Hierarchical Scheduling for Symmetric Multiprocessors

Abhishek Chandra, *Member*, *IEEE*, and Prashant Shenoy, *Senior Member*, *IEEE*

Abstract—Hierarchical scheduling has been proposed as a scheduling technique to achieve aggregate resource partitioning among related groups of threads and applications in uniprocessor and packet scheduling environments. Existing hierarchical schedulers are not easily extensible to multiprocessor environments because 1) they do not incorporate the inherent parallelism of a multiprocessor system while resource partitioning and 2) they can result in unbounded unfairness or starvation if applied to a multiprocessor system in a naive manner. In this paper, we present *hierarchical multiprocessor scheduling (H-SMP)*, a novel hierarchical CPU scheduling algorithm designed for a symmetric multiprocessor (SMP) platform. The novelty of this algorithm lies in its combination of space and time multiplexing to achieve the desired bandwidth partition among the nodes of the hierarchical scheduling tree. This algorithm is also characterized by its ability to incorporate existing proportional-share algorithms as auxiliary schedulers to achieve efficient hierarchical CPU partitioning. In addition, we present a *generalized weight feasibility constraint* that specifies the limit on the achievable CPU bandwidth partitioning in a multiprocessor hierarchical framework and propose a *hierarchical weight readjustment* algorithm designed to transparently satisfy this feasibility constraint. We evaluate the properties of H-SMP using *hierarchical surplus fair scheduling (H-SFS)*, an instantiation of H-SMP that employs surplus fair scheduling (SFS) as an auxiliary algorithm. This evaluation is carried out through a simulation study that shows that H-SFS provides better fairness properties in multiprocessor environments as compared to existing algorithms and their naive extensions.

Index Terms—Multiprocessor, hierarchical, scheduling, proportional share.

1 Introduction

1.1 Motivation

Recent advances in computing have seen the emergence of a wide diversity of computing environments, including servers (for example, Web servers and multimedia servers), versatile desktop environments (running compilers, browsers, and multiplayer games), and parallel computing and scientific applications. These environments comprise collections of interacting threads, processes, and applications. Such applications often have aggregate performance requirements, imposing the need to provide collective resource allocation to their constituent entities. In general, the notion of collective resource allocation arises in several contexts:

- *Resource sharing.* Applications such as Web servers and FTP servers that partition resources such as CPU and network bandwidth and disk space between different (unrelated) concurrent client connections can benefit from the consolidation of their overall resource allocation.
- *Physical resource partitioning.* Multiple applications and threads may be grouped together as part of a physically partitioned runtime environment. A common example of such an environment is a virtual-machine monitor [1], [2], where each virtual machine may have certain resource requirements.
- *Aggregate performance requirements.* An application may have collective performance requirements from its components. For example, all threads of a parallel application should be proceeding at the same rate to minimize its "makespan" or completion time.
- *Scheduling criteria.* Many applications can be grouped together into *service classes* to be scheduled by a common scheduler specific to their requirements. For instance, different multimedia applications such as audio and video servers may require soft real-time (SRT) guarantees and hence may be scheduled together by a SRT scheduler instead of the default operating system scheduler. QLinux [3] is an operating system that provides class-specific schedulers for the CPU, network interface, and the disk.

Such aggregation-based resource allocation is particularly desirable in multiprocessor and multicore environments due to several reasons. First, resource partitioning can help in making large multiprocessor systems scalable by reducing excessive interprocessor communication, bus contention, and cost of synchronization. Cellular Disco [4] is an example of a virtual-machine-based system designed to achieve resource partitioning in a large multiprocessor environment. Moreover, a multiprocessor system provides inherent opportunities for parallelism, which can be exploited better by an application employing multiple

- *A. Chandra is with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455. E-mail: chandra@cs.umn.edu.*
- *P. Shenoy is with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003. E-mail: shenoy@cs.umass.edu.*

Manuscript received 20 Apr. 2006; revised 12 Feb. 2007; accepted 6 July 2007; published online 26 July 2007.
Recommended for acceptance by D. Trystram.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0103-0406.
Digital Object Identifier no. 10.1109/TPDS.2007.70755.

threads, requiring the system to provide some notion of aggregate resource allocation to the application. Support for such applications in multiprocessor environments is becoming critical as more systems move to multicore technology [5], [6].[1] Dual-core machines are already common, and uniprocessors are expected to become the exception rather than the rule, particularly in server environments.

Traditional operating system schedulers are not suitable for collective resource allocation for several reasons. First, they typically perform fine-grained scheduling at the process or thread level. Moreover, they do not distinguish threads of different applications from those of the same application. Second, traditional schedulers are designed to maximize systemwide metrics such as throughput or utilization and do not meet application-specific requirements. Therefore, the resource allocation achieved by such schedulers is largely agnostic of aggregate application requirements or the system's resource partitioning requirements. Some mechanisms such as scheduler activations [7] and resource containers [8] can be used to aggregate resources and exploit parallelism. However, these mechanisms are mainly accounting and protection mechanisms, and they require a complementary scheduling framework to exploit their properties.

Hierarchical scheduling is a scheduling framework that has been proposed to group together processes, threads, and applications to achieve aggregate resource partitioning. Hierarchical scheduling enables the allocation of resources to collections of schedulable entities and further perform fine-grained resource partitioning among the constituent entities. Such a framework meets many of the requirements for the scenarios presented above. Hierarchical scheduling algorithms have been developed for uniprocessors [9] and packet scheduling [10]. However, as we will show in this paper, these existing algorithms are not easily extensible to multiprocessor environments because 1) they do not incorporate the inherent parallelism of a multiprocessor system while resource partitioning and 2) they can result in unbounded unfairness or starvation if applied to a multiprocessor system in a naive manner. The design of a hierarchical scheduling algorithm for multiprocessor environments is the subject of this paper.

1.2 Research Contributions

This paper presents a novel hierarchical scheduling algorithm for multiprocessor environments. The design of this algorithm is motivated by the limitations of existing hierarchical algorithms in a multiprocessor environment, which are clearly identified in this paper. The design of the algorithm has led to several key contributions.

First, we have designed *hierarchical multiprocessor scheduling (H-SMP)*, a hierarchical scheduling algorithm that is designed specifically for multiprocessor environments. This algorithm is based on a novel combination of space and time multiplexing and explicitly incorporates the parallelism inherent in a multiprocessor system, unlike existing hierarchical schedulers. One of its unique features is that it incorporates an auxiliary scheduler to achieve hierarchical partitioning in a multiprocessor environment. This auxiliary scheduler can be selected from among several existing proportional-share schedulers [11], [12], [13], [14], [15], [16], [17], [18]. Thus, H-SMP is general in its construction and can incorporate suitable schedulers based on trade-offs between efficiency and performance requirements. We show that H-SMP provides bounds on the achievable CPU partitioning, independent of the choice of the auxiliary scheduler, although this choice can affect how close H-SMP is to the ideal partitioning.

Second, we have derived a *generalized weight feasibility constraint* that specifies the limit on the achievable CPU bandwidth partitioning in a multiprocessor hierarchical framework. This feasibility constraint is critical to avoid the problem of unbounded unfairness and starvation faced by existing uniprocessor schedulers in a multiprocessor environment. We have developed a *hierarchical weight readjustment* algorithm that is designed to transparently adjust the shares of hierarchical scheduling tree nodes to satisfy the feasibility constraint. This readjustment algorithm transforms any given weight assignment of the tree nodes to the "closest" feasible assignment. Moreover, this algorithm can be used in conjunction with any hierarchical scheduling algorithm.

Finally, we illustrate H-SMP through *hierarchical surplus fair scheduling (H-SFS)*, an instantiation of H-SMP that employs surplus fair scheduling (SFS) [19] as the auxiliary scheduler, and we evaluate its properties through a simulation study. The results of this study show that H-SFS provides better fairness properties in multiprocessor environments as compared to existing hierarchical algorithms and their naive extensions.

2 Background and System Model

2.1 Background

Hierarchical scheduling is a scheduling framework that enables the grouping together of threads, processes, and applications into service classes [3], [9], [10]. CPU bandwidth is then allocated to these classes based on the collective requirement of their constituent entities.

In a hierarchical scheduling framework, the total system CPU bandwidth is divided proportionately among various service classes. *Proportional-share scheduling* algorithms [11], [12], [13], [14], [15], [16], [17], [18], [20] are a class of scheduling algorithms that meet this criterion. Another requirement for hierarchical scheduling is that the scheduler should be insensitive to fluctuating CPU bandwidth available to it. This is because the CPU bandwidth available to a service class depends on the demand of the other service classes in the system, which may vary dynamically. A proportional-share scheduling algorithm such as start-time fair queuing (SFQ) [14] has been shown to meet all these requirements in uniprocessor environments and has been deployed in a hierarchical scheduling environment [9]. However, SFQ can result in unbounded

1. In the rest of this paper, we will refer to both multicore and multiprocessor machines as multiprocessors. The issues raised and the solutions presented are applicable to both environments.

unfairness and starvation when employed in multiprocessor environments, as illustrated in [19].

