Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 06-018

Hierarchical Scheduling for Symmetric Multiprocessors

Abhishek Chandra and Prashant Shenoy

May 22, 2006

Hierarchical Scheduling for Symmetric Multiprocessors

Abhishek Chandra
Department of Computer Sc. and Engg.
University of Minnesota
Minneapolis, MN 55455
chandra@cs.umn.edu

Prashant Shenoy
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
shenoy@cs.umass.edu

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 (i) they do not incorporate the inherent parallelism of a multiprocessor system while resource partitioning, and (ii) they can result in unbounded unfairness or starvation if applied to a multiprocessor system in a naive manner. In this paper, we present a novel hierarchical scheduling algorithm designed specifically for multiprocessor environments that overcomes the limitations of existing algorithms in several ways. 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 then present *hierarchical multiprocessor scheduling (H-SMP):* a 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 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. We evaluate the properties of this algorithm 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

I. INTRODUCTION

A. Motivation

Recent advances in computing have seen the emergence of a wide diversity of computing environments, including servers (e.g., Web servers and multimedia servers), versatile desktop environments (running compilers, browsers, and multi-player games), as well as parallel computing and scientific applications. These environments are comprised of 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 run-time 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, e.g., all threads of a parallel application should be proceeding at the same rate to minimize the “makespan” of the application or its 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 guarantees, and hence, may be scheduled together by a soft-real time scheduler, instead of the default scheduler provided by the operating system they are running on. 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 multi-core environments due to several reasons. First of all, resource partitioning can help in making large multiprocessor systems scalable by reducing excessive inter-processor 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 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 and more systems move to multi-core technology [5], [6][1]. Dual-core machines are already common, and uniprocessors are expected to become exceptions 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 system-wide 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 have to be deployed in a complementary scheduling framework to exploit their properties.

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

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 (i) they do not incorporate the inherent parallelism of a multiprocessor system while resource partitioning, and (ii) 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.

B. 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]–[18] 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 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. Surplus Fair Scheduling [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 III, this weight feasibility constraint is not sufficient for a hierarchical scheduling framework, and hence, these algorithms by themselves are inadequate for direct use in a hierarchical multiprocessor scheduling environment.

Besides 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 IV, 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. Therefore, there is a need for a hierarchical scheduler designed explicitly for multiprocessor environments.

Lottery scheduling [20] 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 non-trivial. Tickets by themselves can be used as an accounting mechanism in a hierarchical framework, and are orthogonal to our discussion here.

C. Research Contributions

This paper presents a novel hierarchical scheduling algorithm designed specifically 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 present 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.

We then present *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. Thus, H-SMP is general in its construction, and can incorporate suitable schedulers based on tradeoffs 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, though this choice can affect how close H-SMP is to the ideal partitioning.

Finally, we present *hierarchical surplus fair scheduling (H-SFS):* an instantiation of H-SMP that employs surplus fair scheduling (SFS) [19] as the auxiliary algorithm, and 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.

The rest of this paper is organized as follows. In Section II, we present the system model assumed in this paper. Section III presents a feasibility constraint on the achievable bandwidth partitioning in a symmetric multiprocessor (SMP) system, and a hierarchical readjustment algorithm that can be plugged in with a scheduler to meet this constraint. Section IV presents the hierarchical multiprocessor scheduling algorithm (H-SMP), and describes a particular instantiation of it using surplus fair scheduling [19]. We present simulation results in Section V to evaluate the algorithm, and conclude in Section VI with a summary and discussion of future research directions.

II. SYSTEM MODEL

Our system model consists of a p-CPU symmetric multiprocessor 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

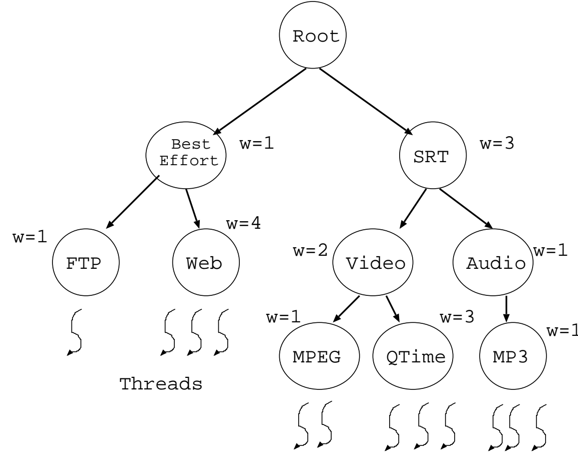


Fig. 1. A scheduling hierarchy: multiple threads and applications are grouped together in a scheduling tree.

