to denote the start time vector of the jobs and $\vec{\mathbf{e}} = [e_1, e_2, \ldots, e_n]^T$ to denote the execution time vector of the jobs. The execution time of the

job J_i is known to vary in the interval $[l_i, u_i]$. The jobs are constrained by complex timing constraints; these constraints are represented by the system:

$$\mathbf{A} \cdot [\vec{\mathbf{s}} \ \ \vec{\mathbf{e}}]^{\mathbf{T}} \leq \vec{\mathbf{b}}, \tag{1}$$

where,

- $\mathbf{A}$ is an $m \times 2 \cdot n$ rational matrix; we assume the set of constraints imposed on the jobs to be strict difference constraints.
- $\vec{\mathbf{b}}$ is an m-vector of rationals.

A typical constraint will be of the form: J_1 should finish before J_2 starts; that would result in the linear constraint: $s_1 + e_1 \leq s_2$.

Definition 2.1 A partially clairvoyant schedule of an ordered set of jobs is a vector $\vec{\mathbf{s}} = [s_1, s_2, \ldots, s_n]^T$, where s_i, $1 \leq i \leq n$, is a function of the execution times of jobs that are sequenced before J_i.

Accordingly, the partially clairvoyant query is as follows:

$$\exists s_1 \ \forall e_1 \in [l_1, u_1] \ \exists s_2 \ \forall e_2 \in [l_2, u_2], \ldots \exists s_n \ \forall e_n \in [l_n, u_n] \quad \mathbf{A} \cdot [\vec{\mathbf{s}} \ \ \vec{\mathbf{e}}]^{\mathbf{T}} \leq \vec{\mathbf{b}}? \tag{2}$$

The algorithm in [25] converts the above query into a constraint network and either declares that the system is infeasible, owing to the existence of a negative cost partially clairvoyant cycle, or it determines the dispatch functions for each job J_i. It is important to note that the start time s_i is a function of $\{e_1, e_2, \ldots, e_{i-1}\}$.

Consider a simple example of a robot trying to move an object from one place to another. The speed of the robot depends on the mass of the object and the surface on which the robot is moving and the time the robot takes to change direction depends on the angle it has to turn. Consider the following algorithm followed in a simple motion controller. A robot finds its speed (J_1) by sensing the environment and varies its speed (J_2) according to the requirement, after which the robot finds (J_3) and adjusts (J_4) its direction. Suppose this happens once in every forty units of time with the additional constraints that the robot should start finding the direction of motion between five to ten units of finding its speed. Assume that the robot takes around three to seven units of time to find the speed, five to six units of time to adjust its speed, two to seven units of time to find the direction in which it is moving and eight to twelve units of time to adjust the direction. Assume the constraints imposed on the system are as follows:

- J_1 finishes at least 2 units before J_2 starts.
  $s_1 + e_1 + 2 \leq s_2$
- J_3 starts after the completion of J_2.
  $s_2 + e_2 \leq s_3$
- J_3 starts after 5 units ad before 10 units of the completion of J_1.
  $s_3 \leq s_1 + e_1 + 10$
  $s_1 + e_1 + 5 \leq s_3$

- J_3 finishes at least 5 units before J_4 starts.
  $s_3 + e_3 + 5 \leq s_4$
- J_4 completes within 40 units of time.
  $s_4 + e_4 \leq 40$
- $e_1 \in [3,\ 7]$
- $e_2 \in [5,\ 6]$
- $e_3 \in [2,\ 7]$
- $e_4 \in [8,\ 12]$.

The partially clairvoyant schedule for the example is derived in the Appendix A and is as follows:

1. $0 \leq s_1 \leq 1$
2. $s_1 + e_1 + 2 \leq s_2 \leq 10$
3. $\max(s_1 + e_1 + 5, s_2 + e_2) \leq s_3 \leq \min(s_1 + e_1 + 10,\ 16)$
4. $s_3 + e_3 + 5 \leq s_4 \leq 28$.

Definition 2.2 A feasible partially clairvoyant schedule is said to be dispatchable on a machine M, if for every job J_i, machine M can start executing J_i such that none of the constraints are violated.

Definition 2.3 A safety interval for a job is the time interval during which the job can be started without violating any of the constraints imposed on it.

From the example above, assuming that $s_1 = 0$ and $e_1 = 6$, the safety interval for s_2 is [8,10].

In general, the dispatch functions produced have the following form:

$$\max(f_1,\ f_2,\ \ldots,\ f_{i-1}) \leq s_i \leq \min(f'_1,\ f'_2,\ \ldots,\ f'_{i-1}).$$

where f_j and f'_j are linear functions depending on the start and execution times of job J_j $(j < i)$. Machine M computes the dispatch functions and obtains the safety interval during which the job can be dispatched without violating the constraints. The job J_i is not dispatchable when the computation time exceeds the safety interval. This phenomenon is referred to as *Loss of Dispatchability*.

For the example above, assume the first two jobs take the worst case time and that the first job starts at time $t = 0$, then the third job has the safety interval [15,16]. If the dispatcher takes more than one unit of time to compute the safety interval, then the third job cannot be dispatched.

In this paper, we are concerned with the dispatching problem, i.e., how to compute the safety intervals of the jobs, such that the jobs can be dispatched safely within the proper time intervals, assuming that a partially clairvoyant schedule was obtained from the Query (2). We use a shared memory cluster to compute the safety intervals and attempt to prevent Loss of Dispatchability.

3 Related Work

Subramani [26] proposes the `E-T-C` framework to formalize problems in real-time systems which takes into account the variability of execution time, complex

relationships between jobs and clairvoyance of the system. In [26] various types of constraints and execution time domains capturing the constraints imposed on real-time systems are explored. The problems are formalized and schedulability guarantees for systems with different degrees of clairvoyance are explored.

Saksena [18] introduced partially clairvoyant scheduling to reduce the inflexibility of static scheduling in hard real-time systems. The authors proposed a matrix column elimination method based on Fourier–Motzkin elimination to decide the schedulability of the real-time system and evaluated the performance of the system. Partially clairvoyant scheduling is explained in detail in [5,22,25]. Subramani [25] presents a dual approach to decide the schedulability of a partially clairvoyant system. The author constructs a constraint graph from the given set of constraints and decides the schedulability by contracting vertices and searching for a negative cycle in the resulting constraint graph.

Gerber et al. [5] proposes a sequential online dispatching algorithm. The algorithm stores lists of dispatch functions and has dispatch time linear in the number of jobs. The computation cost of the online dispatcher can cause constraint violation, i.e., the time after computing the safety interval $(l_b,\ r_b)$ exceeds r_b. This phenomenon by which a job cannot be dispatched is called *Loss of Dispatchability*. Subramani [22] proposes a parallel online algorithm for eliminating *Loss of Dispatchability* for partially clairvoyant schedules. The original algorithm proposed in [22] assumes that there are as many processors as the number of jobs n. The jobs are executed on a central processor which writes the values of the start and execution time of the job completed into the shared memory. Each supporting processor reads the start and execution time and computes the safety interval by relaxing the 4 constraints between the job completed and the job assigned to it. After the job assigned to them is executed, the processors idle. This algorithm has $O(1)$ dispatch time per job and uses $O(n)$ space per processor.