This unbounded unfairness occurs, because it is not possible to partition the CPU bandwidth arbitrarily in a multiprocessor environment, since a thread can utilize at most one CPU at any given time. This requirement is formalized as a *weight feasibility constraint* [19] on the amount of achievable bandwidth partitioning in a multiprocessor environment. SFS [19] and Group Ratio Round-Robin [16] achieve proportional-share scheduling in multiprocessor environments by employing weight readjustment algorithms to explicitly satisfy the weight feasibility constraint. However, as we will show in Section 3.2, this weight feasibility constraint is not sufficient for a hierarchical scheduling framework, and hence, these algorithms, by themselves, are inadequate for direct use in an H-SMP environment.

Aside from the choice of a suitable scheduling algorithm to be employed within the hierarchical framework, existing hierarchical schedulers are also limited in that they are designed to handle only a single resource (such as a uniprocessor). As we will illustrate in Section 3.1, these schedulers are unable to exploit the inherent parallelism in a multiprocessor environment and cannot be extended easily to run multiple threads or processes belonging to a service class in parallel.

Lottery scheduling [21] also proposes hierarchical allocation of resources based on the notion of tickets and lotteries. Lottery scheduling itself is a randomized algorithm that can meet resource requirements in a probabilistic manner, and extending it to multiprocessor environments is nontrivial. Tickets, by themselves, can be used as an accounting mechanism in a hierarchical framework and are orthogonal to our discussion here.

2.2 System Model

Our system model consists of a p-CPU symmetric multiprocessor (SMP) system with n runnable threads in the system. The threads are arranged in a hierarchical scheduling framework consisting of a *scheduling hierarchy* (or *scheduling tree*) of height h. Each node in the scheduling tree corresponds to a thread[2] or an aggregation of threads such as an application or a service class. In particular, the leaf nodes of the tree correspond to threads, whereas each internal (nonleaf) node in the hierarchy corresponds to either a service class or a multithreaded application.[3] The root of the tree represents the aggregation of all threads in the system. Fig. 1 illustrates an example scheduling hierarchy.

The goal of hierarchical scheduling is to provide CPU allocation to each node in the tree according to its requirement. Every node in the tree is assigned a weight and receives a fraction of the CPU service allocated to its parent node. The fraction that it receives is determined by its weight relative to its siblings. Thus, if P is an internal node in the tree and C_P is the set of its children nodes, then the CPU service A_i received by a node $i \in C_P$ is given by

$$A_i = \frac{w_i}{\sum_{j \in C_p} w_j} \cdot A_P, \tag{1}$$

where A_P is the CPU service available to the parent node P, and w_j denotes the weight of a node j. For instance, based on the node weights shown in Fig. 1, (1) specifies that the BE and SRT classes should receive 25 percent and 75 percent of the system CPU service, respectively. The FTP and Web servers should then share the CPU service allocated to the BE class in the ratio 1:4, thus receiving 5 percent and 20 percent of the system CPU service, respectively.

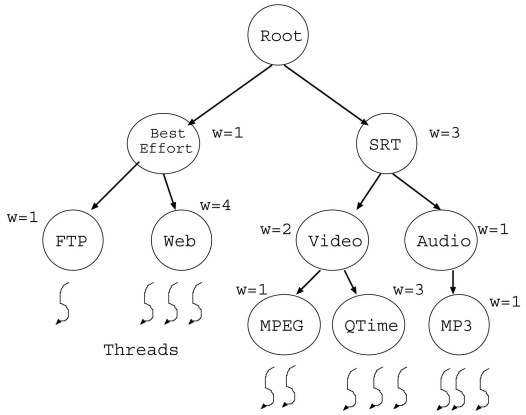

Fig. 1. A scheduling hierarchy with two service classes: best effort (BE) and SRT. The BE class consists of an FTP and a multithreaded Web server. The SRT class is further subdivided into two classes—video and audio—that consist of multithreaded applications: MPEG, Quicktime, and MP3.

For each node in the tree, we define two quantities—its *thread parallelism* and its *processor assignment*—to account for the node's location in the scheduling tree and its resource allocation, respectively. These quantities correspond respectively to the total number of threads in a node's subtree and the number of CPUs assigned to the node for multiplexing among threads in its subtree.

Definition 1: Thread parallelism θ_i. *The thread parallelism of a node i in a scheduling tree is defined to be the number of independent schedulable entities (threads) in node i's subtree, that is, threads that node i could potentially schedule in parallel on different CPUs.*

The thread parallelism of a node is given by the following relation:

$$\theta_i = \sum_{j \in C_i} \theta_j, \tag{2}$$

where C_i is the set of node i's children nodes. This equation states that the number of threads schedulable by a node is the sum of the threads schedulable by its children nodes. By this definition, $\theta_i = 1$ if node i corresponds to a thread in the system.

2. In the rest of this paper, we will refer to the smallest independently schedulable entity in the system as a *thread*. In general, this could correspond to a kernel thread, process, scheduler activation [7], etc.

3. In general, the leaf node of a tree could also correspond to a class-specific scheduler that schedules threads on the processors [3], [9]. However, we consider leaf nodes to be threads here for ease of exposition.

Definition 2: Processor assignment π_i. *Processor assignment for a node i in a scheduling tree is defined as the CPU bandwidth, expressed in units of the number of processors, assigned to node i for running threads in its subtree.*

The processor assignment of a node depends on its weight and the processor assignment of its parent node:

$$\pi_i = \frac{w_i}{\sum_{j \in C_P} w_j} \cdot \pi_P, \tag{3}$$

where P is the parent node of node i, and C_P is the set of node P's children nodes. This equation states that the CPU bandwidth available to a node for scheduling its threads is the weighted fraction of the CPU bandwidth available to its parent node. Since the root of the tree corresponds to an aggregation of all threads in the system, $\pi_{root} = \min(p, n)$ for the root node in a p-CPU system with n runnable threads.[4]

Although the thread parallelism of a node is determined solely based on the structure of its subtree in the scheduling hierarchy, its processor assignment is dependent on its weight assignment relative to its siblings and parent in the hierarchy. These quantities are independent of the scheduling algorithm being used and are useful for accounting purposes within the hierarchy.

3 Limitations of Existing Hierarchical Scheduling Algorithms

We begin by describing how hierarchical scheduling is performed in uniprocessor environments and show the limitations of such approaches and their naive extensions in multiprocessor environments. In particular, we present two problems: 1) that of inherent parallelism due to the presence of multiple processors and 2) that of infeasible weights due to the presence of monolithic schedulable entities or threads at the lowest levels of the scheduling hierarchy.

3.1 Problem of Parallelism

Algorithm 1: **hier_sched**().
1: $node \leftarrow root$
2: **while** $node$ is not a leaf **do**
3: $node \leftarrow$ **gen_sched**$(node)$ {**gen_sched** is an algorithm that selects a child of node for scheduling}
4: **end while**

Algorithm 1 shows a generic hierarchical scheduling algorithm that has been used for hierarchical scheduling on uniprocessors [3], [9]. This algorithm works as follows: Whenever a CPU needs to be scheduled, the hierarchical scheduler schedules a "path" from the root of the tree to a leaf node (or thread). In other words, the algorithm iteratively "schedules" a node at each level of the tree until it reaches a thread. This thread is then selected to run on the CPU. Scheduling an internal node of the tree corresponds to restricting the choice of the next scheduled thread to those in the node's subtree. Fig. 2a illustrates this method of scheduling by scheduling one node at a time along a path from the root to a thread. The approach described above

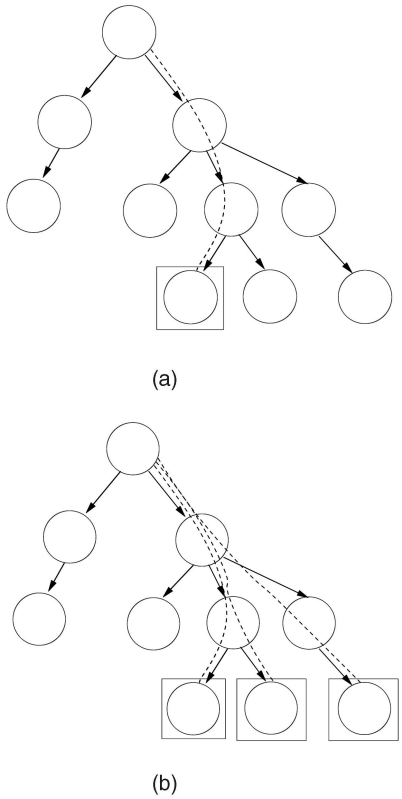

Fig. 2. Hierarchical scheduling represented as scheduling of paths from the root to threads (leaf nodes). (a) A single thread, hence a single path, is scheduled in a uniprocessor environment. (b) Multiple threads, hence multiple overlapping paths, in the scheduling tree can be concurrently scheduled in a multiprocessor environment.

requires certain properties from the algorithm gen_sched for selecting a node at each level. In uniprocessor environments, a thread-scheduling[5] proportional-share algorithm can be employed to schedule internal nodes as well. Such an algorithm runs for each set of sibling nodes (treating them as threads), scheduling one node at each level to achieve proportional allocation among them. For instance, the hierarchical SFQ algorithm [9] employs SFQ [14], a thread-scheduling proportional-share algorithm.