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, while each internal (non-leaf) node in the hierarchy corresponds to either a service class or a multi-threaded application[3]. The root of the tree represents the aggregation of all threads in the system.

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 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.

Example 1: Figure 1 illustrates an example scheduling hierarchy that has two service classes: *best-effort* (BE) and *soft real-time* (SRT). The BE class consists of two multi-threaded applica-

[2]In the rest of the 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.

tions: an FTP server and a Web server. The SRT class is subdivided into two classes: audio and video. The video class consists of an MPEG and a Quicktime server application, while the audio class consists of an MP3 server application. The weight of each of these nodes in the scheduling hierarchy is illustrated in the figure. Based on these weights, the BE and SRT classes should receive 25% and 75% 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% and 20% of the system CPU service respectively. The desired shares of other nodes can be similarly computed in a top-down manner.

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): 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, i.e., threads that node i could potentially schedule in parallel on different CPUs.

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.

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 number of processors, assigned to node i for running threads in its subtree.

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].

While 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.

III. Hierarchical Weight Readjustment

From the description of the hierarchical scheduling model, we see that the partitioning of the CPU bandwidth is critically dependent on the weights assigned to the nodes in the scheduling tree. These weights are typically assigned externally based on the requirements of applications and application classes. However, as shown in [19], in the context of proportional-share scheduling, an infeasible weight assignment leads to unbounded unfairness or starvation among threads in multiprocessor environments. In this section, we first present a formal constraint on feasible weight assignments that is applicable to the nodes of a scheduling tree, and then present a hierarchical readjustment algorithm that transparently converts any weight assignment to such a feasible assignment.

A. Generalized Weight Feasibility Constraint

A proportional-share algorithm can suffer from unbounded unfairness or starvation problem in multiprocessor environments, as illustrated by the following example.

Example 2: Consider a server that employs the start-time fair queueing (SFQ) algorithm [14] to schedule threads. SFQ is a GPS-based fair scheduling algorithm that assigns a weight w_i to each thread and allocates bandwidth in proportion to these weights. To do so, SFQ maintains a counter S_i for each application that 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 processor. Assume that the server has two processors and runs two compute-bound threads that are assigned weights $w_1 = 1$ and $w_2 = 10$, respectively. Let the quantum

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

duration be $q = 1ms$. Since both threads are compute-bound and SFQ is work-conserving,[5] each thread gets to continuously run on a processor. After 1000 quantums, we have $S_1 = \frac{1000}{1} = 1000$ and $S_2 = \frac{1000}{10} = 100$. 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.

While 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 3 threads in a 4-CPU system can utilize $\frac{3}{4}^{th}$ of the total CPU bandwidth, and is thus not constrained by the weight feasibility constraint (Relation 4). However, the number of CPUs that a node can utilize is still constrained by the number of threads it has in its subtree: for the above example, the node cannot utilize *more than* $\frac{3}{4}^{th}$ 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 (Equation 3), Relation 5 can be written as

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

[5] 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.

where, P is the parent node of node i and C_P is the set of P's children nodes. We refer to Relation 6 as the *generalized weight feasibility constraint*. Intuitively, this constraint specifies that a node cannot be assigned more CPU capacity than it can utilize through its parallelism. Note that Relation 6 reduces to the weight feasibility constraint (Relation 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 I):

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.

B. Hierarchical Weight Readjustment

As shown above, Relation 6 specifies a feasibility constraint on the weights assigned to nodes in a scheduling hierarchy. However, given a scheduling hierarchy, it is possible that some nodes in the hierarchy have infeasible weights. This is possible because the externally-assigned weights to the nodes 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 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.

Algorithm 1 gen_readjust(array $[w_1 ... w_n]$, float π), Returns $[\phi_1 ... \phi_n]$

1: **if** $\left(\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi} \right)$ **then**
2:     **gen_readjust**($[w_2 ... w_n]$, $\pi - \theta_1$)
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**

Algorithm 1 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 Relation 6.