The motion controller of a robot requires complex modeling and has to consider various kinematics equations which require different computing times [17,30]. Yang et al. [30] proposes to use neural networks for path planning of a robot in a real-time non stationary environment. Embedded designers are conservative and use 8, 16 or 32-bit processors in most of their applications, which do not have the sophisticated architecture and instruction set support available in modern processors. NASA still uses the reliable IBM RISC6000 chips in some of its projects.

Traffic Alert and Collision Avoidance system (TCAS) is used in commercial aircrafts to avoid collisions. Hull et al. [7] uses imprecise computation techniques to meet the necessary deadlines. Each job is broken into a mandatory job and an optional job; the mandatory jobs are to meet strict end to end deadlines while the optional jobs are scheduled in between with intermediatory deadlines. In such situations, our algorithm of using a parallel dispatcher of jobs is helpful for ensuring that all deadlines are met. The current model can be modified to support parallel execution of jobs on different processors.

Marti et al. [15] proposes a flexible sampling and timing intervals in the control problems. The job modeling strategy used by the authors is similar to our approach, in that, they allow the start time to vary according to the controller but they use the worst case execution times to decide if a schedule exists. There are constraint sets which do not have a schedule in case the worst case execution time is assumed as shown

in [25]. In case the number of parameters of the system increase, the efficiency of the controller decreases due to the heavy computation required. Our algorithm would reduce the load on the controller and would ensure that the controller functions with high efficiency.

Tsigas and Zhang [29] proposes a non-blocking protocol that allows real-time tasks to share data in a multiprocessor system. The protocol is optimal with respect to space requirement and has a lower overhead compared to lock based protocols. The protocol describes a procedure that can be used by concurrent real-time tasks to read and write shared data and the procedure allows multiple write and multiple read operations to be executed concurrently. Tsigas and Zhang [28,29] investigates to determine how the performance and speedup of applications would be affected by using non-blocking rather than blocking synchronization in parallel systems. These papers propose a set of efficient and simple translations that show how typical blocking operations found in parallel applications, such as simple locks, queues and lock trees can be translated into non-blocking equivalents that use hardware primitives common in modern multiprocessor systems.

Traditional scheduling models such as Earliest Deadline First, Rate Monotonic Analysis [12], Priority Ceil Protocol [20] schedule preemptive jobs assuming worst case execution times to determine if a schedule exists. Mok et al. [16] models real-time constraints using an interval as the start time and the worst case execution time. They decide the schedulability of such a system by extending the All-pairs shortest path algorithm for interval timestamps. They also explore situations in distributed applications where the global clock and the local clocks are not synchronized with each other. Kweon and Shin [10] proposes to make real-time communication on the ethernet more predictable by limiting the packet-arrival rate allowed into the Medium Access Control (MAC) layer. The authors analyze the Ethernet MAC protocol using a semi-Markov model and derive a network wide input limit for achieving the desired transmission throughput.

4 Architecture and Algorithm

The Concurrent Read Exclusive Write (CREW) shared memory architecture is discussed in detail in [8]. Each processor has a separate memory in addition to the common shared memory and maintains a copy of the data it requires in its local memory. Any changes to the shared data are made in its local memory and the data is flushed for memory coherence. Memory Coherence depends on the protocol followed [11], i.e., the shared data variable is marked invalid or updated in the other memories as soon as a local copy is changed or a explicit flush command needs to be executed by the processor to achieve memory coherence. A read from the local memory has less memory latency than a remote read (Non-Uniform Memory Access machine). While reading a shared variable, the value resulting from the most recent write is loaded into the local memory.

4.1 Architecture

In the dispatcher, the processors share data with each other through the shared memory as indicated in Fig. 1 and (s_i, e_i) and $(l_{b_{i+1}}, r_{b_{i+1}})$ are shared variables. The central

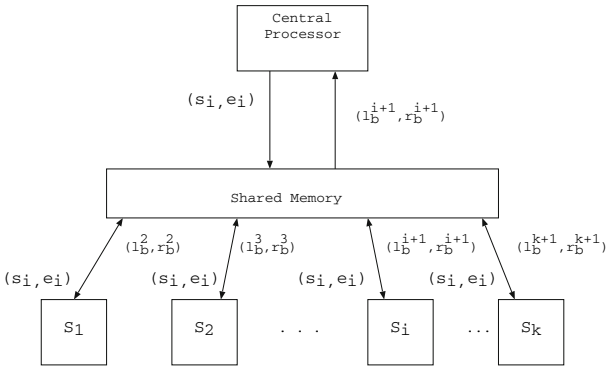

Fig. 1 Shared memory dispatcher architecture

processor C executes a Job J_i and stores $(s_i,\ e_i)$ into the memory. After which C updates a flag $flag_1$ and waits on another flag $flag_2$. Each satellite processor S_j updates and reports the safety intervals for a class of jobs C_j. The satellite processor S_m which has $J_{i+1} \in C_m$ writes the safety interval $(l_{b_{i+1}},\ r_{b_{i+1}})$ in the memory and updates the flag $flag_2$. On updating the safety intervals of all the remaining jobs in their class, the satellite processors wait for flag $flag_1$ to be updated by C.

In this implementation, there are no communication costs as compared to a network distributed model but there is a cost for achieving memory coherence. In the current implementation, the processors are required to flush the start and execution times and the safety intervals for every job, which is an extra overhead. In the current implementation, we assume the jobs are statically distributed among the processors in a cyclic fashion. *In the algorithm, we assume that the satellite processors are waiting on the flag at the instant the central processor completes executing a job.*

4.2 Algorithms

The sequential dispatcher executes a job and then updates the safety intervals of all the jobs having dispatch functions depending on the start and execution time of the completed job. The load on the sequential dispatcher is maximum at the start of the dispatching as in the worst case it has to update the safety intervals of the remaining jobs. The performance of the sequential dispatcher depends on the number of jobs and the minimum duration between the completion of a job and the start of the next job.

In Algorithm (1), the time to update all the safety intervals depending on $(s_i,\ e_i)$ is linear in the number of jobs. Hence Algorithm (1) have a dispatch complexity of $\Omega(n)$. It is important to note that the constrained jobs are provided to the dispatching algorithms as a constraint network $G=<V,\ E>$, where vertex v_i represents job J_i and there is an edge from v_i to v_j with weight $f(e_i, e_j)$, if there is a constraint of the form $s_i - s_j \leq f(e_i, e_j)$, in the input constraint set. Additional details of the constraint network can be obtained from [25].