However, using a similar approach, namely, using a thread-scheduling algorithm to schedule the internal nodes of the scheduling tree, fails in a multiprocessor environment because of the following reason. On a uniprocessor, only a single thread can be scheduled to run on the CPU at any given time so that only a *single path* from the root needs to be scheduled at any instant (Fig. 2a). However, in a multiprocessor environment, multiple threads can be scheduled to run on multiple CPUs concurrently. This corresponds to choosing *multiple paths* from the root, many of which could have overlapping internal nodes (Fig. 2b). In other words, in multiprocessor scheduling, it is possible to have multiple threads with a common ancestor node running in parallel on different CPUs. This inherent

4. Note that min is required in this relation to account for the case where $n < p$.

5. By a thread-scheduling algorithm, we mean a scheduling algorithm that is designed to schedule individual threads or schedulable entities that do not have parallelism (unlike internal nodes in a scheduling tree that consist of multiple threads in their subtree, which can be scheduled in parallel).

parallelism can be achieved only by scheduling an internal node multiple times concurrently, that is, by assigning multiple CPUs to the node simultaneously.

This form of scheduling cannot be performed by a thread-scheduling algorithm, because it is not designed to exploit the inherent parallelism of individual schedulable entities. The key limitation of a thread-scheduling algorithm is that it has no mechanism for assigning multiple CPUs to the *same node* concurrently. This limitation prevents it from scheduling multiple threads from the same subtree in parallel. Therefore, a thread-scheduling proportional-share algorithm cannot be used to schedule the internal nodes in a scheduling hierarchy on a multiprocessor.

From the discussion above, we see that a multiprocessor hierarchical algorithm should have the ability to assign multiple CPUs to the same node concurrently. One way of designing such an algorithm is to extend a thread-scheduling algorithm by allowing it to assign multiple CPUs to each node simultaneously. However, such an extension raises several questions such as the following:

- Which nodes should we select at a given scheduling instant?
- How many CPUs should we assign to each node concurrently?
- How do we ensure that the CPU assignments achieve fair bandwidth partitioning?

We present *H-SMP* in Section 4, which answers these questions and overcomes the problem of parallelism described here.

3.2 Problem of Infeasible Weights

Aside from the lack of support for inherent parallelism, a proportional-share algorithm can also suffer from unbounded unfairness or starvation problem in multiprocessor environments [19], as illustrated by the following example.

Example 1. Consider a server that employs the SFQ algorithm [14] to schedule threads. SFQ is a proportional-share scheduler that works as follows: SFQ maintains a counter S_i for each thread with weight w_i, which is incremented by $\frac{q}{w_i}$ every time the thread is scheduled (q is the quantum duration). At each scheduling instance, SFQ schedules the thread with the minimum S_i on a CPU. Assume that the server has two CPUs and runs two computationally bound threads that are assigned weights $w_1 = 1$ and $w_2 = 10$, respectively. In addition, let $q = 1\ ms$. Since both threads are computationally bound and SFQ is work conserving,[6] each thread gets to continuously run on a processor. After 1,000 quanta, we have $S_1 = \frac{1,000}{1} = 1,000$ and $S_2 = \frac{1,000}{10} = 100$. Assume that a third CPU-bound thread arrives at this instant, with a weight $w_3 = 1$. The counter for this thread is initialized to $S_3 = 100$ (newly arriving threads are assigned the minimum value of S_i over all runnable threads). From this point on, threads 2 and 3 get continuously scheduled until S_2 and S_3 "catch up" with S_1. Thus, although thread 1 has the same weight as thread 3, it starves for 900 quanta, leading to unfairness in the scheduling algorithm.

6. A scheduling algorithm is said to be work conserving if it never lets a processor idle, so long as there are runnable threads in the system.

Although this example uses SFQ as an illustrative algorithm, this problem is common to most proportional-share algorithms. To overcome this unfairness problem, [19] presents a *weight feasibility constraint* that must be satisfied by all threads in the system:

$$\frac{w_i}{\sum_j w_j} \leq \frac{1}{p}, \tag{4}$$

where w_i is the weight of a thread i, and p is the number of CPUs in the system. A weight assignment for a set of threads is considered infeasible if any thread violates this constraint. This constraint is based on the observation that each thread can run on at most one CPU at a time.

However, in the case of hierarchical scheduling, since a node in the scheduling tree divides its CPU bandwidth among the threads in its subtree, it is possible for a node to have multiple threads running in parallel. Thus, with multiple threads in its subtree, a node in the scheduling tree can utilize more than one CPU in parallel. For instance, a node with three threads in a four-CPU system *can utilize* 3/4 of the total CPU bandwidth and is thus not constrained by the weight feasibility constraint (4). However, the number of CPUs that a node can utilize is still constrained by the number of threads that it has in its subtree: for the above example, the node *cannot utilize more than* 3/4 of the total CPU bandwidth. In particular, the number of processors assigned to a node should not exceed the number of threads in its subtree. In other words

$$\pi_i \leq \theta_i \tag{5}$$

for any node i in the tree. Using the definition of processor assignment (3), (5) can be written as

$$\frac{w_i}{\sum_{j \in C_P} w_j} \leq \frac{\theta_i}{\pi_P}, \tag{6}$$

where P is the parent node of node i, and C_P is the set of P's children nodes.

We refer to (6) as the *generalized weight feasibility constraint*. Intuitively, this constraint specifies that a node cannot be assigned more CPU capacity than what it can utilize through its parallelism. Note that (6) reduces to the weight feasibility constraint (4) in a single-level scheduling hierarchy consisting only of threads. The generalized weight feasibility constraint is a *necessary* condition for any work-conserving algorithm to achieve hierarchical proportional-share scheduling in a multiprocessor system, as it satisfies the following property (proved in Appendix B, which can be found on the Computer Society Digital Library at www.computer.org/tpds/archives.htm):

Theorem 1. *No work-conserving scheduler can divide the CPU bandwidth among a set of nodes in proportion to their weights if any node violates the generalized weight feasibility constraint.*

In Section 5, we present a *hierarchical weight readjustment* algorithm designed to transparently satisfy this constraint.

In the next section, we assume that the weights assigned to the nodes in our scheduling tree satisfy this constraint.

4 Hierarchical Multiprocessor Scheduling

In this section, we present *H-SMP*, a scheduling algorithm designed to achieve hierarchical scheduling in a multiprocessor environment.

4.1 Hierarchical Multiprocessor Scheduling

H-SMP uses a combination of space and time multiplexing to achieve the desired partitioning of CPU bandwidth within the scheduling tree. H-SMP has the following salient features. First, it is designed to assign multiple CPUs to a tree node concurrently so that multiple threads from a node's subtree can be run in parallel. Second, it employs an auxiliary thread-scheduling proportional-share algorithm to perform this CPU assignment in order to achieve the desired CPU service for the tree nodes.

Intuitively, H-SMP consists of two components: a *space scheduler* and an *auxiliary scheduler.* The space scheduler is a scheduler that statically partitions the CPU bandwidth in an integral number of CPUs to assign to each node in the hierarchy. The auxiliary scheduler is a thread-scheduling proportional-share algorithm (such as SFQ [14], SFS [19], etc.) that is used to partition the residual CPU bandwidth among the tree nodes proportional to their weights. These components work together as follows at each level of the scheduling tree: If the processor assignment of a node is π_i, then the node should ideally be assigned π_i CPUs at all times. However, since a node can be assigned only an integral number of CPUs at each scheduling instant, H-SMP ensures that the number of CPUs assigned to the node is within one CPU of its requirement. The space scheduler ensures this property by first assigning $\lfloor \pi_i \rfloor$ number of CPUs to the node at each scheduling instant. Thus, the residual processor requirement of the node becomes $\pi'_i = \pi_i - \lfloor \pi_i \rfloor$. Meeting this requirement for the node is equivalent to meeting the processor requirement for a virtual node with processor assignment π'_i. Since $0 \leq \pi'_i < 1$, this residual processor requirement can be achieved by employing the auxiliary scheduler that time multiplexes the remaining CPU bandwidth among the virtual nodes to satisfy their requirements π'_i. Overall, H-SMP ensures that each node is assigned either $\lfloor \pi_i \rfloor$ or $\lceil \pi_i \rceil$ number of CPUs at each scheduling instant, thus providing lower and upper bounds on the CPU service received by the node.

