RC24167 (W0701-150) January 31, 2007
Computer Science

IBM Research Report

Delay-Cost Scheduling for Multithreaded, Multicore Machines

Peter A. Franaszek, Dan Poff
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Delay-Cost Scheduling for Multithreaded, Multicore Machines

Peter A. Franaszek and Dan Poff

IBM Thomas J. Watson Research Center
Yorktown Heights, New York, USA

Abstract. We outline a generalization of the delay-cost objective function approach employed in the IBM System I scheduler to systems with possibly large numbers of multithreaded machines. We concentrate on hypervisor scheduling of guest operating systems, although some discussion is included of OS scheduling in such systems, and the results may be applicable in larger contexts, such as management of computer installations. Two main differences between the issues treated here and current schedulers are a) the inclusion of coscheduling of multiple logical processors associated with a guest OS, and b) the inclusion of multiple ways to satisfy a request for logical processing resources, with different utilizations of hardware threads and electrical power as a function of the application. A generalized delay-cost objective function is described, as well as two versions of, or approaches to optimality, respectively the maximization of the objective function, and maximization of this function subject to a fairness criterion. Allocation of resources to jobs is done in two phases: first an amount is determined, and secondly this is matched with physical entities. The report includes a development and discussion of the generalized delay-cost function, the algorithms used in scheduling, as well as a variety of experimental results which suggest that the overall approach is reasonable, since both the speed of execution, as well as the system power utilization, can depend on the number and identity of the system resources utilized by specific applications.

1. Introduction

Delay-cost scheduling [1] is a paradigm successfully applied in IBM's OS400, the operating system for iSeries servers. Its basis is a quasi-economic measure of a time-varying cost which is charged to the system for delaying the processing of a job. The system attempts to minimize its cost by appropriate scheduling. In the case of a single work queue, the job next chosen for processing is that with the highest marginal cost [1,2]. The original development of the delay-cost paradigm for processor scheduling was limited to the case of non-affinity scheduling, and a single work queue. Aside from the treatment of delay-cost scheduling, much of the analysis in [1] was devoted to contrasting this with the deadline scheduler employed in the IBM VM system, which was shown to be unstable under certain conditions. That is, it was shown that in that system, actions taken to improve the response time for a class of jobs could actually delay the processing of some of these jobs. Another aspect of that work was

to show that delay-cost functions could be chosen so as to mimic the behavior of a deadline scheduler operating in its stable region. This was confirmed via implementation of the scheduler on an experimental system.

Today's more complex systems are increasingly incorporating features such as large scale multiprocessing, multiple cores, multithreading, and heterogeneous capabilities. As shown below, the speed with which jobs execute, as well as their power utilization, may depend on both the job placement as well as the number of cores utilized. Jobs may also have significant processor affinity. There currently appears to be no general framework for efficient scheduling in this context. Formulation of such a framework, based on a generalization of the delay-cost paradigm, is a goal of this report. The advantages of this approach include transparency, simplicity and guaranteed stability. From the viewpoint of priority based scheduling, the approach permits the computation of time-varying priorities which yield tradeoffs which incorporate such factors as time-sensitivity, energy utilization, and the benefits of various possible thread allocations on multicore, multithreaded machines. That is, it can yield an objective function for a scheduler.

The first part of this report concentrates on issues associated with allocating a requested number of logical processors, as in hypervisor scheduling of "guest" operating systems (OSs) in a multicore, multithreaded machine. The OSs or jobs scheduled might require single or multiple logical processors (each associated with a hardware thread), and the system have possibly large numbers of cores, hardware threads, and guest OSs. .

Some other examples of current schedulers are those in the Linux operating system [3], and the Xen [4] and VMWare [5] hypervisors. Linux, for example, maintains two job queues, one active and the other inactive, of single-threaded jobs. The active queue is ordered by priority, with each job allocated a time slice. Once all jobs on the active queue have received their time slice (which may be interrupted by such events as I/O), the two queues are switched. In contrast to an OS scheduler, a hypervisor scheduler's function is to allocate some number of logical processors to each OS "guest". A current Xen hypervisor scheduler (for guest OSs) assigns a "fair share" quantity to each guest running on each logical processor. Guests are assigned time slices of length T. Those which have received less than their fair share are given priority over those that have received more. The VMWare hypervisor also employs periods or slices of length T, and time slices are given to guest OSs according to a share allocation. Both hypervisors enforce what can be viewed as a fairness policy. Although these schedulers can in some instances provide good performance, they do not explicitly address some issues relevant to our current context. For example, they do not handle gang scheduling of software threads, their placement on processors, or the resulting energy utilization.

Some definitions are in order. We consider a system with N processors P(i), each of which consists of what might be termed *subprocessors* or *hardware threads* P(i,j) which are considered *logical processors*. We use the term *hardware thread* to indicate a logical processor, on which a *software thread*, or simply *thread*, may be dispatched. For simplicity, we assume that all hardware threads are equal. We use the term *job* interchangeably for either a guest OS. There are M(t) jobs J(i) at time t. Our scheduler only considers current jobs, whose number we denote by M without loss of generality. Each J(i) has a time varying quantity C(i,t), the *delay-cos*t, which may be

viewed as the cost of denying processing to J(i) at time t, or alternatively what it might be willing to pay for such processing. This is for a processor of nominal speed/efficiency, and C(i,t) may adjusted for example for processor speed or the effect of affinity, as discussed below. The overall benefit of scheduling a job may in general include the above quantities as well as ones associated with the cost of power, and/or multithreading effects. For example, allocating an extra thread to run on a core may change the effective speed of execution of other threads [7]. We capture the overall per-hardware thread benefit of running a job J(i) on a particular allocation of hardware threads by a quantity Q .

For convenience, we will assume that Q is constant during what we term a *scheduling period T*. A scheduling period is a time interval in which the values of Q are kept constant. Each guest operating system (OS) is scheduled to run for an integral number of scheduling periods. The system scheduler assigns guests or jobs to processors and subprocessors, and determines the order of execution. Ideally, jobs should be executed in a timely fashion, and the load should be balanced across the processors. The system should also take advantage of processor affinities, coschedule dependent jobs, and take advantage of processor or logical processor nonhomogeneity.