Algorithm 1 Sequential dispatcher for partially clairvoyant schedules

Function SEQUENTIAL-ONLINE-DISPATCHER ($G =< V, E >$)
1: Let $[l_{b_i}, r_{b_i}]$, $(l_{b_i} < r_{b_i})$ denote the current safety interval of J_i.
2: set current time to 0.
3: **for** ($i = 1$ **to** n) **do**
4: **if** (current-time $< l_{b_i}$) **then**
5: Sleep (l_{b_i}-current-time)
6: **end if**
7: **if** (current-time $\in [l_{b_i}, r_{b_i}]$) **then**
8: Execute job J_i
9: Update all safety intervals depending on (s_i, e_i)
10: Read $(l_{b_{i+1}}, r_{b_{i+1}})$ from memory
11: **else**
12: Return (Schedule is not dispatchable)
13: **end if**
14: **end for**

The shared dispatcher executes a job and then updates the safety intervals of the remaining jobs in parallel. The central processor executes a job and writes the values of the start and execution time to memory. The central processor then waits for the safety interval of the next job to be executed by busy waiting on a flag. The satellite or supporting processors update the safety intervals of the set of jobs assigned to them and write the safety interval of a job to memory when the job is to be executed.

In the shared memory algorithm, after the central processor completes executing a job J_i, the constraints which need to be computed, if they exist, before determining the safety interval $[l_{b_{i+1}}, r_{b_{i+1}}]$ of J_{i+1} are as follows:

1. $s_i + c_1 \leq s_{i+1}$
2. $s_i + e_i + c_2 \leq s_{i+1}$
3. $s_{i+1} \leq s_i + c_3$
4. $s_{i+1} \leq s_i + e_i + c_4$

where c_1, c_2, c_3 and c_4 are real numbers.

Since there are at most 4 constraints between job J_i and J_{i+1}, Algorithm (2) takes at most $O(1)$ time, for each job sequenced before it. As stated in [22], relaxing 4 constraints takes at most 4 additions and comparisons, i.e., $4 \cdot (T_{add} + T_{comp})$, where T_{add} and T_{comp} are the times taken to perform an addition and a comparison, respectively.

Let w_1 be the cost of writing a floating point number to the shared memory. C requires to flush the present values of (s_i, e_i, f_1) to the memory. S_k will have to write $(l_{b_{i+1}}, r_{b_{i+1}}, f_2)$ in the memory.

The time required to compute the safety interval is $4 \cdot (T_{add} + T_{comp}) + 6 \cdot w_1$.

Remark 4.1 The algorithm uses a preset number of processors in contrast to the algorithm proposed in [22], which required n processors.

The Algorithm (2) updates the dispatch functions in parallel to the execution of the next job. Let k be the number of processors. In such a case, each processor has to update constraints between the completed job and a fraction $(=\frac{1}{k})$ of the remaining jobs. In case n is very large, the time required to update the constraints is larger than

Algorithm 2 Shared dispatcher for partially clairvoyant schedules

Function SHARED-ONLINE-DISPATCHER ($G =< V, E >$)
1: Let $[l_{b_i}, r_{b_i}]$, ($l_{b_i} < r_{b_i}$) denote the current safety interval of J_i.
2: Let P denote the number of satellite processors.
3: **for** ($i = 1$ **to** n) **in parallel do**
4: **if** (central processor) **then**
5: **if** (current-time $< l_{b_i}$) **then**
6: Sleep (l_{b_i}-current-time)
7: **end if**
8: **if** (current-time $\in [l_{b_i}, r_{b_i}]$) **then**
9: Execute job J_i
10: Save (s_i, e_i) to memory
11: Update $flag_1$ and save to memory
12: Wait till $flag_2$ is updated
13: Read $(l_{b_{i+1}}, r_{b_{i+1}})$ from memory
14: **else**
15: Return (Schedule is not dispatchable)
16: **end if**
17: **end if**
18: **if** (satellite processor S_m) **then**
19: Compute S_k, the satellite processor required to report the safety interval
20: Wait till $flag_1$ is updated
21: Read (s_i, e_i)
22: **if** $S_k = S_m$ **then**
23: Update-constraints($i, i + 1$)
24: Write safety interval to memory
25: Update $flag_2$ and write to memory
26: Update-constraints(i, q) $\forall Jobs J_q \in C_k$
27: **else**
28: Update-constraints(i, q) $\forall Jobs J_q \in C_m$
29: **end if**
30: **end if**
31: **if** ($i = n$) **then**
32: Return (schedule is dispatchable)
33: **end if**
34: **end for**

Algorithm 3 Update function of shared dispatcher

Function UPDATE-CONSTRAINTS(i, q) (s_i, e_i)
1: Relax constraints between J_i and J_q into absolute constraints of J_q.
2: Compare each absolute constraint with the existing safety interval for J_q
3: **if** (new constraint is not redundant) **then**
4: Update Safety Interval ($[l_{b_q}, r_{b_q}]$)
5: **else**
6: Leave the Safety Interval unchanged
7: **end if**

the execution time of the current job and would cause the next job to lose dispatchability. Increasing the number of processors would help dispatchability if the memory coherence cost is not great.

5 Empirical Analysis

5.1 Machine Description

Our experiments were conducted on a SGI Origin 2000 in a shared environment with load sharing. The dispatcher sets the number of threads that run in parallel. In the best case, one thread will run on one processor; however, in most cases multiple threads are scheduled on a single processor.

The hardware specification of the machine and environment are listed in Tables 1 and 2, respectively. The jobs were submitted in the batch queue.

5.2 Generation of Partially Clairvoyant Schedules

A test-case is a set of jobs with execution time belonging to a certain time period and several constraints between the jobs. The duration between two adjoint jobs is capped by creating constraints depending on the start and execution times of the first job. We create test-cases by varying the number of jobs or the execution time period or the threshold value of the cap. The dual algorithm [25] generates partially clairvoyant schedules from these test-cases. For the purpose of testing the dispatchers, we create constraints which will ensure that a schedule exists.

A detailed description of the parameters required for the schedule generation are described in Sect. 5.2.1. The procedure followed by the schedule generating algorithm GA is described in the Sect. 5.2.2.

Table 1 Machine specifications of SGI Origin2000 of NCSA

Component	Description
Architecture	Distributed shared memory
Processors	MIPS R10000
Available number of processors	64 (or 128)
Clock speed	250 MHz or 195 MHz
Instruction cache size	32 Kbytes
Data cache size	32 Kbytes
User virtual address space	4 GB
Interconnect between machines	Gigabit Ethernet

Table 2 Software

Component	Description
Operating system	Irix 6.5
Compiler	C
Programming models	OpenMP
Floating point format	IEEE
Batch system	Load sharing batch system

5.2.1 Parameters

The parameters required by the generation algorithm are as follows:

- Number of jobs n: The number of jobs in the schedule.
- Execution time $[l, u]$: The lower and upper limit of the execution time of the jobs.
- Spacing time $[p, q]$: This is used to create constraints which would ensure that the next job would begin between $[p, q]$ seconds after the completion of a job. The value p prevents constraints which force the two jobs to be very close to each other while q prevents a large interval between the two jobs. The upper bound is necessary to force jobs close together and test the response time of the dispatcher. The lower bound is necessary to prevent constraints which would require jobs to be separated by time shorter than a fewer microseconds.
  If $p = 2$ and $q = 6$, we would create constraints which would force jobs to be separated by at least 2 units of time and by at most 6 units of time.
- Number of constraints E: The number of standard constraints between jobs.

where l, u, p and q are real numbers.