This algorithm determines the adjusted weight of a node based on its original weight as well as the number of threads it can schedule. Intuitively, if a node demands more CPUs than the number of threads it can schedule, the algorithm assigns it as many CPUs as 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 non-increasing 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 Relation 6. Ordering the nodes by their weight-parallelism ratio ensures that constraint-violating (infeasible) nodes are always placed before nodes satisfying the constraint (feasible nodes)[6]. 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 2 hier_readjust(tree_node *node*)

1: **gen_readjust**(*node*.weight_list, π_{node})
2: **for all** *child* in the set of *node*'s children C_{node} **do**
3: $\quad \pi_{child} \leftarrow \left(\frac{\phi_{child}}{\sum_{j \in C_{node}} \phi_j}\right) \cdot \pi_{node}$
4: $\quad$ **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 2). Intuitively, this algorithm traverses the tree in a depth-first manner[7], and for each node P, the algorithm (i) applies the generalized weight readjustment algorithm to the children of node P, and (ii) computes the processor assignment for the children of P using Equation 3 based on their adjusted weights ϕ_i. The working of the hierarchical weight readjustment is illustrated in the following example:

Example 3: Once again consider the scheduling hierarchy described in Example 1 (shown in Figure 1), now running on a 16-CPU system. Here, we see how the hierarchical weight

[6]We prove this property of the ordering in Appendix I. The intuitive reason is that nodes that have higher weights or have fewer number of threads to schedule are more likely to violate the feasibility constraint (Relation 6).

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

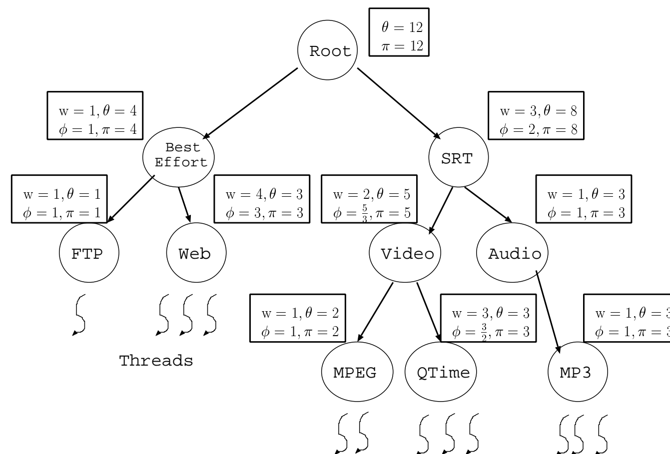


Fig. 2. Application of the hierarchical weight readjustment algorithm on a scheduling hierarchy.

readjustment algorithm (Algorithm 2) is used to modify the weights of the nodes in the scheduling tree. Figure 2 shows the original (w) and adjusted weights (ϕ) for the nodes in the scheduling tree along with their values of thread parallelism and processor assignment after applying the algorithm. To begin with, since the value of $\theta_{root} = 12$, we have $\pi_{root} = \min(12, 16) = 12$. In the next step, the weights of the Best-Effort and SRT node are modified using the generalized weight readjustment algorithm (Algorithm 1). The weight of the SRT node is infeasible to begin with (as it has a θ value of 8, while its CPU demand based on its original weight is 9 CPUs). This weight is adjusted so that the CPU demand of the SRT node becomes 8 CPUs, that can be utilized by the threads in its subtree. The weights of the other nodes in the tree are similarly adjusted in a top-down manner.

C. Properties of Hierarchical Weight Readjustment

In this section, we first present the properties of the generalized weight readjustment algorithm. We then present its running time 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 I.

First of all, the generalized weight readjustment algorithm ensures that no node demands more CPU service than it can utilize. The following theorem states this correctness property of the algorithm:

Theorem 2: The adjusted weights assigned by the generalized weight readjustment algorithm satisfy the generalized weight feasibility constraint.

Besides 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, while the remaining nodes retain their original weights. This property of the generalized weight readjustment algorithm is stated in the following theorem:

Theorem 3: 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 they can utilize.

These properties intuitively specify that the algorithm does not change the weight of a node unless 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[8]. Since the generalized weight readjustment algorithm examines only the infeasible nodes, its time complexity is given by the following theorem:

Theorem 4: 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.

[8]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.

Theorem 5: 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 5 implies that the running time 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.

IV. HIERARCHICAL MULTIPROCESSOR SCHEDULING

In the previous section, we presented an algorithm for the readjustment of weights assigned to the nodes in the scheduling tree, in order to make them feasible. Given a set of feasible weight assignments for the tree nodes, the next step is to schedule the threads in a manner that enables different nodes in the hierarchy to meet their CPU requirement. 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. We then present a hierarchical scheduling algorithm that incorporates an auxiliary thread-scheduling[9] algorithm to achieve hierarchical proportional-share scheduling in a multiprocessor environment. This scheduling algorithm is based on a combination of space- and time-multiplexing.