Our goal here is to obtain a general means for the design of a scheduler with the following behavioral features, which are obtained via a combination of the form of the function Q and the allocation and assignment methods: i) when the load is light and power cheap, guests are provided with lavish amounts of resources, ii) as power costs increase, eventually fewer cores are activated, iii) high priority jobs are generally scheduled sooner, with more resources, than low priority jobs, and iv) hardware threads allocated to an OS guest tend to be assigned in close physical proximity.

This report is organized as follows. Section 2 provides a brief discussion of the system, and generalizes the delay-cost measure discussed in (1) by incorporating affinity, processor speed, energy usage, and some multithreading effects. We then consider what might be the criterion for optimality in scheduling. The issue arises because unlike the case in (1), a job or guest scheduled to run on a multithreaded machine can be allocated more than one configuration of hardware resources. A main feature of our approach is to partition the scheduling problem into two parts: resource allocation (which involves quantities of resources), and resource assignment, where allocations are matched to specific cores and hardware threads. Two optimization criteria are considered: a) maximizing the sum of the Qs with no restrictions, which we term airline pricing,, and b) maximizing this sum subject to a fairness criterion. The result of the latter is a scheduler which has some properties of what might be viewed as a market for logical processors. That is, something akin to a market price is used for allocating resources, as well as balancing the load. The result of the former, that is, one which maximizes the sum of the Qs, uses what might be viewed as a market price for incremental gains to do allocation. Section 3 describes algorithms for resource allocation under the two criteria, that is, the determination of which jobs or guests should be scheduled in a given period, as well as the type or amount of processor resources assigned. Section 4 considers the assignment problem, using a buddy system (6) approach, also used for example in (7). Section 5 discusses issues

associated with scheduling as for example in an operating system, where jobs may be active during only a fraction of the scheduling period, and scheduling events occur at event instances, rather than on regular intervals. Section 6 discusses questions related to how an application of guest OS should choose the number of software threads or logical processors. Section 7 shows some examples of execution speeds and power utilization for some benchmarks running on Intel and AMD-based blade servers. The results illustrate speedups from allocating additional hardware threads, the performance implications of mixing applications on multithreaded cores, scaling effects related to cache sharing, and some power utilization results. These suggest that for such mutltithreaded or multicore systems, scheduling decisions should in part be determined by how well specific applications can take advantage of hardware resources, and the way these resources are structured. These properties are in this report encapsulated by the parameters v(i,j) and U(i,j) defined above. Section 8 summarizes the main results.

The appendix briefly considers some special cases of optimal schedules, which provide some insight into the forms of solutions. These include two cases of scheduling of single logical software threads under airline pricing, as well as an example of market pricing. Here optimal solutions are available with algorithms of O(MlogM) complexity, where M is he number of jobs, as indicated above. The first special case of airline pricing is optimal affinity scheduling of single threads on nonthreaded processors. Here airline pricing is equivalent to market pricing. The second treats nonaffinity scheduling of single threads on dual-threaded processors, and contrasts this with a market pricing solution.

2. Overall System Structure and Delay-Cost Optimization

The systems we consider may have multiple processor components, each comprising chips with multiple multithreaded cores. For purposes of illustration, we assume four-way multithreading, as for example in the Sun Microsystems Niagara. We further assume each core has a private L1 cache, and four cores share an L2, the slowest on-chip cache. The processor chip may have a multiple of such four core L2-sharing groups. We further assume that each core can be powered down to conserve energy. The systems we consider may be sufficiently large so that more than one job queue is required. Each such queue is associated with what we term a processor *pool*. The load then needs to be balanced across the pools.

Each such guest J(i) requests some number R(i) of logical processors in a time period T. The numbers we use for illustration, are R(i) equal to 1, 2 or 4. The logical processors are as defined above, namely hardware threads P(i,j). We assume a thread will run faster the fewer other threads it needs to share on the same core, and thus the same L1. This is consistent with results described in [7], which includes an analysis and simulation results for multithreading on PowerPC based machines.

We now define parameters j and k associated with an allocation of logical processors. Let k correspond to the number of requested logical processors for a guest OS, and j to the number of hardware threads per logical processor. We term the parameter j the *allocation number*, and refer to such an allocation as being *of type j*. Thus if k=4, a guest J(i) requires 4 logical processors. Since each core has four hardware threads, k=4 may be satisfied by allocating one, two, or 4 cores. We say this corresponds to j=1, 2, or 4 respectively. If 2 logical processors are required, we may allocate two, one, or half a core, for j=4, 2, or 1 respectively. We say an allocation A(k,j) is for k logical processors and with an allocation number j. As mentioned above, we assume that the delay-cost C(i,t) remains constant during a scheduling period T, defined below. We then refer to it as simply C(i). The delay-cost C(i,t) is the value for a job J(i) at time t for an assignment of nominal processing power. We call L(i,j) the number of hardware threads corresponding to an allocation of type j for job or guest J(i). That is, L(i,j) is the number of requested logical processors k times the allocation number j. We denote by v(i,j) the normalized expected speed or value of processing of J(i) on an allocation of type j.

Some results from [7] may be worth mentioning. Here cores have two hardware threads in our terminology, and a job may run alone, or with another job on the same core. Performance improvements are described for throughput on benchmarks, using two threads rather than just one. Some examples are: approximately .34 for compress, .27 for TPC-C, and .39 for 2 SPEC-int. On a per thread basis, the performance using two hardware threads is then .67, .635, and .695 respectively of that running a single thread. This is suggestive of the performance effects of thread scheduling, but does not quite fit our model, since it does not capture the interactions that might occur with two threads from the same benchmark running on separate processors.

Energy usage can be a factor to be considered in scheduling. We assume that such usage is the sum of some system constant (i.e. for memory, I/O, power supplies, etc,) plus a term proportional to the number (possibly fractional) of cores allocated. This is rather oversimplified, as for example in some systems, the number of active nodes or blades required may in general be a function of the allocation. We return to this topic briefly below in Section 5.