5.2.2 Constraint Generation

We specify the number of jobs n, the number of constraints E, the execution time $[l, u]$ and the spacing time $[p, q]$. We also specify a random seed for generating the constraints. The generating algorithm GA does as follows:

- For each job, GA generates and prints two numbers between l and u ($l < u$), which bound the execution time of the job.
- Between every job J_i and $J_{i+1}(1 \leq i \leq n-1)$, GA generates standard constraints of the form $s_i + e_i \leq s_{i+1}$ and $s_{i+1} \leq s_i + e_i + c$ where c is a random number between the p and q. For any test-case, there are least $2 \cdot n$ constraints generated.
- If $E > 2 \cdot n$ then $(E - 2 \cdot n)$ constraints between the finish times of two randomly chosen jobs (say J_x and J_y, $1 \leq x, y \leq n$) are generated such that a partially clairvoyant schedule would exist. If $x < y$ then a small negative real number $c_1 < l$ is generated such that $s_x + e_x \leq s_y + e_y + c_1$ is true; and if $x > y$ then a very large real number c_2 is generated such that $s_x + e_x \leq s_y + e_y + c_2$ is trivially true.

We increase the time to update safety intervals on the satellite processors by choosing a large value of E.

5.2.3 Schedule Generation

A constraint graph can be constructed from a test-case [25], as the constraints imposed on the system are strict difference constraints. The dual algorithm in [25] was used to decide the schedulability of the system and produce the dispatch functions.

5.2.4 Schedule Execution

The dispatcher takes as input the number of jobs, execution time periods, a random seed and the dispatch functions. The dispatch functions are stored in a two-dimensional

triangular array. Arrays are maintained to store the start time, execution time and execution time periods of the jobs.

For each job J_i, a random number t' between the execution time bounds $[l_i, u_i]$ is generated. A job is executed by the central processor C only if the time at the start of the job is between the safety interval of the job. The central processor C simulates the execution of a job by a busy wait for time t'. The central processor C then updates the flag $flag_1$ and writes the triplet $(s_i,\ e_i,\ f_1)$ to the memory. Following which, the central processor C waits for flag $flag_2$ to be updated before reading the safety interval of the next job. We use the function gettimeofday() to find the time T_{bef} before writing $(s_i,\ e_i,\ f_1)$ into the shared memory and the time T_{aft} after reading the safety interval $(l_{b_{i+1}},\ r_{b_{i+1}})$ from the memory. The time difference $T_{aft} - T_{bef}$ gives the time required to update and report the safety interval to C. In our simulation, time starts at time $t = 0$ and proceeds by adding $s_i + e_i + T_{aft} - T_{bef}$ to obtain the time at the instant C is ready to execute the job J_{i+1}.

5.3 Runtime Approximations

The time required to write the start time, execution time and the flag in the memory depends on various factors like the load on the processors, the system bus, processors accessing the shared data, etc., The time the central processor C waits to read the safety interval depends on the time, the concerned satellite processor takes to read the updated flag $flag_1$, compute the dispatch functions and update flag $flag_2$.

With the execution of jobs, the number of constraints to be updated decreases and the update times should decrease. We measured the update time required by the dispatcher using different number of processors and plotted histograms showing the frequency of the update times in certain regions. Figure 2 shows the frequency of the observed update times in various intervals while dispatching a job set of 5,000 jobs on 16 processors. We observed that the update times are heavily concentrated in a certain time interval and that a few discrepancies are located far away from this

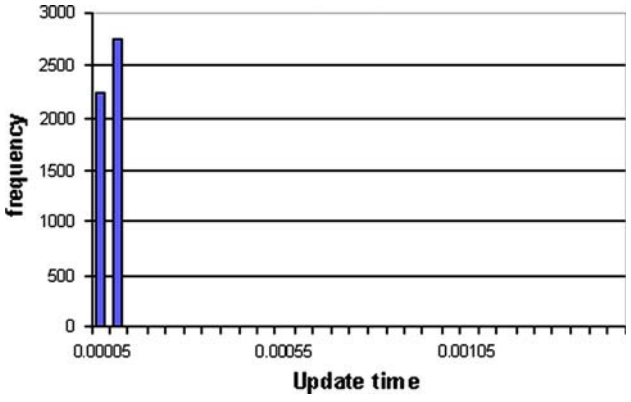

Fig. 2 Observed update time frequency for 5,000 jobs on 16 processors

interval. These discrepancies occur due to external factors such as the load on the batch system and the scheduling policy of the Iris operating system.

We performed many experiments and observed that the maximum difference between the frequent update times is four times the mean update time. We neglect these large update times by checking if the current update time is greater than 4 times the previous update time and neglect these extreme values. In our results, we neglect the large update times by taking the previous update time for at most five consecutive overshoots. After five consecutive overshoots, we do not consider the observed update time to be due to external factors. *These overshoots were neglected as real-time systems require and use dedicated machines.*

5.4 Results

We set the maximum number of threads to run in parallel in our experiments. The optimal condition for the dispatcher would be for one thread to execute on one processor alone, equivalent to setting the desired number of processors. In all other cases, there are multiple threads running on a processor or there are fewer threads running in parallel. The outcome in such cases is increase in the reliability of the dispatcher.

The dispatcher stores the dispatch functions as constraints in two triangular arrays. We created constraints to ensure that the duration between the finish time of a job and the start time of the next job is in $[p,\ q]$. The update time of the sequential dispatcher increases linearly with the number of jobs. While dispatching a certain number of jobs N_1, the update time is greater than the spacing time and the job set would no longer be dispatchable.

We conducted experiments to measure the update time of the sequential and the shared memory dispatchers. The sequential dispatcher can either update all the existing constraints depending on the start and execution time of the completed job or follow a lazy approach for computing the safety intervals. In either case, the number of constraints that are required to be updated will be linear in the number of jobs.

For the sequential dispatcher, the difference between T_{bef} (the time after completing a job) and T_{aft} (the time after completing the update of all the dispatch functions) gives the update time. While in the case of the shared dispatcher, T_{bef} is the time before writing the start and execution time of a completed job to memory and T_{aft} is the time after reading the safety interval of the next job. The shared dispatcher takes around the same update time for any number of jobs as it includes the time required to read $(s_i,\ e_i,\ f_1)$ and update 4 constraints between J_i and J_{i+1} and write back $(l_{b_{i+1}},\ r_{b_{i+1}},\ f_2)$. The satellite processors of the shared dispatcher update the remaining constraints in parallel while the central processor is executing job J_{i+1}.

Figures 3 and 4 plot the update time in seconds versus the number of jobs in the schedule. Figure 3 compares the update time of the sequential dispatcher and the shared dispatcher with two processors. Figure 3 shows that the update time of the sequential dispatcher increases with the number of jobs while that of the shared dispatcher with two processors is almost constant with the number of jobs and takes at most 2.5×10^{-5} s to find the safety interval of the next job.