A. Limitations of Existing Hierarchical Scheduling Algorithms

Algorithm 3 hier_sched()

```
1: node ← root
2: while node is not leaf do
3:    node ← gen_sched(node) {gen_sched is an algorithm that selects a child of node for scheduling}
4: end while
```

Algorithm 3 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

[9]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).

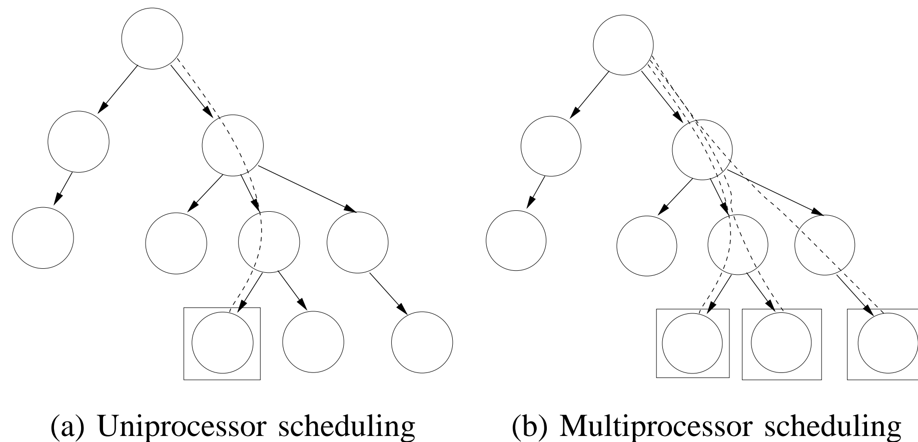


Fig. 3. Hierarchical scheduling represented as scheduling of paths from the root to threads (leaf nodes): (a) shows that only a single thread and hence a single path is scheduled in a uniprocessor environment, while (b) shows that multiple threads and hence multiple overlapping paths in the scheduling tree can be concurrently scheduled in a multiprocessor environment.

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. Figure 3(a) 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 requires certain properties from the algorithm (gen_sched) for selecting a node at each level. In uniprocessor environments, a thread-scheduling 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 start-time fair queuing (SFQ) [14], a thread-scheduling proportional-share algorithm.

However, using a similar approach, namely to use 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. Scheduling a thread to run on a CPU is equivalent to scheduling each node on the path from the root to the node (Figure 3(a)). These intermediate nodes can then be scheduled (or selected) using a thread-scheduling algorithm. 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. Figure 3(b) illustrates this distinction for multiprocessor environments. 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 parallelism can be achieved only by scheduling an internal node multiple times concurrently, i.e., 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 to assign 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 to design 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:

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

Next, we present an algorithm that addresses these challenges by employing a combination of space- and time-multiplexing.

B. H-SMP: Hierarchical Multiprocessor Scheduling

In this section, we present *hierarchical multiprocessor scheduling* (H-SMP), a scheduling algorithm designed to achieve hierarchical scheduling in a multiprocessor environment. 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 proportional-share thread-scheduling algorithm to perform this CPU assignment in order to achieve desired CPU service for the tree nodes.

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 integral number of CPUs to assign to each node in the hierarchy. The auxiliary scheduler is a proportional-share thread-scheduling algorithm (such as SFQ, surplus fair scheduling (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 remaining 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 additional processor requirement can be achieved by employing the auxiliary scheduler. This auxiliary scheduler time-multiplexes the remaining CPU bandwidth among the virtual nodes to satisfy their remaining 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,

$$r_i = \sum_{j \in C_i} r_j, \tag{7}$$

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 are scheduled in FIFO order by the space scheduler.

- *Low-threshold set:* The low-threshold set consists of those 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.
- *High-threshold set:* This set consists of those nodes for which $r_i \geq \lceil \pi_i \rceil$, i.e., the ones that are currently assigned at least the upper threshold of their requirement. These nodes are considered ineligible for scheduling.

We next illustrate a practical instantiation of H-SMP using surplus fair scheduling (SFS) [19] as the auxiliary scheduler.

C. H-SFS: An Instantiation of H-SMP

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 surplus fair scheduling (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 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]:

$$\alpha_i = A_i(0,T) - A_i^{ideal}(0,T), \tag{8}$$

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 subsection, 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 computing the surplus of a node i at a time T using Equation 8:

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

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[10]. 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), as in that case, the weight feasibility constraint requires that $0 < \pi_i \leq 1, \forall i$, which means that all threads are either low-threshold (if they are not currently running) or high-threshold (if they are currently running) at any scheduling instant. The low-threshold threads (i.e., the ones in the run-queue) are then scheduled in the order of their surplus values.