We denote by U(i,j) the *energy usage cost* for an allocation of type j to job J(i). We denote by Z(i,j) the *generalized delay-cost* of running a job J(i) with an allocation of type j. We then have:

$$Z(i,j)= v(i,j)C(i)-U(i,j) \quad (\mathbf{1})$$

Alternatively, we might have:

$$Z(i,j)=v(i,j)C(i)/U(i,j) \quad \mathbf{(2)}$$

That is, Z(i,j) can be viewed as the advantage accrued to the system by running J(i) with relative speed v(i,j), and with U(i,j) power. The greater the speed, the greater the advantage, adjusted by the cost of the power required. In (1) we subtract the cost of power; in (2) we consider the advantage per unit power.

Def.: We denote by Q(i,j) the normalized cost Z(i,j) per hardware thread allocated as in the definition for A(i,j).

We now consider the issue of optimization. When allocations are always for a single thread on non-threaded processors, a natural approach is to maximize the sum of the {Z(i,j)}. This corresponds to simply determining which jobs should be active at any given time. However, in the problem being investigated here, the choice also involves the amount of resources that should be allocated. Two possible approaches are:

A) Maximize the sum of the Z(i,j). We call this approach *airline pricing.*
B) Maximize the sum of the Z(i,j) subject to a fairness criterion. We call this *market pricing.*

Approach A "charges" the maximum price from each customer, with no uniformity. In approach B, hardware threads are allocated at the maximum cost for which there is sufficient "demand". That is, all jobs willing to "pay" some "market" price Q will receive their allocation. More precisely, in approach B , we require that if J(i) is allocated resources at a "price" Q(i,j) per hardware thread, that no J(r) is required to "pay" an amount Q(k,r) if there is a feasible allocation A(r,s) for J(r) with Q(i,j)<=Q(r,s)< Q(k,r). Another way to look at this is that a job or OS guest may run at more than one priority. Thus if the load is heavy, a job may be required to run at higher priority, and thus use fewer resources.

For another view of the difference between the above two pricing approaches, consider the following, which may be regarded as a matching problem in bipartite graphs (10). Suppose there are N cores, each with 2 hardware threads, and some number of jobs {J(i)}, each requiring a single logical processor. Suppose we associate with each J(i) a "shadow job" sJ(i), which represents the gain from having J(i) running alone on a core. It can be shown that an optimal airline pricing scheduler would choose the top 2N jobs among the {J(i), sJ(i)}, so that a job J(i) and its shadow would be chosen if *each* was in the top 2N. Under market pricing a job and its shadow would run if their *average* cost was in the top 2N. Basically, this means that under airline pricing a job J(i) might be required to "pay" more for resources on a per hardware thread basis than another job J(k).

Which criterion is best depends on the goals of the scheduler. Airline pricing maximizes a measure of worth to the system. Market pricing corresponds more closely to a priority-based approach, as the market price may be viewed as representing a system-wide priority at which jobs are guaranteed to run. It also more easily permits a comparison of congestion at various processors or processor pools.

3. Thread Allocation

3.1 Allocation at a clearance price

The scheduler allocates resources to each job or guest J(i) at each period T. Consider an allocation of type A(j,k). We would expect, and will assume that v(i,j)> v(i,k) if j>k. That is, the more hardware threads allocated per logical processor, the faster the speedup. This is because of such factors as decreased sharing of the L1 caches and execution units in the cores. However, it may be expected that the speedup is less

than proportional to the number of hardware threads allocated. That is, Z(i,j) is expected to increase with j, and Q(i,j) to decrease with j.

We now more formally describe the allocation criterion, as well as an algorithm to implement it.

Def.: We say that *the demand from job J(i) for threads at a price Q* is the maximum number of hardware threads corresponding to a value Q(i,j) >=Q.

Def.: The *clearing price* Q* is the highest value of Q at which the total demand over all jobs for threads is equal to or greater than the amount available.

That is, as the price Q is lowered, the demand increases. Eventually, the demand is high enough to include all hardware threads.

The following procedure is a framework for the allocation process.

Procedure A:

A) For each J(i), determine the highest value of Q at which this job's demand corresponds to an allocation of type j.

B) Determine the clearing price Q* by obtaining the total demand at each Q.

C) Choose an allocation of not more than 4N hardware threads with the greatest Qs.

Note that, from the above definition, either the clearing price corresponds exactly to the number of available hardware threads, or it corresponds to more than the number of threads.

A straightforward method to implement the above procedure is as follows.

Algorithm 1:

A) For each J(i), obtain the values L(i,j) and Q(i,j) for j=1,2,3,4.

B) Enter the values for Q(i,j) and L(i,j) in a row v(i), ordered by decreasing value of Q, of a matrix V.

C) Starting with the first column of V, and for every column until the demand exceeds 4N, for each value of Q(i,j) in this column, determine the largest L(p,q) in each row which corresponds to a Q(p,q) not smaller than Q(i,j). The sum of the obtained L(q,p) obtained is then the demand at price Q(i,j). That largest Q(i,j) which yields a demand not smaller than 4N is the market price.

D) If the total allocation is greater than 4N, reduce the demand by not scheduling a job or jobs, or giving some jobs a smaller allocation. This can be done by reducing allocations to jobs with the lowest values of Q.

In the above algorithm, C is the step with maximum complexity. Its complexity is of O(M**2), as each entry is paired with a comparison of the items from each row. The number of columns is the number of hardware threads per processor. An alternative algorithm is of complexity O(MlogM):

Algorithm 1a:

A1) Step A as above.

B1) Construct the matrix V as above.

C1) Sort the Qs in the matrix V (complexity O(MLogM)).

C2) Via a binary search on the sorted Qs, find the largest Q corresponding to a demand of at least 4N. The latter operation requires O(M) steps to obtain the demand from each job at a cost Q, and there are O(log M) such steps.

D) Step D as above.

The following example illustrates the procedure. We assume N=3, with each core having 4 threads. We assume 4 jobs or guests, J(1), J(2), J(3) and J(4). The number of threads, and the Q(i j) for j=1,2,3,4 are shown in Table 1 below.

Table 1. Cost versus total demand for the example.