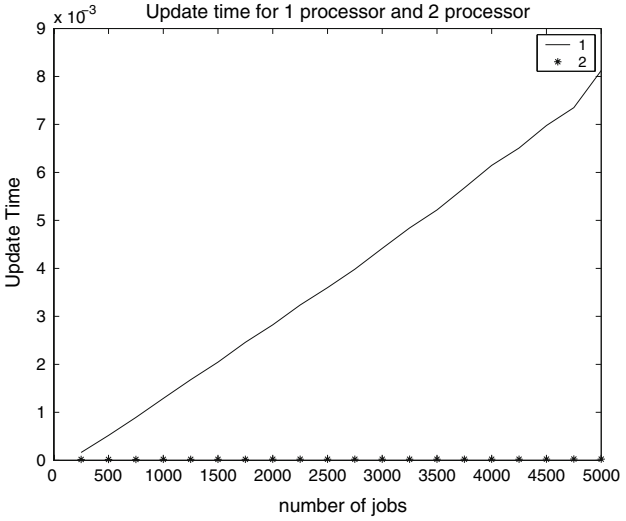

Fig. 3 Update time of sequential dispatcher and a 2 processor dispatcher for schedules

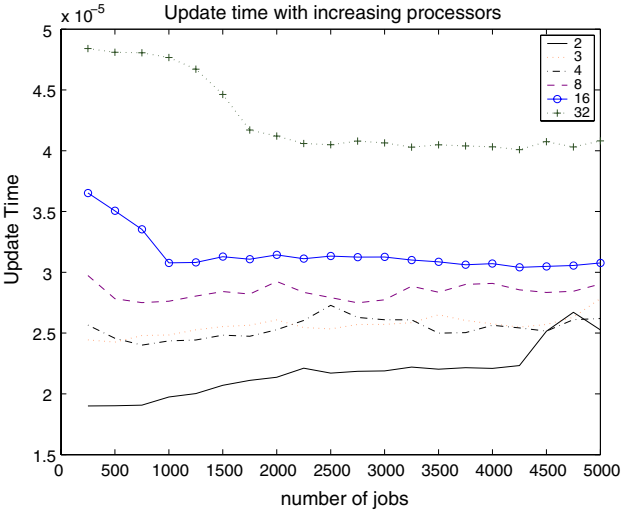

Fig. 4 Update time taken by shared dispatcher using different number of processors

Using best fit, we calculated the equation of line passing through the single processor update times. We obtained $a = 1.52 \times 10^{-6}$, $b = -2.214 \times 10^{-4}$ for the line $y = a \cdot x + b$. Using the above line and the maximum update time of the two processor shared dispatcher, we numerically estimated that after 163 jobs the shared memory dispatcher with two processors is superior to the sequential dispatcher.

Figure 4 plots the update time taken by the shared memory dispatcher for job sets of different size with varying number of processors. We observe that the update time of

Table 3 Results of dispatching schedules of different size by the sequential and shared dispatcher

Processors	Number of jobs							
	250	500	750	1,000	2,000	3,000	4,000	5,000
1	√	√	√	√	×	×	×	×
2	√	√	√	√	√	√	√	√
3	√	√	√	√	√	√	√	√
4	√	√	√	√	√	√	√	√

√ is when the schedule was successfully dispatched and × was not. $[l, u] = [1\,\text{ms}, 5\,\text{ms}]$; $[p, q] = [1\,\text{ms}, 5\,\text{ms}]$

the shared dispatcher is almost constant and in the range of 10^{-5} s. These figures show that the shared memory dispatcher has a much smaller update time than the sequential dispatcher.

Figure 4 shows that the update time of the shared dispatcher increases with the number of processors. This can be accounted to the increased contention to obtain a lock on the shared data as processors are augmented. In case the number of jobs increase, the number of safety intervals to be updated increase and the satellite processors would have an update time linearly increasing with the jobs as the sequential dispatcher in Fig. 3. The increase in the update time would make our assumption in Sect. 4.1 void as the satellite processors would be updating safety intervals when the central processor C updates flag $flag_1$ after completing a job. Following which, the dispatcher would not be able to dispatch the next job if the concerned satellite processor was delayed. On increasing the number of satellite processors, the average number of safety intervals to update per processor decreases and hence the update time; thereby maintaining our assumption in Sect. 4.1. We should increase the number of processors till the time to achieve memory coherence is less that the spacing time, after which the assumption will not be valid.

In Fig. 3 time is represented in 10^{-3} s, whereas in Fig. 4 time is represented in 10^{-5} s; this explains why the update time when there are two processors appears negligible in Fig. 3 and tangible in Fig. 4.

We tested the shared and sequential dispatcher with job sets of different sizes. These experiments were repeated three times with different random seeds. Table 3 summarizes the results of the experiments performed. Similar results were observed in the three cases, i.e., the schedules were dispatchable or not dispatchable. In all the three cases, the sequential dispatcher broke at different jobs as it took longer to update the safety intervals of all the jobs. While in the case of the shared dispatcher, the update times were small and the jobs were dispatched in their safety intervals.

5.4.1 Scalability

In this section, we intend to show the effect of increasing the number of processors on the dispatchability of job sets. We generated partially clairvoyant schedules of jobs, with the number of jobs increasing from 1,000 up to 9,750. The jobs in the schedules

Table 4 Scalability of the shared dispatcher

Processors	Maximum size of job set dispatched
1	100
2	2,500
3	3,500
4	4,250
5	5,500
6	6,500
7	8,750
8	8,500
9	9,250
10	9,750

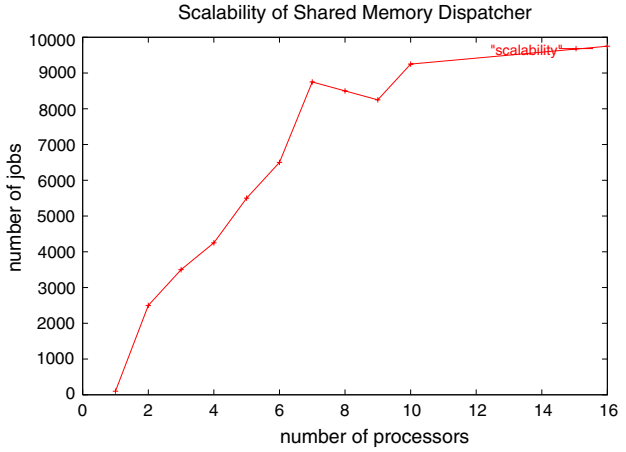

Fig. 5 The number of jobs that can be successfully dispatched by a given number of processors, where the job execution time was between 1 and 5 ms and the spacing time was between 0.1 and 0.5 ms

have execution time periods varying between one to five milliseconds and the spacing time between two adjacent jobs to be between one-tenth to one-half a millisecond.

In our experiments, we varied the number of processors to be used by the dispatcher and find the largest job set successfully dispatched by the dispatcher. Table 4 summarizes the results of the experiments conducted and shows the largest job set up to which all the schedules were dispatched with a certain number of processors. We observed that the dispatcher with fewer processors failed as the size of the job set increased. An increase in the size of the job set causes the update time on the satellite processors to increase which in turn delays the satellite processors from reading the start and execution time of the next job as soon as it is stored in the memory by the central processor. Figure 5 plots the largest job set dispatched by a given set of processors and shows the scalability of the shared memory algorithm. In our experiments, the largest test-case created had 9,750 jobs and with 10 processors all the schedules were dispatched successfully.