In practice, the H-SMP algorithm works as follows on a set of sibling nodes in the scheduling tree: For each node in the scheduling tree, the algorithm keeps track of the number of CPUs currently assigned to the node, a quantity denoted by r_i. Note that assigning a CPU to a node corresponds to scheduling a thread from its subtree on that CPU. Therefore, for any node i in the scheduling tree, $r_i = \sum_{j \in C_i} r_j$, where C_i is the set of node i's children. Then, H-SMP partitions each set of sibling nodes in the scheduling tree into the following subsets based on their current CPU assignment:

- *Deficit set.* A node is defined to be in the deficit set if the number of CPUs currently assigned to the node $r_i < \lfloor \pi_i \rfloor$. In other words, the current CPU assignment for a node in the deficit set is below the lower threshold of its requirement. The scheduler gives priority to deficit nodes, as scheduling a deficit node first allows it to reach its lower threshold of $\lfloor \pi_i \rfloor$ CPUs. Since the goal of H-SMP is to assign at least $\lfloor \pi_i \rfloor$ CPUs to each node at all times, it is not important to order these nodes in any order for scheduling, and they can be scheduled in a FIFO order by the space scheduler.
- *Auxiliary set.* This set consists of nodes for which $\lfloor \pi_i \rfloor = r_i < \lceil \pi_i \rceil$. These are the nodes that are currently assigned the lower threshold of their requirement and are scheduled if there are no deficit nodes to be scheduled. Scheduling these nodes emulates the scheduling of corresponding virtual nodes with processor assignment π'_i and are scheduled by the auxiliary scheduler.
- *Ineligible set.* This set consists of nodes for which $r_i \geq \lceil \pi_i \rceil$, that is, the ones that are currently assigned at least the upper threshold of their requirement. These nodes are considered ineligible for scheduling.

A node can move between these sets by being selected by H-SMP during a scheduling instant, by having one of its subtree threads finish their quantum, due to the arrival/departure of threads or due to changes in node weights in the tree.

Algorithm 2: **gen_smp**(tree_node $node$).

```
1:  new_node ← NULL
2:  if node.deficit_set is nonempty then
3:    new_node ← get_from(node.deficit_set)
4:    r_new_node ← r_new_node + 1
5:    if r_new_node ≥ ⌈π_new_node⌉ then
6:      move_to_ineligible_set(new_node)
7:    else if r_new_node = ⌊π_new_node⌋ then
8:      move_to_auxiliary_set(new_node)
9:    end if
10: else if node.auxiliary_set is nonempty then
11:   new_node ← auxiliary_sched(node.auxiliary_set)
      {auxiliary_sched is the auxiliary scheduling
      algorithm}
12:   r_new_node ← r_new_node + 1
13:   move_to_ineligible_set(new_node)
14: end if
15: return(new_node)
```

Algorithm 2 shows the pseudocode for the node selection algorithm gen_smp employed by H-SMP at each level of the tree when it needs to schedule a thread on a CPU. gen_smp can be thought to replace the gen_sched algorithm specified in Algorithm 1. We next illustrate a practical instantiation of H-SMP using SFS [19] as the auxiliary scheduler.

4.2 Hierarchical Surplus Fair Scheduling: An Instantiation of Hierarchical Multiprocessor Scheduling

H-SMP could theoretically employ any thread-scheduling proportional-share algorithm as its auxiliary scheduler. However, such an algorithm should be designed to work

on an SMP system and should reduce the discrepancy between the ideal CPU service and the actual CPU service received by the tree nodes as much as possible. We present an instantiation of H-SMP using SFS [19], a multiprocessor proportional-share scheduling algorithm, as an auxiliary scheduler. SFS maintains a quantity called *surplus* for each thread that measures the excess CPU service that it has received over its ideal service based on its weight. At each scheduling instant, SFS schedules threads in the increasing order of their surplus values. The intuition behind this scheduling policy is to allow threads that lag behind their ideal share to catch up while restraining threads that already exceed their ideal share. Formally, the surplus α_i for a thread i at time T is defined to be [19]

$$\alpha_i = A_i(0,T) - A_i^{ideal}(0,T), \tag{7}$$

where $A_i(0,T)$ and $A_i^{ideal}(0,T)$ are respectively the actual and the ideal CPU service for thread i by time T.

The choice of SFS as an auxiliary scheduler for H-SMP is based on the following intuition. As described in the previous section, a node with processor assignment π_i can be represented as a virtual node with processor assignment $\pi_i' = \pi_i - \lfloor \pi_i \rfloor$ under H-SMP for the purpose of satisfying its residual service requirement. Then, it can be shown that the surplus for the virtual node is the same as that for the actual node. This can be seen by rewriting (7) as

$$\begin{aligned}\alpha_i &= A_i(0,T) - \pi_i \cdot T \\ &= A_i'(0,T) + \lfloor \pi_i \rfloor \cdot T - \left(\lfloor \pi_i \rfloor + \pi_i'\right) \cdot T \\ &= A_i'(0,T) - A_i'^{ideal}(0,T) \\ &= \alpha_i',\end{aligned}$$

where the dashed variables (such as A_i') correspond to the values for the virtual node with processor assignment π_i'. These equations imply that scheduling nodes in the order of their surplus values is equivalent to scheduling the corresponding virtual nodes in the order of *their* surplus values.[7] This property means that SFS can be used with original node weights to achieve the same schedule without having to maintain a separate set of virtual nodes with residual weights and running the algorithm on them. This simplifies the implementation of H-SFS, as opposed to using a different auxiliary algorithm, which does not satisfy this property.

Note that H-SFS reduces to SFS in a single-level hierarchy (corresponding to a thread-scheduling scenario). In that case, the weight feasibility constraint (4) requires that $0 < \pi_i \leq 1$, $\forall i$, implying that all threads are either in the auxiliary set (if they are not currently running) or are ineligible (if they are currently running) at any scheduling instant. The auxiliary threads (that is, the ones in the run queue) are then scheduled in the order of their surplus values.

4.3 Properties of Hierarchical Multiprocessor Scheduling

We now present the properties of H-SMP in a system consisting of a fixed scheduling hierarchy, with no arrivals and departures of threads and no weight changes. Furthermore, we assume that the scheduling on the processors is synchronized. In other words, all p CPUs in the system are scheduled simultaneously at each scheduling quantum. For the nontrivial case, we would also assume that the number of threads $n \geq p$. In such a system, H-SMP satisfies the following properties (the proof of these properties are given in Appendix A, which can be found on the Computer Society Digital Library at www.computer.org/tpds/archives.htm):

Theorem 2. *After every scheduling instant, for any node i in the scheduling tree, H-SMP ensures that*

$$\lfloor \pi_i \rfloor \leq r_i \leq \lceil \pi_i \rceil.$$

Corollary 1. *For any time interval $[t_1, t_2)$, H-SMP ensures that the CPU service received by any node i in the scheduling tree is bounded by*

$$\lfloor \pi_i \rfloor \cdot (t_2 - t_1) \leq A_i(t_1, t_2) \leq \lceil \pi_i \rceil \cdot (t_2 - t_1).$$

From Theorem 2, we see that H-SMP ensures that the number of processors assigned to each node in the scheduling tree at every scheduling quanta lies within one processor of its requirement. This result leads to Corollary 1, which states that the CPU service received by each node in the tree is bounded by an upper threshold and a lower threshold that are dependent on its processor assignment. In Section 6, we relax the system assumptions made here and discuss the impact on the properties and performance of H-SMP.

5 Hierarchical Weight Readjustment

As described in Section 3.2, the weights assigned to the nodes in a scheduling hierarchy must satisfy the generalized weight feasibility constraint (6) to avoid unbounded unfairness or starvation. However, it is possible that some nodes in the hierarchy have infeasible weights. This is possible, because node weights are typically assigned externally based on the requirements of applications and application classes and may not satisfy the feasibility constraint. Even if the weights are chosen carefully to be feasible to begin with, the constraint may be violated because of the arrival and departure of threads or changes in weights and tree structure. We now present an algorithm that transparently adjusts the weights of the nodes in the tree so that they all satisfy the generalized weight feasibility constraint, even if the original weights violate the constraint.

5.1 Generalized Weight Readjustment

Algorithm 3: **gen_readjust**(array $[w_1 \ldots w_n]$, float π), returns $[\phi_1 \ldots \phi_n]$.

1: **if** $\left(\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi}\right)$ **then**
2: **gen_readjust**$([w_2 \ldots w_n], \pi - \theta_1)$
3: $\phi_1 \leftarrow \left(\frac{\theta_1}{\pi - \theta_1}\right) \cdot \sum_{j=2}^{n} \phi_j$
4: **else**
5: $\phi_i \leftarrow w_i$, $\forall i = 1, \ldots, n$
6: **end if**

7. In practice, SFS approximates the ideal definition of surplus and, hence, the relative ordering of nodes is also an approximation of the desired ordering.

Algorithm 3 shows the *generalized weight readjustment* algorithm that modifies the weights of a set of sibling nodes in a scheduling tree so that their modified weights satisfy (6). This algorithm determines the adjusted weight of a node based on its original weight and the number of threads that it can schedule. Intuitively, if a node demands more CPUs than the number of threads that it can schedule, the algorithm assigns it as many CPUs as what is allowed by its thread parallelism; otherwise, the algorithm assigns CPUs to the node based on its weight.