Jobs	L(i,1)	Q(i,1)	L(i,2)	Q(i,2)	L(i,4)	Q(i,4)
J(1)	4	10	8	7.5	16	5
J(2)	4	8	8	6	16	4
J(3)	2	6	4	4.5	8	3
J(4)	2	4	4	3	8	2
Q=	2	4	6	7.5		
D=	48	38	18	12		

Table 1 also shows the demand for various values of Q. An allocation of 2 cores to J1 and 1 core to J2 is the result.

Here we have that a price of 7.5 corresponds exactly to a demand of 12 hardware threads, which is the number available. This would correspond to 2 cores (4 threads apiece) granted to J(1) and one core to J(2).

3.2 Allocation with Airline Pricing

In order to do allocation under this criterion, we consider, rather than the per hardware thread value of Q for a given job, the marginal advantage of allocating additional hardware threads to a given job. We then perform a steepest descent optimization.

Def.: The marginal per hardware thread advantage of allocating additional hardware threads to a given job J(i) is defined as:

$$F(i,m,n)=(L(i,m)Q(i,m)-L(i,n)Q(i,n))/(m-n) \quad (2)$$

Def.: We say that *the demand from job J(i) for threads at a marginal price F* is the maximum number of hardware threads corresponding to a value F<=F(i,m,n).

We can find an allocation via the following algorithm, which at each point adds additional hardware threads to that job with the highest marginal advantage, until all hardware threads are allocated. This can be done via a modified version of Algorithm 1, under the assumption that the demand is monotonically increasing with decreasing F. We define the *clearing marginal price* F* as the least value of F at which the total demand for threads is equal to or greater than the amount available.

We note that if the magnitude of the slope of Q(i,m) is as might be expected monotonically decreasing with m, then F(i,m,n) is decreasing with increasing m for fixed (m-n) and decreasing with increasing (m-n) with fixed n. This means that the total demand will increase monotonically with decreasing F.

Algorithm 2:

A) For each J(i), obtain the values L(i,m) and F(i,m,n) for m,n=1,2,3,4 and m>n. (this in an O(M) complexity operation)

B) Enter the quantities F(i,m,n) into a sorted list**.** This is of complexity O(MlogM).

C) Via a binary search on the sorted Fs, find the largest F corresponding to a demand of at least 4N. The latter operation requires O(M) steps to obtain the demand from each job at a cost F, and there are O(logM) such steps.

D) If the total allocation is greater than 4N, reduce the demand by not scheduling some jobs or giving them a smaller allocation (complexity O(n)).

Step D is not optimal, as the constraints on the discrete number of threads that can be allocated to a job means that a solution which includes jobs with a lower value of F may better. Table 2 shows the demand for various values of F for the above example.

Table 2. Marginal Cost versus total demand for the example.

Jobs	L(1,1)	F(1,4,0)	L(1,2)	F(1,8,4)	L(1,4)	F(1,16,8)
J(1)	4	10	8	5	16	2.5
J(2)	4	8	8	4	16	2
J(3)	2	6	4	3	8	1.5
J(4)	2	4	4	2	8	1
F=	4	5	6			
D=	48	14	10			

We note that F*=5 in this example. Here J(1) is allocated 8 hardware threads, J(2) is allocated 4 and J(3) allocated 2. This adds to a value of 104, and 14 hardware threads, which is more than the 12 available. To obtain an allocation, step D above could allocate 8 to J(1) and 4 to J(2), and none to J(3). This yields total value of 92, the same result as in the market price approach above.

4. Pool and Thread Assignment

We now briefly consider the issues of load balancing and of assigning physical resources corresponding to the above allocations.

The market price obtained in a period T for hardware threads in a pool of processors might be used to assign jobs arriving for the next periode. Unlike say the

length of the queue, this price is adjusted for thread usage, affinity, and power utilization. Jobs may then be steered to that pool with the lowest price. If at some time the prices become sufficiently nonuniform across pools, jobs might be moved from from pools of high to those of low cost.

An assignment of hardware threads to a job should attempt to preserve spatial locality to the extent possible. For example if 4 hardware threads are assigned to a job, then these should be assigned on the same core. This is also the case for 2 hardware threads. More generally, if the number of identical cores on a chip is a power of two (today's processors tend to have 1, 2, or 8 such cores), and the number of chips on a compute node is also power of two, then a buddy system for hardware thread assignment, as sometimes used to allocate contiguous units of memory, and as suggested in (9) for processor allocation, appears advantageous. For example, the following outlined procedure could be employed.

Algorithm 3: Thread assignment.

A) Threads not committed are gathered into buddy clusters.
B) Sort the J(i) according to the number of hardware threads allocated.
C) In order of decreasing thread allocation, assign hardware threads to meet the allocation with the least number of breakups of members of buddy classes if the allocation can be done within a buddy class; otherwise do the assignment with the fewest separate members.

A modification to the above would be to preserve assignments between periods if the allocations remain unchanged.

5. Event-based Scheduling

Here scheduling is done on events, rather than on time periods, as jobs may be active for only part of a period.. This is typical of scheduling in an operating system, as opposed to a hypervisor, which, as mentioned above, generally does scheduling on fixed time periods. The system is as above, with each processor chip containing possibly multiple cores, each with multiple (here assumed to be 4) threads.

As before, we recompute the normalized costs Q every T seconds. The scheduler potentially dispatches a job on every event. Events include:

a) Completion of a time slice. Here the job queue is examined for possible candidates.

b) Suspension. Here a job is suspended, and the job queue is searched for a suitable job.

c) Arrival. A new job arrives, is added to the job queue, and the job queue examined.

d) Recomputation of the the delay costs.

The scheduler must decide:

I) Which, if any, job to dispatch.

ii) How many hardware threads it should use. Note that, as above, a logical processor may occupy a whole core, or only part of one.

iii) Which core or hardware thread should be assigned.

Consider a scheduling decision at time t. Let M be the number of jobs on the wait queue, and let

G be the current configuration of free hardware threads. We can do allocation as above, namely to find a clearing price for the number of free threads. However, this may not be the best alternative, as a job may arrive for example fortuitously when the number of free threads is large, and an allocation at a low price may produce congestion for later arrivals. Instead, a better alternative may be to insist on a price which is no lower than say c percentage lower than the average for the system, or for the pool, if the system has multiple job pools. Note that this differs depending on whether we use market or airline pricing. In the former case, we insist that the average normalized cost or benefit per hardware thread allocated to each job is above a certain threshold. In the latter, we insist that the average normalized cost is above a given threshold for each hardware thread allocated to each job.