In Fig. 5, we observe that the slope of the curve decreases for schedules of larger size even when the number of processors are increased. While updating the safety intervals, the satellite processors need to access different locations of the array causing frequent cache misses and page faults. As the size of the schedule increases, the satellite processors need to access memory locations in the main memory for every constraint they relax. The number of read and write operations to the main memory increase and slow down the updating process. After a certain number of jobs, the time to relax the constraints takes longer and causes the concerned satellite processor to be significantly delayed in reading the start and execution time of the completed job. This shows that the memory to processor bus latency would form a bottleneck in this architecture and that increasing the processors would not help when the memory latency and the spacing time are of the same order. Another reason for the increase in latency is the increasing number of read/write requests from all the processors. There is limited bandwidth between the shared memory and the processors which would prevent all the processors from accessing and updating the memory at the same time. This shows the requirement of high speed connections and high memory bandwidth for the dispatchability of the schedules.

Experiments were also conducted using 32 processors and observed that they fail for job sets dispatched by 7 processors. The update time for 32 processors as observed in Fig. 4 is comparable to the spacing time of the test cases and here, the memory coherence time is significant.

5.4.2 Effect of Execution Time

An increase in the execution time of the jobs would give the satellite processors more time to update the safety intervals making it easier to uphold our assumption in Sect. 4.1. A decrease in the execution time reduces the parallel update time making it harder to uphold our assumption in Sect. 4.1. Hence longer the job execution time, larger is the size of the schedule that can be dispatched.

In the test cases we created, the spacing time between two adjacent jobs was set between one-tenth to one-half of a millisecond and execution time was varied to be [0.1 ms, 0.5 ms], [0.5 ms, 1 ms], [1 ms, 5 ms] and [5 ms, 10 ms]. The number of jobs in the schedule were increased from 250 to 5, 000. Experiments were conducted by setting the number of processors to be used by the dispatcher and finding the size of the largest schedule successfully dispatched. Each entry in Table 5 indicates the maximum number of jobs that were dispatched successfully with the number of processors, when the execution time of jobs is in the given range. The sequential dispatcher was not able to dispatch any job set used in these experiments.

The observed results are listed in Table 5 and are in excellent co-ordination with the expected behavior. A table entry of 5,000 implies that the dispatcher dispatched all the schedules created and can dispatch larger schedules.

Figure 6 plots the largest job sets dispatched with a given number of processors for four different intervals of job execution time. Clearly, schedules with higher job execution time intervals got dispatched with lesser number of processors. From the Fig. 6, we conclude that greater the execution time, greater is the number of jobs that can be dispatched by the shared dispatcher.

Table 5 Spacing versus execution time of jobs on the shared dispatcher assuming a spacing time interval [0.1 ms, 0.5 ms]

Execution time (ms)	Processors											
	2	3	4	5	6	7	8	9	12	16	20	24
5–10	5,000	5,000	5,000	5,000	5,000	5,000	5,000	5,000	5,000	5,000	5,000	5,000
1–5	2,500	3,500	4,250	5,000	5,000	5,000	5,000	5,000	5,000	5,000	5,000	5,000
0.5–1	750	1,750	2,250	2,750	3,250	3,750	4,000	4,750	5,000	5,000	5,000	5,000
0.1–0.5	750	750	1,000	1,000	1,500	1,500	1,750	1,750	2,750	3,500	3,500	4,750

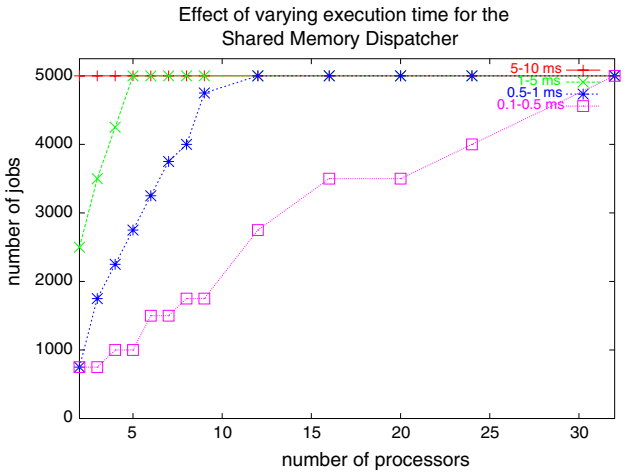

Fig. 6 For a fixed execution time, increasing the number of processors causes larger job-sets to be dispatched

5.4.3 Effect of Spacing Time

On decreasing the spacing time, the allowed time between the finish time of a job and the start time of the next job is decreased. The satellite processors need to compute the safety interval of the next job within this time as soon as it is written by the central processor making the assumption in Sect. 4.1 a necessity to uphold. The satellite processors need to complete updating all the safety intervals by the time the central processor completes executing the next job.

We generated partially clairvoyant schedules with spacing times of [0.1 ms, 0.5 ms], [0.5 ms, 1 ms] and [1 ms, 5 ms]. For each spacing time, we generated schedules with the number of jobs increasing from 250 to 5,000 and execution times of [0.1ms, 0.5 ms], [0.5 ms, 1 ms], [1 ms, 5 ms] and [5 ms, 10 ms]. We fixed the number of processors to be used by the dispatcher and empirically determined the size largest schedule successfully dispatched by the dispatcher. Table 6 summarizes the results of the experiments conducted.

Table 6 Effect of varying the spacing time and execution time on the shared dispatcher

Processors	Spacing time	Execution time			
		5–10 ms	1–5 ms	0.5–1 ms	0.1–0.5 ms
2	0.1–0.5	5,000	2,500	750	750
	0.5–1	5,000	3,000	1,000	750
	1–5	5,000	2,000	1,500	1,500
3	0.1–0.5	5,000	3,500	1,750	750
	0.5–1	5,000	4,000	1,500	1,250
	1–5	5,000	4,000	2,750	1,750
4	0.1–0.5	5,000	4,250	2,250	1,000
	0.5–1	5,000	4,000	2,750	1,500
	1–5	5,000	5,000	3,000	2,000
5	0.1–0.5	5,000	5,000	2,750	1,000
	0.5–1	5,000	5,000	3,500	2,000
	1–5	5,000	4,000	4,000	2,250
6	0.1–0.5	5,000	5,000	3,250	1,500
	0.5–1	5,000	5,000	4,250	2,000
	1–5	5000	5,000	5,000	2,750
7	0.1–0.5	5,000	5,000	3,750	1,500
	0.5–1	5,000	5,000	4,750	2,750
	1–5	5,000	5,000	5,000	3,250
8	0.1–0.5	5,000	5,000	4,000	1,750
	0.5–1	5,000	5,000	5,000	3,000
	1–5	5,000	5,000	5,000	3,250
10	0.1–0.5	5,000	5,000	4,750	1,750
	0.5–1	5,000	5,000	5,000	4,000
	1–5	5,000	5,000	5,000	4,750
12	0.1–0.5	5,000	5,000	5,000	2,750
	0.5–1	5,000	5,000	5,000	4,500
	1–5	5,000	5,000	5,000	4,750
16	0.1–0.5	5,000	5,000	5,000	3,500
	0.5–1	5,000	5,000	5,000	5,000
	1–5	5,000	5,000	5,000	5,000

We plotted the largest job set that could be dispatched for different values of spacing time versus the number of processors in Fig. 7. Figure 7 shows that the size of the job set dispatched increases with increasing the spacing time. We wanted to show the effect of execution time of jobs with spacing time. The non-crossed lines show job sets with higher execution time of [0.5 ms, 1 ms] while the crossed lines have execution time of [0.1 ms, 0.5 ms]. Table 6 and Fig. 7 show that increasing the spacing time will allow schedules of larger size to be dispatched using the same number of processors and also that increasing the execution times of the jobs increases the number of jobs dispatched for a given value of the spacing time.