As input, the algorithm takes a list of node weights, where the nodes are sorted in the nonincreasing order of their weight-parallelism ratio $\left(\frac{w_i}{\theta_i}\right)$. The algorithm then recursively adjusts the weights of the nodes until it finds a node that satisfies (6). Ordering the nodes by their weight-parallelism ratio ensures that infeasible nodes are always placed before feasible nodes.[8] This ordering makes the algorithm efficient, as it enables the algorithm to first examine the infeasible nodes, allowing it to terminate as soon as it encounters a feasible node.

Algorithm 4: **hier_readjust**(tree_node *node*).

1: **gen_readjust**$(node.\text{weight\_list}, \pi_{node})$
2: **for all** *child* in the set of *node*'s children C_{node} **do**
3: $\pi_{child} \leftarrow \left(\frac{\phi_{child}}{\sum_{j \in C_{node}} \phi_j}\right) \cdot \pi_{node}$
4: **hier_readjust**(*child*)
5: **end for**

The generalized weight readjustment algorithm can be used to adjust the weights of all the nodes in the tree in a top-down manner using a *hierarchical weight readjustment algorithm* (Algorithm 4). Intuitively, this algorithm traverses the tree in a depth-first manner,[9] and for each node P, the algorithm 1) applies the generalized weight readjustment algorithm to the children of node P and 2) computes the processor assignment for the children of P by using (3) based on their adjusted weights ϕ_i.

5.2 Properties of Hierarchical Weight Readjustment

In this section, we first present the properties of the generalized weight readjustment algorithm. We then present its runtime complexity that allows us to determine the time complexity of the hierarchical weight readjustment algorithm. Detailed proofs and derivations of the properties and results presented in this section can be found in Appendix B, which can be found on the Computer Society Digital Library at www.computer.org/tpds/archives.htm.

First, the generalized weight readjustment algorithm is correct in that it ensures that no node demands more CPU service than what it can utilize, as stated by the following theorem:

Theorem 3. *The adjusted weights assigned by the generalized weight readjustment algorithm satisfy the generalized weight feasibility constraint.*

Aside from satisfying the generalized weight feasibility constraint, the adjusted weights assigned by the generalized weight readjustment algorithm are also "closest" to the original weights in the sense that the weights of nodes violating the generalized weight feasibility constraint are reduced by the minimum amount to make them feasible, whereas the remaining nodes retain their original weights. This property of the generalized weight readjustment algorithm is stated in the following theorem:

Theorem 4. *The adjusted weights assigned by the generalized weight readjustment algorithm satisfy the following properties:*

1. *Nodes that are assigned fewer CPUs than their thread parallelism retain their original weights.*
2. *Nodes with an original CPU demand exceeding their thread parallelism receive the maximum possible share that they can utilize.*

These properties intuitively specify that the algorithm does not change the weight of a node, unless it is required to satisfy the feasibility constraint, and then, the change is the minimum required to make the node feasible.

To examine the time complexity of the generalized weight readjustment algorithm, note that for a given set of sibling nodes in the scheduling tree, the number of infeasible nodes can never exceed the processor assignment of their parent node.[10] Since the generalized weight readjustment algorithm examines only the infeasible nodes, its time complexity is given by the following theorem:

Theorem 5. *The worst-case time complexity $T(n, \pi)$ of the generalized weight readjustment algorithm for n nodes and π processors is $O(\pi)$.*

Since the hierarchical weight readjustment algorithm employs the generalized weight readjustment algorithm to adjust the weights of sibling nodes at each level of the tree, we can extend the analysis of the generalized weight readjustment algorithm to analyze the complexity of the hierarchical weight readjustment algorithm, which is given by the following theorem:

Theorem 6. *The worst-case time complexity $T(n, h, p)$ of the hierarchical weight readjustment algorithm for a scheduling tree of height h, with n nodes running on a p-CPU system, is $O(p \cdot h)$.*

Theorem 6 states that the runtime of the hierarchical weight readjustment algorithm depends only on the height of the scheduling tree and the number of processors in the system and is independent of the number of runnable threads in the system.

6 Implementation Considerations

In this section, we discuss some of the system issues and considerations for implementing the H-SMP and hierarchical weight readjustment algorithms in a real system.

8. We prove this property of the ordering in Appendix B, which can be found on the Computer Society Digital Library at www.computer.org/tpds/archives.htm. The intuitive reason is that nodes that have higher weights or have smaller number of threads to schedule are more likely to violate the feasibility constraint (6).

9. We can also use other top-down tree traversals such as breadth first, where a parent node is always visited before its children nodes.

10. This is because if a node demands more CPU service than its thread parallelism, then its demand exceeds at least one processor (since its thread parallelism > 0), and the number of nodes demanding more than one processor cannot exceed the number of processors available to them, namely, the parent node's processor assignment π.

Height of scheduling hierarchy. Each internal node of a scheduling tree typically corresponds to either a multithreaded application or an application class. Most systems have the need for only a few statically defined application classes (such as BE, real time, etc.), and hence, scheduling trees in real implementations can be expected to be broad rather than deep. The height h of a typical scheduling tree can thus be expected to be a small constant. A special case is that of a two-level hierarchy, consisting only of independent threads and no further grouping into application classes, which can be the default configuration of the scheduling tree at system initialization. Note that the height of the tree h is not dependent on the number of threads n in the system (for example, $h \neq O(log(n))$, which is the usual assumption in many parallel models [22]). This is because a scheduling tree is not binary (or k-ary for a constant k), and there can be an arbitrary number of children attached to each node in the tree. Thus, the time complexity of the hierarchical scheduling algorithm (Theorem 6) in real implementations is likely to be truly independent of the number of threads in the system.

Arrivals and departures of threads. Although we made assumptions of a statically known tree of threads in Section 4.3 for the tractability of the proofs, H-SMP is not constrained by these assumptions for its actual functionality. Arrivals and departures of threads or weight modifications in the tree would essentially require the hierarchical weight readjustment to be performed in the tree. This is because these events change the parallelism θ_i or processor availability π_i of the nodes to which they are attached, thus possibly making some of the weights in the tree infeasible. Due to the readjustment in weights, it is possible that some of the nodes in the system may temporarily violate the bounds of $\lfloor \pi_i \rfloor$ and $\lceil \pi_i \rceil$ on their current assignment r_i (Theorem 2). One possibility of preventing this violation is to reschedule *all the CPUs* with the H-SMP algorithm immediately upon each weight readjustment. However, a less expensive approach is to ignore such temporary violations and allow subsequent scheduling events to gradually restore the above property for all nodes. In fact, in our simulations presented in Section 7, we allow arrivals and departures of threads, resulting in such temporary violations, but our results show their impact on the allocation fairness to be negligible.

Reducing tree traversals. Another approach to reduce the overhead of traversing the whole tree at every scheduling instant is by using a different granularity of scheduling/readjustment across different levels of the tree. For instance, a different scheduling quantum size Q can be associated with each level of the tree such as $Q(l) = 2^{h-l} \cdot q$, where h is the height of the tree, l is a tree level (with $l_{root} = 0$), and q is the baseline quantum size used by the system for threads. In this case, scheduling at level l will be done only after $Q(l)$ time units so that scheduling at lower levels of the tree is performed more frequently than at higher levels. For instance, although threads will be scheduled at the system quantum, node selection at upper levels would take place less frequently. Such a use of different granularities will also lead to more effective space partitioning of the CPUs. Of course, such an approach would lead to load imbalances and unfairness within the quantum durations but would lead to more efficient execution.

List management overheads. As outlined in the description of H-SMP, each internal node in the tree maintains three lists: deficit, auxiliary, and ineligible. The deficit list and the ineligible list do not require any specific ordering, and hence, insertions/deletions from these lists will be $O(1)$. The auxiliary list is typically maintained according to the priority order assigned by the auxiliary algorithm (for example, ordered by surplus for SFS). The insertions/deletions from this list could be made $O(log(n))$ by maintaining suitable data structures for the list implementation such as heaps. Similarly, the weight readjustment algorithm maintains a sorted list of nodes by their weight-parallelism ratios. Such a list will typically have to be modified much less frequently, although similar data structures could be employed here.

7 Simulation Study

We now present a simulation study to evaluate the properties of the H-SMP algorithm and compare it to other existing algorithms and their extensions. We first present our simulation methodology and metrics used in our evaluation, followed by the results of the study.

7.1 Simulation Methodology

In our study, we used an event-based simulator to simulate multiprocessor systems with different numbers of processors p. For each simulation, we generated a scheduling tree hierarchy with a given number of internal nodes N and a given number of threads n. These nodes and threads were arranged in the tree in the following manner. The parent of an internal node was chosen uniformly at random from the set of other internal nodes, whereas the parent of a thread was selected uniformly at random from the set of internal nodes without children (to prevent a thread and an internal node from being siblings in the tree). These nodes and threads were then assigned weights chosen uniformly at random from a fixed range of values (1-100).