Once the allocation is done, assignment can proceed as above. In general, as in Section 5, the allocation of say 4 hardware threads to a logical processor requested by J(i) may not be feasible given the current state **G**. What this means in general is that J(i) may be assigned a lower allocation of neighboring threads (e.g. within a buddy group), with the assignment to the J(i) for example in order of the normalized cost per hardware thread.

In the above discussion of hypervisor scheduling, we assumed that the number of logical processors requested is a parameter provided to the scheduler. However, as we discuss below, the application or guest OS could determine this number as a function of the cost per hardware thread required at a given time. For example, if the current cost is high, it might be advantageous to request fewer logical processors or attempt to coschedule fewer software threads. Discuss this issue in greater detail below.

6. Software Thread and Processor Frequency Optimization

We now consider some general issues for system management. These go beyond the operation of the hypervisor or OS scheduler.

In the above, it was assumed that for each application or guest OS, the number of software threads threads was fixed. In practice, this number will need to be determined, although this is not generally a task for the hypervisor, or the system OS scheduler. A further question, not treated above, is the power utilization per core, which can be modified by changing its frequency. That is, the general problem of optimizing system operation requires a solution for the number of software threads for each job or guest OS, the allocation of hardware threads to each OS, and the frequency or power utilization for each core. This appears to require an approach which partitions the general problem into subproblems, coupled with heuristics to obtain a good policy.

To illustrate the issues, we consider two cases which arise when a server is runs a single application.

The first, Example A has one application or OS running, and the power utilization for each active core is fixed (i.e. the core frequency of each active core is fixed). We might then determine the number of software threads and their allocation (in terms of hardware threads) such as to obtain an optimal tradeoff between performance and power utilization. Let v(i,j,k) be the effective speedup for job J(i) when given an allocation j and frequency indexed by k. Let U(i,j,k) be the power utilization for this allocation. We might then maximize Z(i,j,k)=v(i,j,k)C(i)-U(i,j,k) or Y(I,j,k)=v(,j,k)C(i)/U(I,j,k). Y(I,j,k) is essentially proportional to throughput per unit power. Alternatively, we might choose the largest value of v(i,j,k) subject to a bound on U(i,j,k), for example if there is a requied power bound.

Now suppose that the processor frequencies are fixed, and that the system is running with a per core market price requirent of W. A given application or OS guest might then request the number of logical processors which maximizes Z(i,j,k) or Y(i,j,k) subject to the average per core value of at least W.

The second, Example B, again has a single application, with each each core utilizing a variable amount of energy, for example by changing the frequency. Assuming that the performance is monotonic with the clock frequency, and that the frequency is the same for each active core, a reasonable heuristic is to find the optimal allocation via Example A for each choice of frequency, then choose the optimal frequency. Note that this might not be a feasible approach for cases of more than one application running concurrently.

7. Experimental Results.

We now consider some experimental results. These were for some benchmarks running on three types of blade processors. The benchmarks were: LU, a simulated CFD, computational fluid dynamics, application, floating-point intensive. LU is one of the NAS Parallel Benchmarks, www.nas.nasa.gov. DB2, Version 9.1, with a 2gb insurance database and typical customer queries, no updates; database was contained entirely within filesystem cache, no disk I/O. Finally, an index search application, typically used for web search engines, we denote by 'Search'. The results illustrate speedups from allocating additional hardware threads, the performance implications of mixing applications on multithreaded cores, scaling effects related to cache sharing, and some power utilization results. These suggest that for such mutltithreaded or multicore systems, scheduling decisions should, as suggested above, be in part determined by how well specific applications can take advantage of hardware resources, and the way these resources are structured.

The runs were done using a Linux OS which permits restricting application software threads to selected logical processors, or alternatively having the scheduler make the selection. The numbering of the logical processors is such that the default tends to spread the load.

7.1 Multithreading on an Intel-based HS20

The HS20 machine has two processor chips or sockets. Each processor is "hyperthreaded", that is, it has two hardware threads, each of which is a logical processor. These are numbered so that (0,2) are on one chip, and (1,3) on the other. Thus if hardware threads are allocated in numerical order, the first two will be on separate processors.

Figure 1 shows performance data for the Search and DB2 running alone on the machine. Each application was restricted to run two software threads. If these are restricted to run on the same processor, then 2 hardware threads are allocated in the parlance of this paper. If they run on separate processors, one hardware thread from each processor is allocated. Thus for the DB2 application, for example, v(i,4) is approximately 1.8 times v(i,2). For the Search application, this multiple is approximately 1.4.

Figure 2 shows the results of running these two applications concurrently. Here DB2 benefits somewhat from running on a separate processor, while Search benefits from sharing. This may be because Search accesses tend to be random, and thus tend to overwrite DB2 cache contents.

It should be pointed out that this experiment is not relevant to cases of hypervisor scheduling where OS guests are not permitted to share a processor concurrently. In the case of web or database servers, guest OS's generally do not share processors; however, for desktops or desktop servers, sharing processors would be typical.

7.2 Scaling on an Intel-based HS21 XM.

This machine has two sockets, with each socket holding the equivalent of two chips each with two single-threaded processors or cores. Two processors share an L2. That is, there are 8 physical cores. Figure 3 illustrates the throughput respectively for Search and LU as the number of software threads and logical processors was increased. Throughput for Search increases almost linearly. This is perhaps because as mentioned above accesses to memory tend to be quite random for this benchmark, so that sharing an L2 is not much of a factor. In contrast, for LU the performance increases linearly until more than 4 processors are used. Here cache sharing becomes a factor. Eventually the number of misses appears to overwhelm the memory bus bandwidth, so using 8 cores provides lower performance than using only three.

Performance for LU with a single package (or socket) is shown by 'LU 1 pkg'. As mentioned above, a package contains two chips. By default two core performance involves one core from each chip, and no L2 sharing. Performance degrades with the third and fourth cores, due to sharing the L2 and memory access bandwidth. Likewise, 'LU 1 chip', shows performance degradation with two cores. Two cores produce 1445 Mop/s with a single chip, while two cores in the LU 1 pkg case provides 2226 Mop/s.