6 Conclusion

In this paper, we implemented a partially clairvoyant dispatcher using a limited number of processors and demonstrated that the shared memory dispatcher performs better

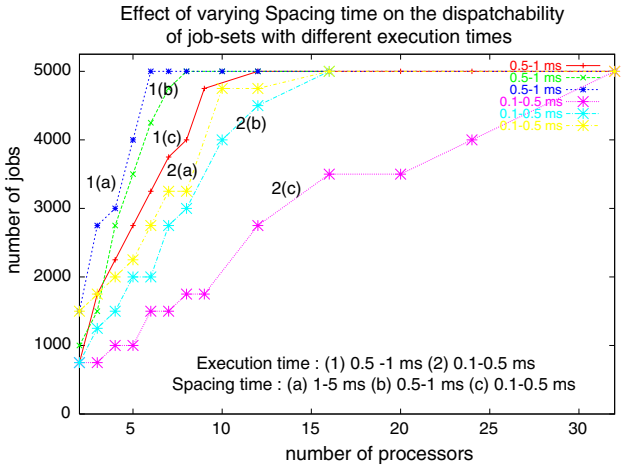

Fig. 7 Increasing the spacing time between jobs causes larger job-sets to be dispatched, with the same number of processors

than the sequential dispatcher in situations where the computation time is greater than or comparable to the memory flush time (time to achieve memory coherence) of the processors. On the whole, our approach is targeted to power up the controller by providing it with more processing power.

The constraints on the available computing time between the end of a job and the beginning of the next one can cause *Loss of Dispatchability*. When the available computing time between two jobs is less, the shared memory dispatcher is better for large job sets. In case the shared memory dispatcher cannot dispatch the schedule while updating constraints, the sequential algorithm would definitely fail.

The sequential dispatcher is not affected by schedules containing jobs having small execution time. The performance of the sequential dispatcher depends only on the time required to compute the safety intervals. However, the shared memory dispatchers suffer as the available parallel computing time decreases. In this case, the sequential dispatcher is a better choice for small job sets.

We demonstrated the scalability of the dispatcher showing that larger job sets can be dispatched by increasing the number of processors provided we have sufficient memory bandwidth.

In the future, we intend to study the effects of using non-blocking reads and writes while accessing the memory as proposed in [27]. We shall look at constructing a hybrid model using the constraint sets in [16] and variable execution time and also at constructing a theoretical model using the job set, constraints to predict the number of processors required to successfully dispatch the job set off line.

Acknowledgements K. Subramani research was supported in part by the Air-Force Office of Scientific Research under Grant FA9550-06-1-0050. K. Yellajyosula work was partially supported by National Computational Science Alliance under [ASC30006N] and utilized the account [kirany].

Appendix A: Obtaining Dispatch Functions

In the following section, we show how the set of dispatch functions are obtained from a given set of constraints using the Dual algorithm described in [25]. We use the example in Sect. 2 as a working example.

- J_1 finishes at least 2 units before J_2 starts.
  $s_1 + e_1 + 2 \leq s_2$
- J_3 starts after the completion of J_2.
  $s_2 + e_2 \leq s_3$
- J_3 starts after 5 units ad before 10 units of the completion of J_1.
  $s_3 \leq s_1 + e_1 + 10$
  $s_1 + e_1 + 5 \leq s_3$
- J_3 finishes at least 5 units before J_4 starts.
  $s_3 + e_3 + 5 \leq s_4$
- J_4 completes within 40 units of time.
  $s_4 + e_4 \leq 40$

The formulation proceeds as follows:

- $\mathcal{J} = \{J_1,\ J_2,\ J_3,\ J_4\}$.
- – $e_1 \in [3,\ 7]$
  – $e_2 \in [5,\ 6]$
  – $e_3 \in [2,\ 7]$
  – $e_4 \in [8,\ 12]$
- 1. $s_1 \geq 0$
  2. $s_1 + e_1 + 2 \leq s_2$
  3. $s_2 + e_2 \leq s_3$
  4. $s_3 \leq s_1 + e_1 + 10$
  5. $s_1 + e_1 + 5 \leq s_3$
  6. $s_3 + e_3 + 5 \leq s_4$
  7. $s_4 + e_4 \leq 40$

- Iteration 1:
  – Elimination of e_4: After substituting e_4 in the constraints, the set of constraints are:
    1. $s_1 \geq 0$
    2. $s_1 + e_1 + 2 \leq s_2$
    3. $s_2 + e_2 \leq s_3$
    4. $s_3 \leq s_1 + e_1 + 10$
    5. $s_1 + e_1 + 5 \leq s_3$
    6. $s_3 + e_3 + 5 \leq s_4$
    7. $s_4 \leq 28$
  – Dispatch functions for J_4:
    * $s_3 + e_3 + 5 \leq s_4 \leq 28$
  – Elimination of s_4: The set of constraints after eliminating s_4 are as follows:
    1. $s_1 \geq 0$
    2. $s_1 + e_1 + 2 \leq s_2$

3. $s_2 + e_2 \leq s_3$
4. $s_3 \leq s_1 + e_1 + 10$
5. $s_1 + e_1 + 5 \leq s_3$
6. $s_3 + e_3 + 5 \leq 28$

- Iteration 2:
  - Elimination of e_3:
    1. $s_1 \geq 0$
    2. $s_1 + e_1 + 2 \leq s_2$
    3. $s_2 + e_2 \leq s_3$
    4. $s_3 \leq s_1 + e_1 + 10$
    5. $s_1 + e_1 + 5 \leq s_3$
    6. $s_3 \leq 16$
  - Dispatch functions for J_3:
    * $s_1 + e_1 + 5 \leq s_3 \leq 16$
  - Elimination of s_3:
    1. $s_1 \geq 0$
    2. $s_1 + e_1 + 2 \leq s_2$
    3. $s_2 + e_2 \leq s_1 + e_1 + 10$
    4. $s_1 + e_1 + 5 \leq s_1 + e_1 + 10$
    5. $s_1 + e_1 \leq 11$
    6. $s_2 + e_2 \leq 16$