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

D. Properties of H-SMP

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. Further, 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. We relax this synchronization requirement in the next subsection and consider the effect on the properties of H-SMP. For the non-trivial 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 II).

Theorem 6: 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 6, 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 1 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 and a lower threshold that are dependent on its processor assignment.

V. Simulation Study

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

A. Simulation Methodology

In our study, we simulated multiprocessor systems with different number of processors. 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, while 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). By using different seeds for our random number generators, we generated different tree structures for the same values of N and n. These nodes and threads were then assigned weights chosen uniformly at random from a fixed range of values. Further, each thread was assumed to be runnable for the whole duration of the simulation.

Each run of the simulation is conducted as follows. The system time is measured in ticks, and the maximum scheduling quantum is defined to be a multiple of ticks. Each CPU is interrupted at a time chosen uniformly at random within its quantum, at which point it calls the hierarchical scheduler to assign the next thread to run on the CPU[11]. We assume a fixed set of threads and a fixed scheduling hierarchy during each run.

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

1) *SFS (Surplus Fair Scheduling):* SFS is a proportional-share algorithm designed to schedule threads in a multiprocessor environment. In our simulations, we employed SFS algorithm directly to schedule nodes at each level of the scheduling hierarchy. Note that SFS is a thread-scheduling algorithm that is not designed to assign multiple CPUs to each scheduling entity, and hence cannot exploit the inherent parallelism of internal tree nodes.
2) *Ext-SFS:* 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

[11]Threads may not use full quantum lengths either because of having used up partial quantum lengths in their previous runs, or due to pre-emption or blocking events. Here, we consider only the first two reasons for variable-length quanta.

tree, as was shown for H-SMP. We use Ext-SFS to represent one possible extension of an existing multiprocessor proportional-share algorithm.

3) *H-RR (Hierarchical Round Robin):* H-RR is an instantiation of H-SMP that employs Round Robin algorithm as the auxiliary scheduler. This algorithm also employs a space scheduler (and the notions of deficit, low-threshold, and high-threshold sets) to assign processors to nodes. However, the nodes in the low-threshold 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 scheduling algorithm.
4) *H-SFS:* This is the instantiation of H-SMP presented in Section IV-C, which uses SFS as an auxiliary scheduler in the H-SMP scheduling framework.

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.

In our study, we simulated multiprocessor systems with 2, 4, 8, 16, and 32 CPUs respectively. We generated scheduling trees with 2, 4, 8, and 10 internal nodes, and a set of values for the number of threads in the system varying from 3 to 100. For each of these parameter combinations, we ran 100 simulations for each scheduler using different random number generator seeds. Next, we present the results of our simulation study.

B. Comparison of Schedulers

Figures 4(a), (b), and (c) show the comparison of the algorithms described above for 2-, 8-, and 32-processor systems respectively. These figures plot the *mean* deviation from the ideal share