7.3 Energy Utilization on an AMD-based LS21.

This machine has 4 cores on two sockets. Each core has a private L2. Figure 4 shows the throughput for 2 or 4 Search software threads running at different clock frequencies. The throughput for 4 cores at 1400 MHz is slightly higher than that for 2 cores at 2600. However, the power utilization is substantially lower, as indicated in Figure 5. This means that the energy required to complete the task, as shown in Figure 6 is also substantially lower. Interestingly, the energy utilization using 4 cores is very similar for both frequencies. For this machine, it appears that using all hardware resources during a run is best, with the frequency adjusted to fit the power requirements.

8. Discussion and Conclusion

This report outlines a generalization of the delay-cost processor scheduling paradigm, employed in IBM iSeries systems, to systems with possibly large numbers of multithreaded machines. The emphasis is on hypervisor scheduling of guest operating systems, where scheduling is done at intervals of some length T, and the guest requests some number of logical processors. Two main differences between the schedulers considered here, and ones in current practice are a) the coscheduling of multiple logical processors associated with a guest OS, and b) that there is more than one way to satisfy a request for processing resources, with different utilization of hardware threads and electrical power. What might be viewed as a market mechanism for resource allocation provides a feasible way to treat various aspects of this problem. This was contrasted with simply maximizing the generalized delay-cost objective function.

For simplicity, the analysis assumed that all processor cores were equal, and that threads are not prioritized at the core level; that is, each assigned thread has equal priority on the processor. Dropping these assumptions is a topic for further investigation.

Inclusion of an energy cost component provides a means to control the number of cores active at any given time, given that such cores can be readily deactivated. This provides an automatic means for increasing or decreasing the number of such active cores as a function of system load and desired response times. However**,** the energy cost component included here may not always be a suitable way to address this issue. One reason for this is that the cost may be more dependent on the number of nodes or blades active at a given time. A way to address this is to do the optimization without the energy parameter U for variable numbers of N processors. The lowest number N which yields adequate performance over some period of time would then be the one chosen.

Some experimental data were included that showed that speedup factors and energy utilization should ideally be determined for each workload, as well as for each processor configuration. Speedups can depend not only on the number of logical processors, but also their cache sharing and multithreading configuration. This suggests that factors such as v(i,j) and U(i,j), as discussed above, may indeed be relevant for decisions in scheduling and resource allocation.

References

1. Franaszek, P.A. and Nelson, R.D.: *Properties of Delay-Cost Scheduling in Time-Sharing Systems*, IBM Journal of Research and Development, Volume 39, Number 3, 1995.
2. Liu, Z., Nain, P., Towsley, D.: *Sample Path Methods in the Control of Queues.* Available as UMass CMPSCI Technical Report 95-09.
3. Aas, Josh: *Understanding the Linux 2.6.8.1 Scheduler*, Silicon Graphics Report, 17 Feb., 2005.
4. *Vmware ESX Server 2: Architecture and Performance Implications*, Vmware White Paper.

5. *Credit-Based CPU Scheduler*, Xen Source http://wikixensource.com/xenwiki/CreditScheduler.
6. Nagarajayya, Nagendra: *Improving Application Efficiency Through Chip Multi-Threading*, Sun Developer Network, March 10, 2005, updated October 2005.
7. Eskesen, F.N., Hack, M., Kimbrel, T., Squillante, M.S., Eickemeyer, R.J., and Kunkel, S.R.: *Performance Analysis of Simultaneous Multithreading in a PowerPC-based Processor*, IBM Research Report RC22454, May 20, 2002.
8. Knuth, D.E.: *The Art of Computer Programming*, Vol. 1, Addison-Wesley, 1969.
9. Ousterhout, J.: *Scheduling techniques for concurrent systems*. In: Proc. of Distributed Computing Systems Conference, pp. 22-30, 1982.
10. Thomas H. Cormen, Charles E. Leiserson, Donald Rivest, and Clifford Stein, *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001, Section 26.3: Maximum bipartite matching, pp.664–669.

Appendix.

We here describe two special cases of optimal schedules for single threaded jobs, respectively on single-threaded processors with affinity, and dual-threaded processors without affinity. The techniques we develop may be amenable to generalization or the design of heuristics. As above, we wish to maximize the sum of the Q(i,j), for jobs {J(i)} running on processors Pr(j). We assume N processors, and M jobs to be scheduled in some time period T.

We first consider the problem of optimal affinity assignment. In combinatorics, this is a maximum bipartite matching problem, with approximately O(N**3) complexity (8.). This level of complexity is not practical in the current setting. However, an O(nlogn) algorithm can be obtained under some simple assumptions:

a) That the processors form non-overlapping affinity classes. and that

b) The benefit of running on an affinity processor (i.e. the incremental value of Q) is a proportional to its delay-cost C(i,t), We call the value of Q on an affinity processor the *affinty value*.

An example of property a) is that running on a core with the same L2 as used recently may avoid some cold-start penalty. Then all cores sharing this L2 would be in the same affinty class. Property b) might be assumed as some system average.

We first state the following proposition.

Proposition 1.

Consider two jobs with affinity to a given processor. That job with the highest normalized cost should be assigned to run.

Proof:

This is immediate from the uniform speedup property. Any gain due to assigning the lower cost job the affinity processor will be less than the loss to the higher cost job due to property b) above.

We now formulate an optimal assignment under the above assumptions.

Algorithm 3.

Step 1: For each affinity class, find the at most N(a) jobs with the highest affinity value Q(i).

Step2: Sort the jobs according to their values Q(k), with all but the jobs from step 1 assigned non-affinity values.

Step3: Assign the top N jobs to processors, with affinity assignments as appropriate (i.e. Those jobs from step 1 in the final schedule given to their affinity processors).

.

Proposition 2.

The assignment of Algorithm 2 is optimal for the case of uniform affinity speedup.

Proof:

Immediate from the property that the jobs assigned have the highest feasible affinity or non-affinity costs.