  The constraint $s_1 + e_1 + 5 \leq s_1 + e_1 + 10$ is redundant and is eliminated as it is trivially true.
- Iteration 3:
  - Elimination of e_2:
    1. $s_1 \geq 0$
    2. $s_1 + e_1 + 2 \leq s_2$
    3. $s_2 \leq s_1 + e_1 + 4$
    4. $s_1 + e_1 \leq 11$
    5. $s_2 \leq 10$
  - Dispatch functions for J_2:
    * $s_1 + e_1 + 2 \leq s_2 \leq \min(10, s_1 + e_1 + 4)$
  - Elimination of s_2:
    1. $s_1 \geq 0$
    2. $s_1 + e_1 + 2 \leq s_1 + e_1 + 4$
    3. $s_1 + e_1 + 2 \leq 10$
    4. $s_1 + e_1 \leq 11$
- Iteration 0:
  - Elimination of e_1:
    1. $s_1 \geq 0$
    2. $s_1 \leq 1$
    3. $s_1 \leq 4$
  - Dispatch functions for J_1:
    * $0 \leq s_1 \leq \min(1, 4)$

References

1. Aydin, H., Melhem, R., Mossé, D., Mejía-Alvarez, P.: Dynamic and aggressive scheduling techniques for power-aware real-time systems. In: The 22nd IEEE Real-Time Systems Symposium (RTSS '01), pp 95–105. IEEE, Washington, Brussels, Tokyo (December 2001)
2. Abdelzaher, T.F., Shin, K.G., Bhatti, N.: Performance guarantees for Web server end-systems: a control-theoretical approach. IEEE Trans. Parallel Distri. Syst. **13**(1), 80–96 (2002)
3. Choi, S., Agrawala, A.K.: Dynamic dispatching of cyclic real-time tasks with relative timing constraints. Real-Time Systems **19**(1), 5–40 (2000)
4. Chodrow, S.E., Jahanian, F., Donner, M.: Run-time monitoring of real-time systems. In: R. Werner (ed.) Proceedings of the Real-Time Systems Symposium — 1991, pp. 74–83. IEEE Computer Society Press, San Antonio, Texas, USA (December 1991)
5. Gerber, R., Pugh, W., Saksena, M.: Parametric dispatching of hard real-time tasks. IEEE Trans. Comput. **44**(3), 471–479 (1995)
6. Guo, X.: An optimal strategy for sellers in an online auction. ACM Trans. Internet Technol. (TOIT) **2**(1), 1–13 (2002)
7. Hull, D.L., Feng, W., Liu, J.W.-S.: Enhancing the performance and dependability of real-time systems. In: Proceedings of International Computer Performance and Dependability Symposium (IPDS'95), pp. 174–182. IEEE Computer Society (April 1995)
8. Ja'Ja', J.: An introduction to parallel algorithms (contents). SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory) **23** (1992)
9. Konana, P., Mok, A.K., Lee, C.-G., Woo, H., Liu, G.: Implementation and performance evaluation of a real-time E-brokerage system. In: Proceedings of the 21st Symposium on Real-Time Systems (RSS-00), p 13. IEEE Computer Society, Los Alamitos, CA, (November 27–30, 2000)
10. Kweon, S.-K., Shin, K.G.: Statistical real-time communication over ethernet for manufacturing automation systems. IEEE Trans. Parallel and Distri. Comput. **14**(3), 322–335 (2003)
11. Li, K., Hudak, P.: Memory coherence in shared virtual memory systems. ACM Trans. Comput. Syst. **7**(4), 321–359 (1989)
12. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in hard real-time environment. J. ACM, **20** (1973)
13. Levi, S.T., Tripathi, S.K., Carson, S.D., Agrawala, A.K.: The `Maruti` hard real-time operating system. ACM Special Interest Group Operat. Syst. **23**(3), 90–106 (1989)
14. Mosse, D., Agrawala, A.K., Tripathi, S.K.: Maruti a hard real-time operating system. In: Second IEEE Workshop on Experimental Distributed Systems, pp. 29–34. IEEE (1990)
15. Marti, P., Fuertes, J.M., Fohler, G., Ramamritham, K.: Improving quality-of-control using flexible timing constraints: metric and scheduling issues. In: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02), pp. 91–100. IEEE Computer Society Press (2002)
16. Mok, A.K., Lee, C.-G., Woo, H., Konana, P.: The monitoring of timing constraints on time intervals. In: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02), pp. 191–200. IEEE Computer Society Press (2002)
17. Rybski, P.E., Gini, M., Hougen, D.F., Stoeter, S.A., Papanikolopoulos, N.: A distributed surveillance task using miniature robots. In: Gini, M., Ishida, T., Castelfranchi, C., Lewis J.W., (eds.) Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02), pp. 1393–1394. ACM Press (July 2002)
18. Saksena, M.: Parametric Scheduling in Hard Real-Time Systems. PhD thesis, University of Maryland, College Park (June 1994)
19. Schiebe, M., Pferrer, S. (eds.): Real-Time Systems Engineering and Applications, vol. 1. Kluwer Academic Publishers (1992)
20. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans. Comput. **39**(9), 1175–1185 (1990)
21. Stankovic, J.A., Spuri, M., Ramamritham, K., Buttazzo G.C. (eds.): Deadline Scheduling for Real-Time Systems. Kluwer Academic Publishers (1998)
22. Subramani, K.: Duality in the Parametric Polytope and its Applications to a Scheduling Problem. PhD thesis, University of Maryland, College Park (August 2000)
23. Subramani, K.: An analysis of zero-clairvoyant scheduling. In: Katoen, J.-P., Stevens, P. (eds.) Proceedings of the 8th International Conference on Tools and Algorithms for the construction of Systems (TACAS), vol. 2280 of Lecture Notes in Computer Science, pp. 98–112. Springer-Verlag (April 2002)

24. Subramani, K.: A specification framework for real-time scheduling. In: Grosky, W.I., Plasil, F. (eds.) Proceedings of the 29th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM), vol. 2540 of Lecture Notes in Computer Science, pp. 195–207. Springer-Verlag (November 2002)
25. Subramani, K.: An analysis of partially clairvoyant scheduling. J. Math. Model. Algorithms **2**(2), 97–119 (2003)
26. Subramani, K.: A comprehensive framework for specifying clairvoyance, constraints and periodicty in real-time scheduling. Comput. J. **48**(3), 259–272 (2005)
27. Tsigas, P., Zhang, Y.: Non-blocking data sharing in multiprocessor real-time systems. In: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, p. 247. IEEE Computer Society (1999)
28. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In: Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 134–143. SIGACT/SIGARCH and EATCS, Crete Island, Greece (July 3–6, 2001)
29. Tsigas, P., Zhang, Y.: Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In: Proceedings of the 3rd International Workshop on Software and Performance (WOSP-02), pp. 55–67. ACM Press, New York (July 24–26, 2002)
30. Yang, S.X., Guangfeng, Y., Meng, M.: Real-time collision-free path planning and tracking control of a nonholonomic mobile robot using a biologically inspired approach. In: Proceedings of Computational Intelligence in Robotics and Automation, pp. 113–118. IEEE Computer Society (2001)