Each run of the simulation is conducted as follows: Similar to a real operating system, the system time is measured in ticks, and the maximum scheduling quantum is defined to be 10 ticks. Each CPU is interrupted at a time chosen uniformly at random within its quantum,[11] at which point it calls the hierarchical scheduler to assign the next thread to run on the CPU. Each thread is assigned a service time at creation, which is generated from a heavy-tailed distribution $(\text{Pareto}, a = 1.01)$. A thread departs the system when it has received a CPU service equal to its desired service time. We model arrivals of new threads as a Poisson process, and thread arrival times are generated from an exponential distribution.[12] Furthermore, we used different random-number streams for each of the different distributions (parent IDs, weights, scheduling times, service times, and arrival times).

11. Threads may not use full quantum lengths either because of having used up partial quantum lengths in their previous runs or due to preemption or blocking events.

12. To emulate a stable system (with the number of arrivals approximating the number of departures), we used a mean $\lambda \propto \frac{1}{p}$.

In our study, we simulated multiprocessor systems with 2, 4, 8, 16, and 32 CPUs, respectively. We generated scheduling trees with 2, 4, 6, 8, and 10 internal nodes and a set of values for the number of threads in the system, varying from 3 to 100. Each simulation (with each of these parameter combinations) was run for 10,000 ticks and repeated 100 times with a different random seed to simulate different thread mixes and tree structures for the same parameter combinations.

In our study, we use H-SFS as an instantiation of H-SMP for evaluation purposes. We compare the H-SFS algorithm with the following algorithms that represent existing algorithms and their extensions for use in a hierarchical scheduling framework.

7.1.1 Surplus Fair Scheduling

As a representative of a uniprocessor hierarchical scheduler (Algorithm 1), we employed SFS algorithm directly to schedule nodes at each level of the scheduling hierarchy. Note that even though SFS is a proportional-share algorithm designed to schedule threads in a multiprocessor environment, it cannot exploit the inherent parallelism of internal tree nodes.

7.1.2 Extended Surplus Fair Scheduling

This algorithm is an extension of SFS for hierarchical scheduling that works as follows: At each scheduling instant, at each level of the scheduling hierarchy, it selects a node with the minimum surplus among a set of sibling nodes and assigns it $\lceil \pi_i \rceil$ number of CPUs. This is a generalization of SFS, as SFS algorithm assigns $\lceil \pi_i \rceil = 1$ CPU to each selected thread, where $0 < \pi_i \leq 1$ for all threads in that case. Thus, this algorithm is designed to extract parallelism for each internal node in the scheduling tree. However, note that this algorithm does not guarantee that $\lfloor \pi_i \rfloor \leq r_i \leq \lceil \pi_i \rceil$ for all nodes in the scheduling tree, as shown for H-SMP. We use Ext-SFS to represent one possible extension of an existing multiprocessor proportional-share algorithm.

7.1.3 Hierarchical Round Robin

Hierarchical Round Robin (H-RR) is an instantiation of H-SMP that employs the Round-Robin algorithm as the auxiliary scheduler. This algorithm also employs a space scheduler (and the notions of deficit, auxiliary, and ineligible sets) to assign processors to nodes. However, the nodes in the auxiliary set are scheduled using the Round-Robin scheduler instead of SFS or another proportional-share algorithm. We use this algorithm for comparison to illustrate that a naive choice of auxiliary scheduler can result in deviations from the fair allocation of bandwidth, even when employing the space scheduler as the first component of the hierarchical scheduler.

To quantify the performance of an algorithm, we measure the normalized deviation D_i of each node i in the scheduling tree from its ideal share: $D_i = \left| \frac{A_i - A_i^{ideal}}{A_{total}} \right|$, where A_i and A_{total} denote the CPU service received by node i and the total CPU service in the system, respectively, and A_i^{ideal} is the ideal CPU service that the node should have received based on its relative weight in the hierarchy. We then use statistics such as the mean and maximum deviation of all the nodes in the scheduling tree to quantify the unfairness of the algorithm. Thus, an algorithm with smaller deviation values is better able to satisfy the CPU requirements of the tree nodes.

7.2 Comparison of Schedulers

We first compare the performance of the various schedulers described above in terms of their ability to satisfy the CPU requirements of the threads and nodes in the scheduling tree. We present results only for some of the parameter combinations due to space constraints.

7.2.1 No Arrivals and Departures

In our first set of experiments, we assume a fixed set of threads and a fixed scheduling hierarchy during each run; that is, there are no thread arrivals/departures. Fig. 3 shows the comparison of the algorithms described above for 2, 8, and 32-processor systems for a scheduling tree with 10 internal nodes. The figure plots both the *mean* and the *maximum* deviation from the ideal share for all the nodes in the tree. Based on the graphs, the SFS algorithm has the highest deviation. This is because SFS assigns at most one CPU to each node, resulting in large deviations for nodes that have a requirement of multiple CPUs. The poor performance of SFS shows the inability of a thread-scheduling algorithm to exploit thread parallelism within the tree. The H-RR algorithm also performs relatively poorly. However, in Figs. 3b, 3d, and 3f, which plot the *maximum* deviation for any node in the tree, we see that the maximum deviation of any node in the tree, in the presence of H-RR, is bounded by about 17.1 percent, 5.89 percent, and 1.75 percent for 2-, 8-, and 32-CPU systems, respectively, which translates to a maximum deviation of about 0.34, 0.47, and 0.54 CPUs, respectively. Since the maximum deviation < 1, this result shows that the number of CPUs available to any node in the scheduling tree is bounded by the upper and lower thresholds of its processor requirement. However, since the Round-Robin algorithm does not differentiate between the requirements of different nodes, the residual bandwidth is not divided proportionately among the nodes. Finally, we see that the Ext-SFS and H-SFS algorithms have small deviation values, indicating that employing a generalization of a proportional-share algorithm is crucial in meeting the requirements. Furthermore, we see that H-SFS has the smallest deviation values, which indicates that a combination of threshold bounds and a proportional-share algorithm provides the best performance in terms of achieving proportional-share allocation.

Similarly, Figs. 4a and 4b show the comparison of the algorithms for scheduling trees with different sizes (6 and 10 internal nodes, respectively) and running on a 32-processor system. As shown in the figures, the results are similar to those obtained above, and the H-SFS algorithm again has the least mean deviation values among the algorithms considered here.

7.2.2 Arrivals and Departures

Next, we allow thread arrivals and departures in the system by using the methodology described in Section 7.1. Fig. 5 shows the mean deviation for the different algorithms for 4, 8, 16, and 32-CPU systems with 10 internal nodes, respectively. Here, we have the initial number of threads in the system on the x-axis (the actual number of threads vary over the duration of the run as threads arrive and depart). Once again,