We now consider a special case where there is no affinity, but where each processor can run two simultaneous threads. Here we might expect that two threads might for example result in say 20% more throughput, so that a job running by itself would be (1/.6), or 4/3, faster. That is, there is a uniform speedup to an individual job from running it alone. Degradation from multiple threads might include factors such as sharing functional units, or contention for cache space.

The approach we take here is to create what we term shadow threads, scheduled somewhat like ordinary threads, which block multitheading when appropriate. We associate with each job J(i) a shadow job J'(i), with an normalized cost Q'(i) which is the difference between the cost for J(i) of running by itself, or with another job. We can then apply a procedure similar to algorithm 2.

Algorithm 4.

Step 1.

Sort the Q(i) along with the Q'(i).

Step 2.

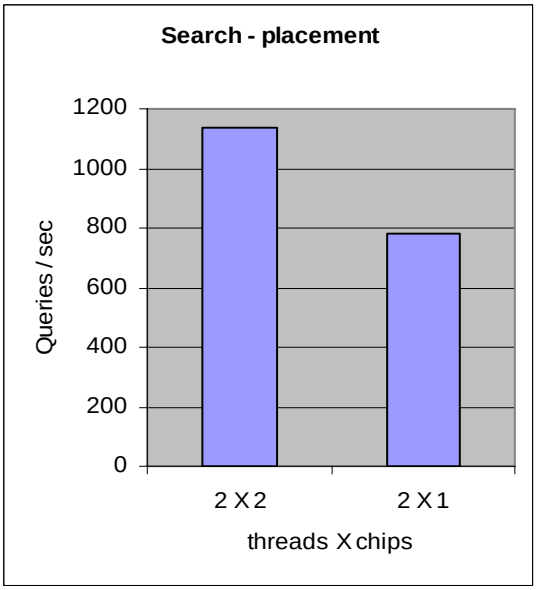
Choose those N jobs or shadow jobs with the highest values. If a job and its shadow are both chosen, schedule these on the same processor.

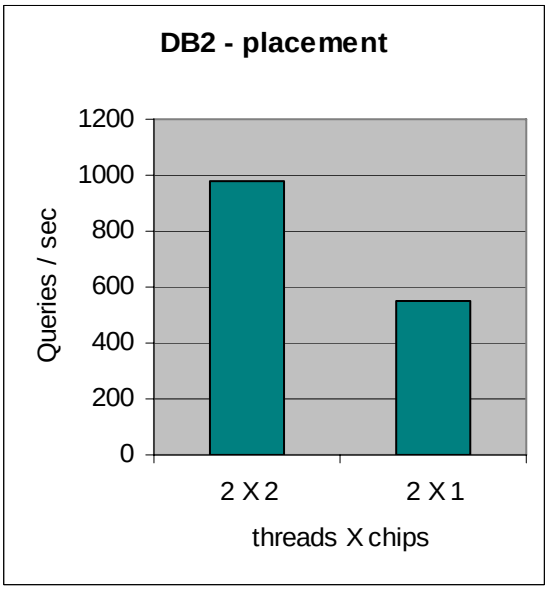
Algorithm 3 obviously chooses the highest cost jobs or shadow jobs to run, and thus minimizes the delay cost.

Suppose there are fewer than N jobs to be scheduled. This means that that some processors or subprocessors will idle. It is then clearly desirable to spread the jobs among the processors, so that none are idle. Algorithm 3 does this, with the advantage that the highest cost jobs are assigned to run single threaded. Algorithm 3 might also be utilized in a system without multithreading, but where two processors share some asset such as an L2 cache.

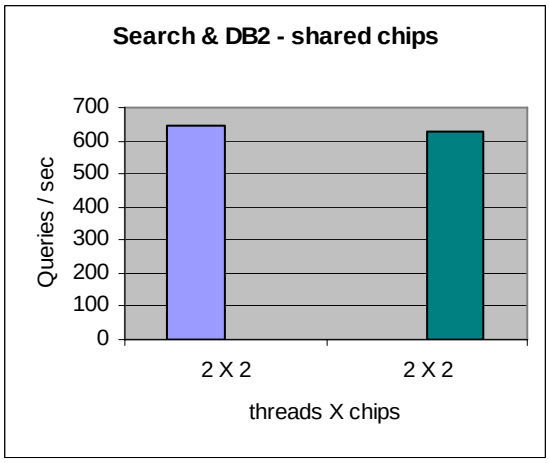
Both Algorithm 2 and Algorithm 3 provide complete solutions to their respective problems, namely both allocation of resources and assignment of those resources. More complex problems, such as gang scheduling, or the scheduling of jobs on cores with more than two hardware threads, appears difficult. These difficulties appear to be more amenable to handling via the approach taken in the main body of this report, where the problem is partitioned into a) allocation and b) assignment.

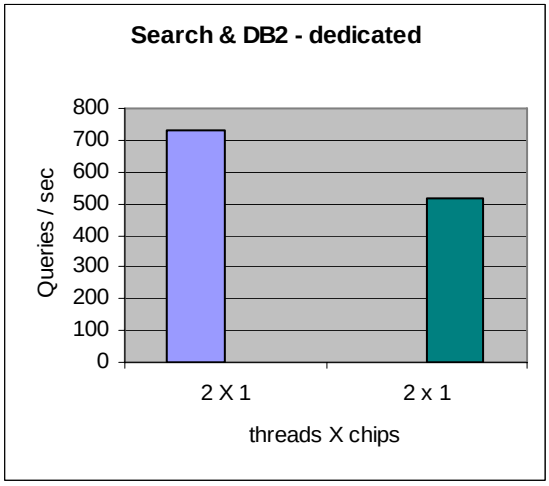
We now briefly contrast the results of Algorithm 3 above with an allocation based on market pricing. Here each job must meet a market price for the hardware threads it consumes. A job whose *average* normalized cost for two threads is greater than the market cost will be allocated the threads. Thus shadow threads may be assigned even if their cost Q(i)' his lower than that for some Q(j). In airline pricing, both the job and its shadow must each have a sufficiently high cost, as in Algorithm 4. The result in the latter case is that the price "paid" per thread is higher for some jobs than others.