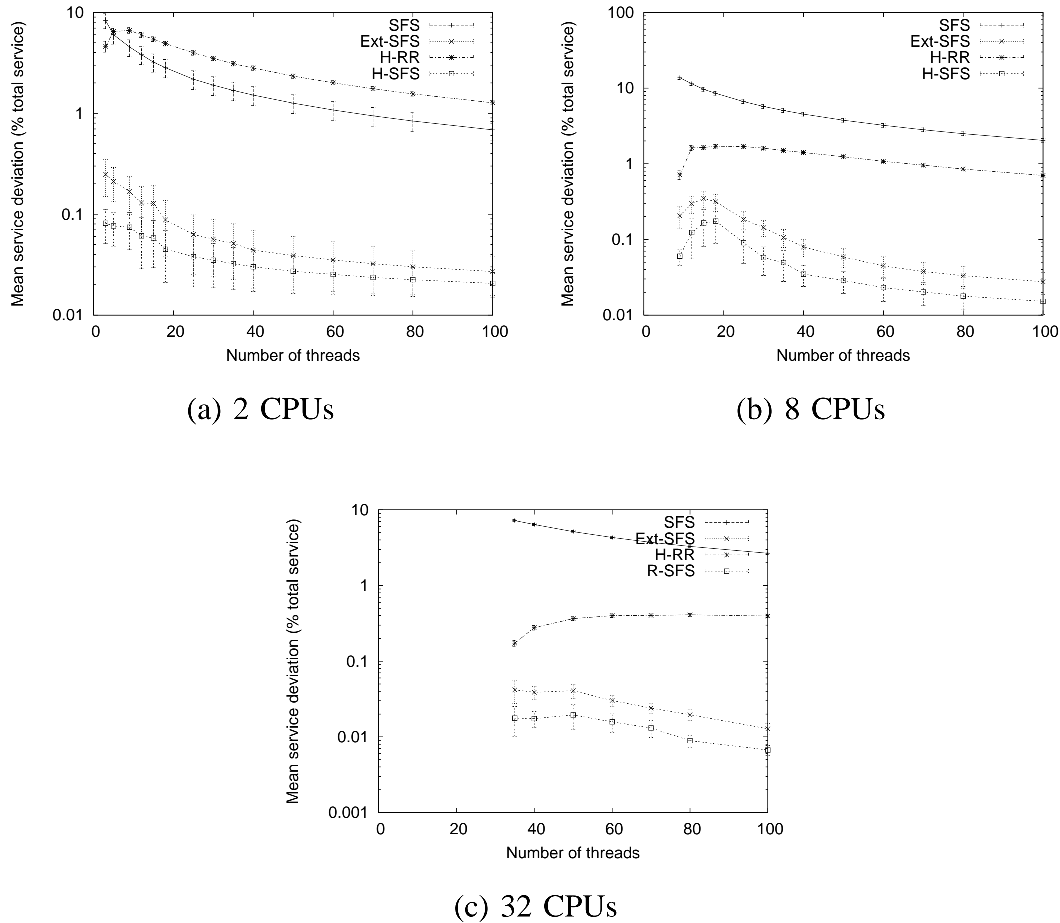


Fig. 4. Mean deviation for scheduling trees with 10 internal nodes on different size multiprocessor systems.

for all the nodes in the tree. These figures show results for scheduling trees with 10 internal nodes in each case. As can be seen from the figures, the SFS algorithm has the highest deviation. This is because SFS assigns at most 1 CPU to each node, resulting in large deviations for nodes which 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, as seen from Figure 5 which plots 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%, 5.89%, and 1.75% for a 2-CPU, 8-CPU, and a 32-CPU system 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