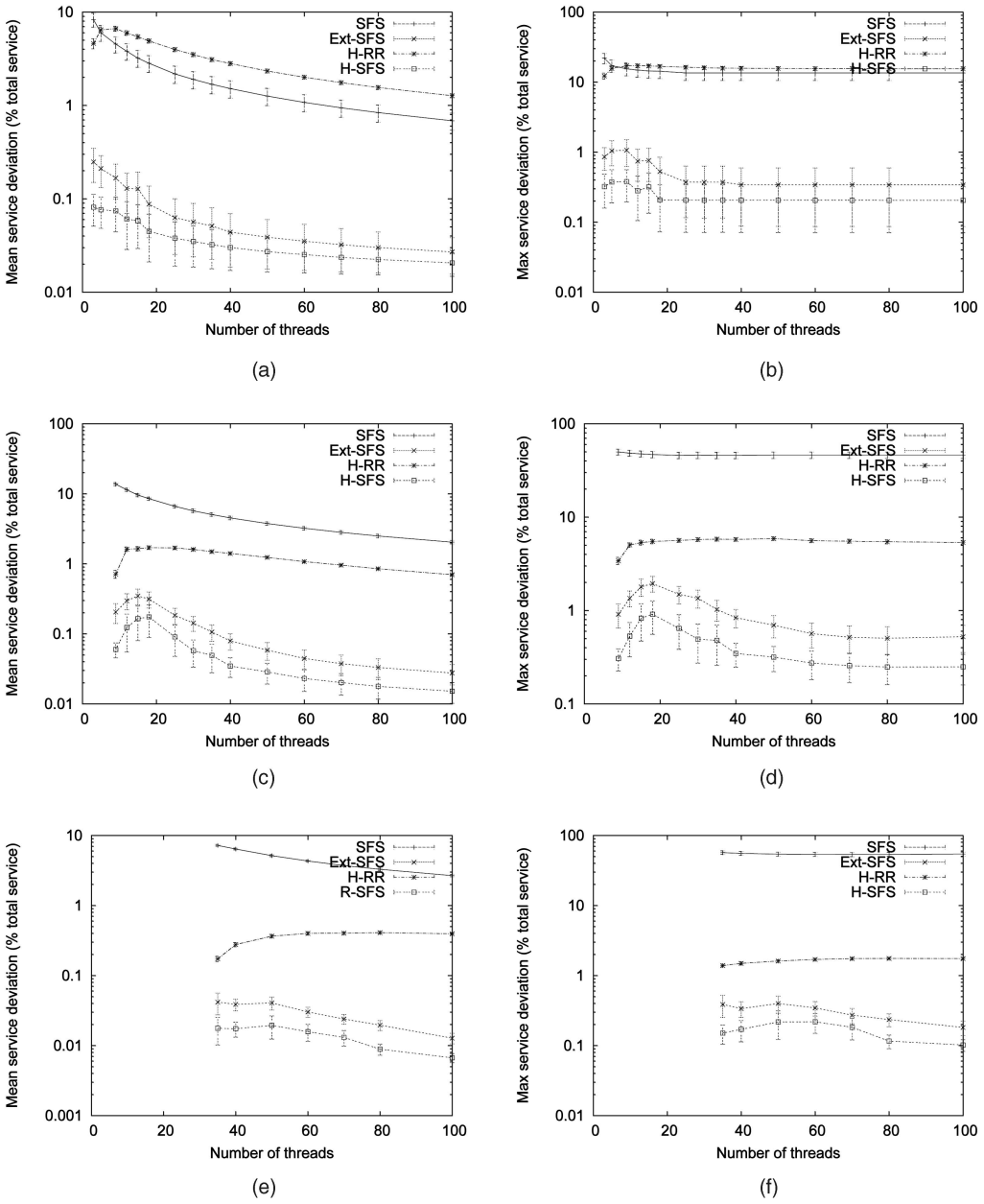

Fig. 3. Mean and maximum deviation for scheduling trees with 10 internal nodes on different sizes of multiprocessor systems and no arrivals/departures. (a) Two CPUs: mean deviation. (b) Two CPUs: maximum deviation. (c) Eight CPUs: mean deviation. (d) Eight CPUs: maximum deviation. (e) Thirty-two CPUs: mean deviation. (f) Thirty-two CPUs: maximum deviation.

we notice that SFS performs the worst and H-SFS performs the best in all cases. However, we notice an interesting trend with respect to the performance of H-RR and Ext-SFS. As the number of CPUs increases, the performance of Ext-SFS becomes comparatively worse, whereas that of H-RR improves marginally. This happens, because with an increasing number of CPUs, there are more arrivals/departures in the system.[13] This result indicates that H-SMP-based algorithms (H-SFS and H-RR) are able to maintain the processor requirement bounds more effectively in the presence of an increasing number of thread arrivals/departures. Moreover, the impact of such arrivals/departures on H-SMP can be seen to be minimal in terms of maintaining these bounds.

Overall, these results demonstrate that a combination of space scheduling coupled with a proportional-share algorithm such as SFS as an auxiliary scheduler achieves the most desirable allocation. We see that a thread-scheduling algorithm is ineffective for exploiting thread parallelism in a scheduling tree. We also see that although a simple extension of a proportional-share algorithm such as SFS is fairly effective in achieving a desirable allocation, its performance is adversely affected by frequent thread arrivals/departures. Finally, fitting a naive algorithm such as Round Robin in the hierarchical framework does not maintain the desired shares effectively.

13. There are more processors available for parallel execution, thus allowing threads to finish faster. In addition, recall that we use a higher arrival rate for more CPUs to balance out the large departure rate in our experiments.

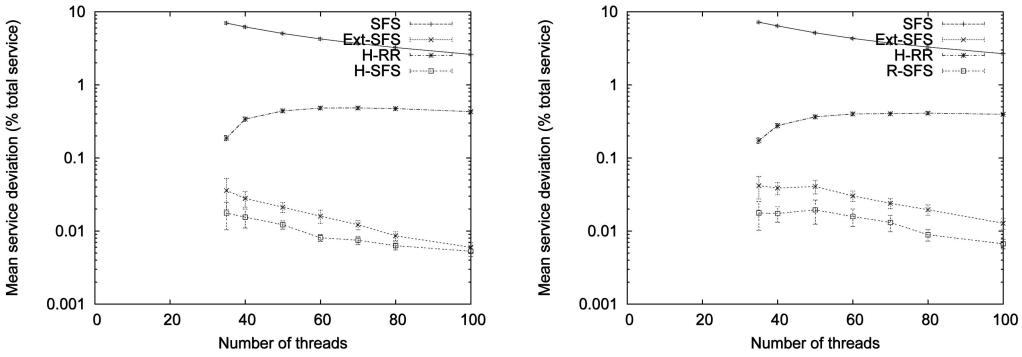

Fig. 4. Mean deviation for scheduling trees with different sizes on a 32-processor system and no arrivals/departures. (a) Six internal nodes. (b) Ten internal nodes.

7.3 Impact of System Parameters

Now, we consider the effect of system parameters such as the number of threads, number of processors, and tree size on the performance of H-SFS. All of these results are for the scenario with arrivals/departures of threads.

7.3.1 Impact of the Number of Threads

The impact of increasing the number of threads on H-SFS can be seen in Figs. 3, 4, and 5 (the lowest curve in all figures), which show that the number of threads has little impact on the allocation deviation, indicating that H-SFS is largely unaffected by the number of threads in the system.

7.3.2 Impact of the Number of Processors

In Fig. 6, we plot the mean deviation as the number of CPUs is varied. In the figure, we see that the mean deviation increases as we increase the number of processors in the system. This result can be explained due to more arrivals/departures happening with more CPUs, as explained in the previous section. Such frequent arrivals/departures impact the adjusted weights more frequently and lead to more temporary imbalances in processor allocation.

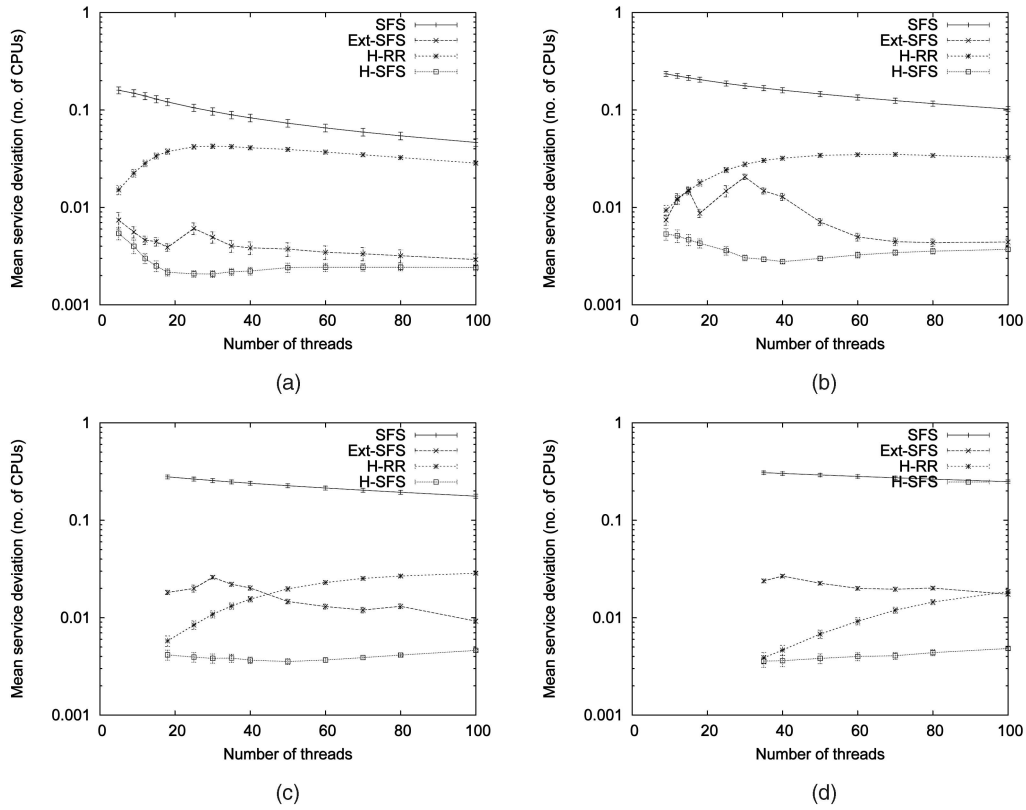

Fig. 5. Mean deviation for scheduling trees with 10 internal nodes on different sizes of multiprocessor systems and with arrivals/departures. (a) Four CPUs. (b) Eight CPUs. (c) Sixteen CPUs. (d) Thirty-two CPUs.