---------- *Figure 1* ----------





---------- *Figure 2* ----------

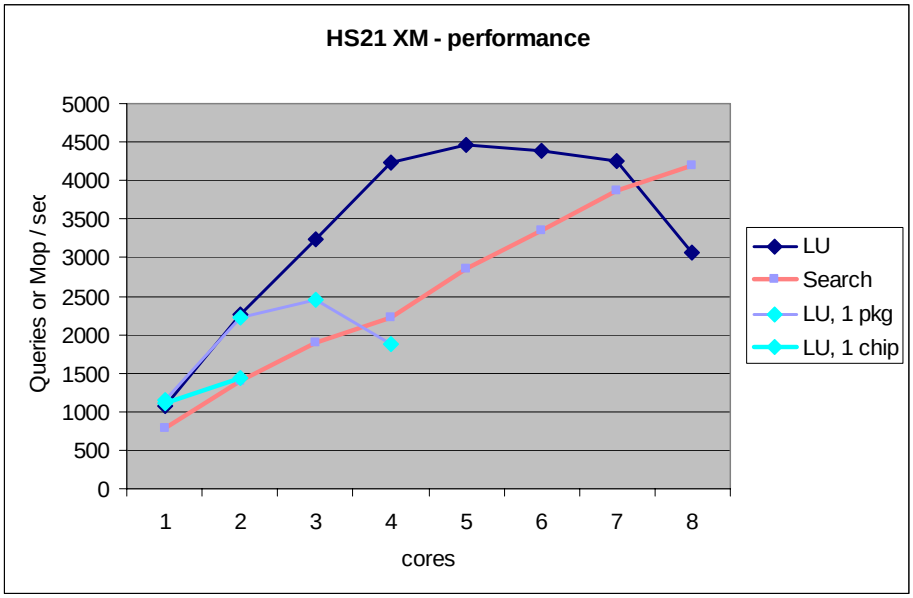
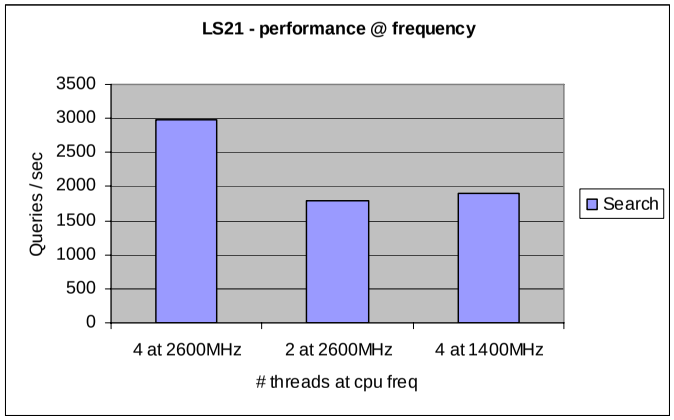
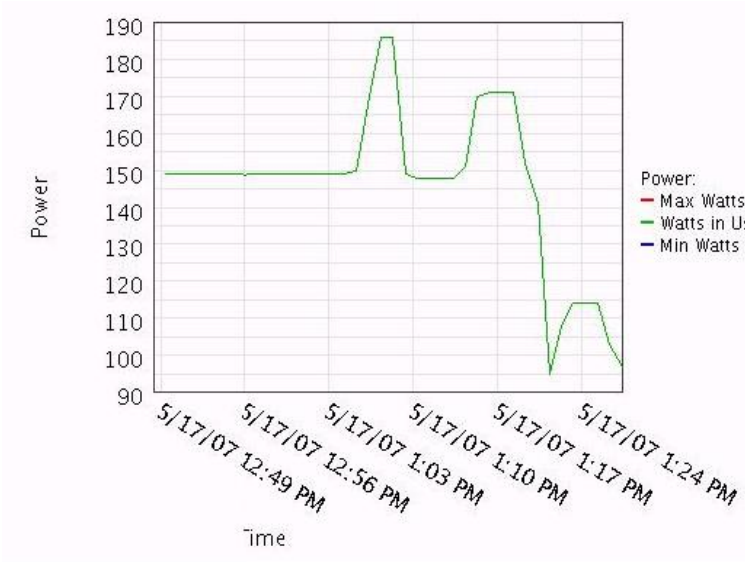
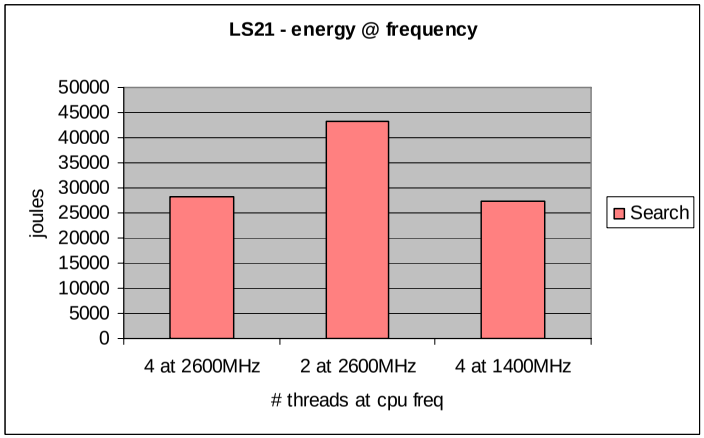

HS21 XM - performance
Queries or Mop / sec
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
1
2
3
4
5
6
7
8
cores
LU
Search
LU, 1 pkg
LU, 1 chip

Figure 3


LS21 - performance @ frequency
3500
3000
2500
2000
1500
1000
500
0
Queries / sec
4 at 2600MHz
2 at 2600MHz
4 at 1400MHz
threads at cpu freq
Search


190
180
170
160
150
140
130
120
110
100
90
Power
5/17/07 12:49 PM
5/17/07 12:56 PM
5/17/07 1:03 PM
5/17/07 1:10 PM
5/17/07 1:17 PM
5/17/07 1:24 PM
Time
Power:
Max Watts
Watts in U
Min Watts


LS21 - energy @ frequency
50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
joules
4 at 2600MHz
2 at 2600MHz
4 at 1400MHz
threads at cpu freq
Search