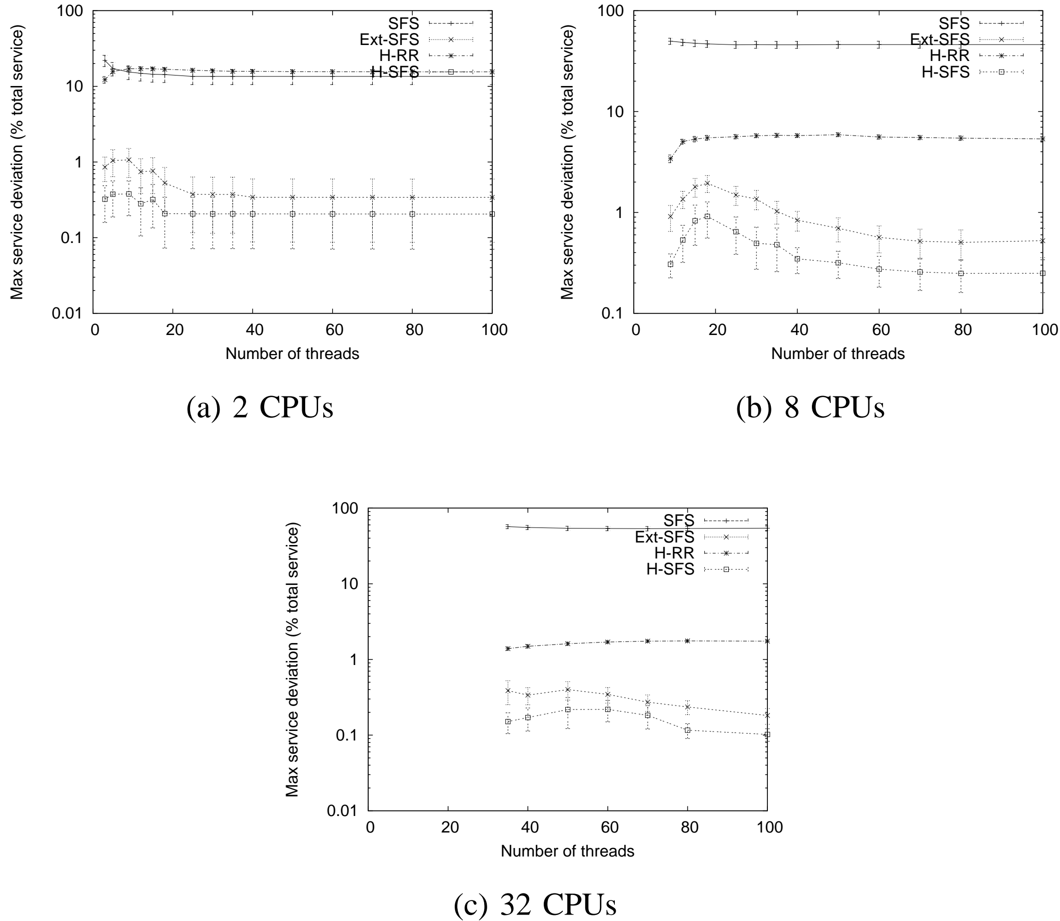
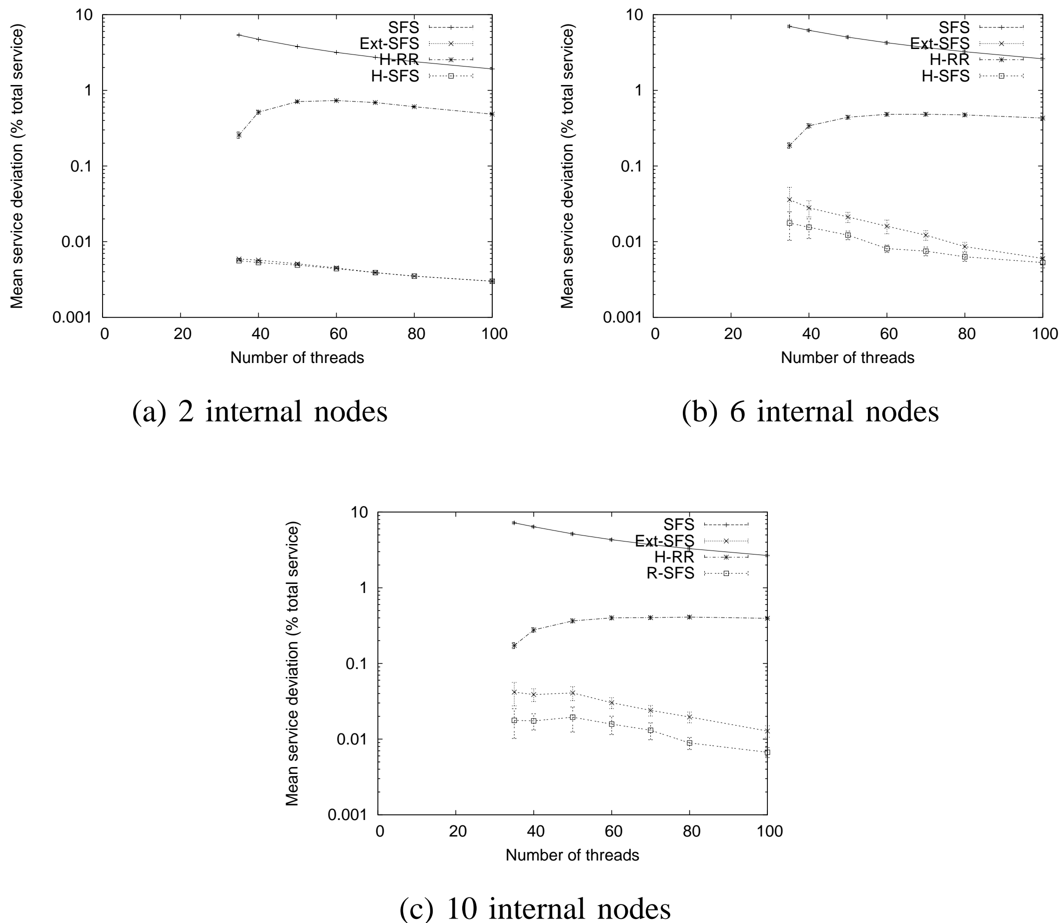


Fig. 5. Maximum deviation for scheduling trees with 10 internal nodes on different size multiprocessor systems.

CPUs available to any node in the scheduling tree is bounded by the upper and lower thresholds of its processor requirement. However, since 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. Further, we see that H-SFS has the smallest deviation values, which indicates that a combination of threshold bounds along with a proportional-share algorithm provides the best performance in terms of achieving proportional-share allocation.