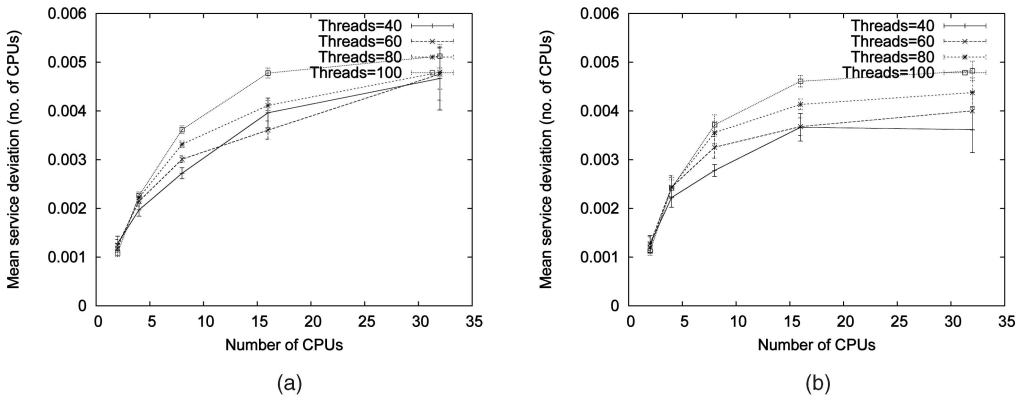

Fig. 6. Effect of the number of processors on the deviation of H-SFS. (a) Six internal nodes. (b) Ten internal nodes.

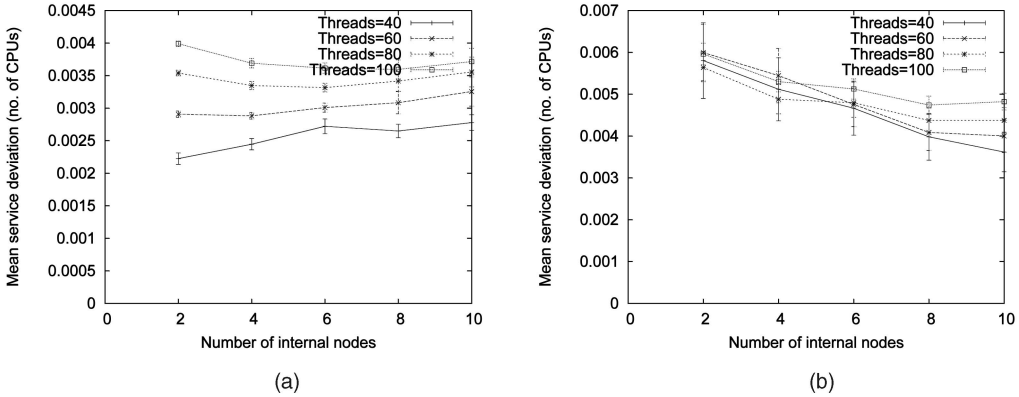

Fig. 7. Effect of tree size on the deviation of H-SFS. (a) Eight CPUs. (b) Thirty-two CPUs.

7.3.3 Impact of Scheduling Tree Size

In Fig. 7, we plot the mean deviation as the tree size (in a number of internal nodes) is varied. As shown in the figure, the number of nodes has little impact on the deviation, indicating that the tree size does not affect the performance of H-SFS.

Overall, our results demonstrate that H-SFS is effective in providing the desired shares of nodes in a scheduling hierarchy. Moreover, the performance of H-SFS is largely unaffected by the tree size and the number of threads in the system, and frequent arrivals/departures have only a small impact on its fairness properties.

8 Concluding Remarks

In this paper, we considered the problem of using hierarchical scheduling to achieve aggregate resource partitioning among related groups of threads and applications in a multiprocessor environment. We described two limitations of existing hierarchical schedulers in multiprocessor systems: 1) their inability to incorporate the inherent parallelism of multiple processors and 2) that of unbounded unfairness due to infeasible weights. We presented H-SMP, a hierarchical CPU scheduling algorithm designed for a multiprocessor platform. This algorithm employs a combination of space and time scheduling and can incorporate existing proportional-share algorithms as auxiliary schedulers to achieve the desired hierarchical CPU partitioning. In addition, we derived a generalized weight feasibility constraint that formalizes the notion of feasible weights to avoid the problem of unbounded unfairness and developed a hierarchical weight readjustment algorithm to transparently satisfy this feasibility constraint. We evaluated the properties of H-SMP using H-SFS, an instantiation that employs SFS as an auxiliary algorithm. This evaluation was carried out through a simulation study that showed that H-SFS provides better fairness properties in multiprocessor environments as compared to existing algorithms and their naive extensions. As part of our future work, we intend to implement H-SMP and its instantiations in a real SMP environment and study its efficiency by using real-world applications. In particular, we would like to evaluate heuristics to achieve better space partitioning and exploit cache affinity.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP '03)*, Oct. 2003.

[2] C.A. Waldspurger, "Memory Resource Management in VMware ESX Server," *Proc. Fifth Usenix Symp. Operating System Design and Implementation (OSDI '02)*, Dec. 2002.

[3] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin, "Application Performance in the QLinux Multimedia Operating System," *Proc. Eighth ACM Int'l Multimedia Conf. (ACM Multimedia '00)*, Nov. 2000.

[4] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum, "Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors," *ACM Trans. Computer Systems*, vol. 18, no. 3, pp. 229-262, 2000.

[5] *Intel Dual-Core Server Processor,* http://www.intel.com/business/bss/products/server/dual-core.htm, 2006.
[6] *IBM xSeries with Dual-Core Technology,* http://www.intel.com/business/bss/products/server/dual-core.htm, 2006.
[7] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *ACM Trans. Computer Systems,* vol. 10, no. 1, pp. 53-79, Feb. 1992.
[8] G. Banga, P. Druschel, and J. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," *Proc. Third Usenix Symp. Operating System Design and Implementation (OSDI '99),* pp. 45-58, Feb. 1999.
[9] P. Goyal, X. Guo, and H. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *Proc. Second Usenix Symp. Operating System Design and Implementation (OSDI '96),* pp. 107-122, Oct. 1996.
[10] J. Bennett and H. Zhang, "Hierarchical Packet Fair Queuing Algorithms," *Proc. ACM SIGCOMM '96,* pp. 143-156, Aug. 1996.
[11] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *Proc. ACM SIGCOMM '89,* pp. 1-12, Sept. 1989.
[12] K. Duda and D. Cheriton, "Borrowed Virtual Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler," *Proc. 17th ACM Symp. Operating Systems Principles (SOSP '99),* pp. 261-276, Dec. 1999.
[13] S.J. Golestani, "A Self-Clocked Fair Queueing Scheme for High-Speed Applications," *Proc. IEEE INFOCOM '94,* pp. 636-646, Apr. 1994.
[14] P. Goyal, H.M. Vin, and H. Cheng, "Start-Time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," *Proc. ACM SIGCOMM '96,* pp. 157-168, Aug. 1996.
[15] J. Nieh and M.S. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications," *Proc. 16th ACM Symp. Operating Systems Principles (SOSP '97),* pp. 184-197, Dec. 1997.
[16] B. Caprita, W. Chan, J. Nieh, C. Stein, and H. Zheng, "Group Ratio Round-Robin: O(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems," *Proc. Usenix Ann. Technical Conf. '05,* Apr. 2005.
[17] *Solaris Resource Manager 1.0: Controlling System Resources Effectively.* Sun Microsystems, Inc., http://www.sun.com/software/white-papers/wp-srm/, 1998.
[18] C. Waldspurger and W. Weihl, "Stride Scheduling: Deterministic Proportional-Share Resource Management," Technical Report TM-528, Laboratory for Computer Science, Mass. Inst. of Technology, June 1995.
[19] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors," *Proc. Fourth Usenix Symp. Operating System Design and Implementation (OSDI '00),* Oct. 2000.
[20] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton, "A Proportional-Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems," *Proc. 17th IEEE Real Time Systems Symp. (RTSS '96),* pp. 289-299, Dec. 1996.
[21] C.A. Waldspurger and W.E. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. First Usenix Symp. Operating System Design and Implementation (OSDI '94),* Nov. 1994.
[22] R.M. Karp and V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines," *Handbook of Theoretical Computer Science—Volume A: Algorithms and Complexity,* pp. 869-941, 1990.

Abhishek Chandra received the BTech degree in computer science and engineering from the Indian Institute of Technology, Kanpur, India, in 1997 and the MS and PhD degrees in computer science from the University of Massachusetts, Amherst, in 2000 and 2005, respectively. He is currently an assistant professor in the Department of Computer Science and Engineering, University of Minnesota. His research interests include operating systems, distributed systems, and Internet systems. He received a US National Science Foundation Faculty Early Career Development (CAREER) Award in 2007, and his PhD dissertation "Resource Allocation for Self-Managing Servers" was nominated for the ACM Dissertation Award in 2005. He is a member of the IEEE, the ACM, and the Usenix.

Prashant Shenoy received the BTech degree in computer science and engineering from the Indian Institute of Technology (IIT), Bombay, in 1993 and the MS and PhD degrees in computer science from the University of Texas at Austin (UT) in 1994 and 1998, respectively. He is currently an associate professor of computer science at the University of Massachusetts, Amherst. His research interests include operating and distributed systems, sensor networks, Internet systems and pervasive multimedia. He is the recipient of a US National Science Foundation Faculty Early Career Development (CAREER) Award, the IBM Faculty Development Award, the Lilly Foundation Teaching Fellowship, the UT Computer Science Best Dissertation Award, and an IIT Silver Medal. He is a senior member of the IEEE and the ACM.