Similarly, Figures 6(a), (b) and (c) show the comparison of the algorithms for scheduling trees with different sized (2, 6, and 10 internal nodes) running on a 32-processor system. As can be seen from the figures, the results are similar to those obtained above, and the H-SFS algorithm



(a) 2 internal nodes (b) 6 internal nodes

(c) 10 internal nodes

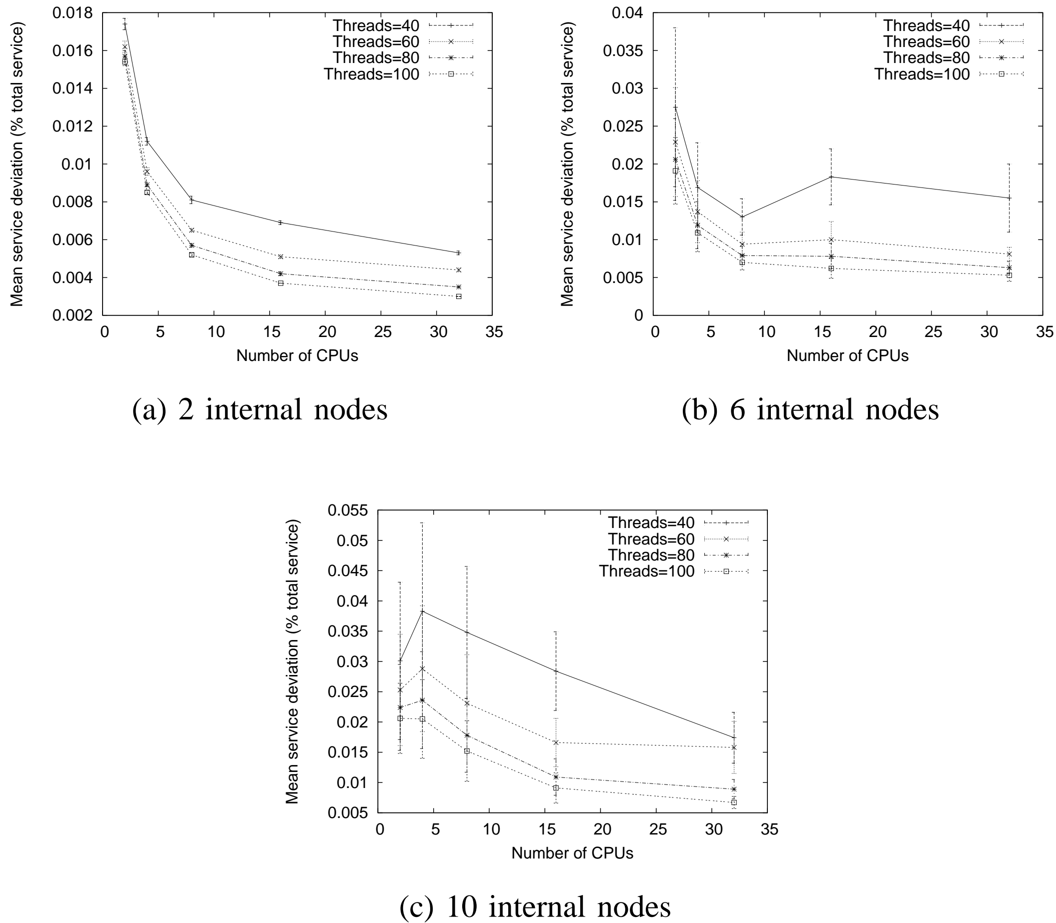
Fig. 6. Mean deviation for scheduling trees with different sizes on a 32-processor system.

again has the least mean deviation values among the algorithms considered here.

These results demonstrate that using a thread-scheduling algorithm is ineffective for exploiting thread parallelism in a scheduling tree. Further, we see that a simple extension of a proportional-share algorithm such as SFS is more effective in achieving desirable allocation than fitting in a naive algorithm such as Round Robin in the hierarchical framework. Finally, the results show that a combination of space scheduling coupled with the employment of a proportional-share algorithm such as SFS as an auxiliary scheduler achieves the most desirable allocation.

C. Impact of System Parameters

Now, we consider the effect of system parameters such as the number of processors, number of threads, and tree size on the performance of H-SFS. In Figure 7, we plot the mean deviation as



(a) 2 internal nodes

(b) 6 internal nodes

(c) 10 internal nodes

Fig. 7. Effect of number of processors on the deviation of H-SFS.

the number of CPUs is varied. As can be seen from the figures, the mean deviation decreases as we increase the number of processors in the system. This is because as the number of processors increases, there are more processors available to schedule the nodes, and hence, there is less contention and waiting delay for each node. Hence, on an average, there is lower deviation from the desired share as we increase the number of CPUs.

In Figure 8, we plot the mean deviation as the tree size (in number of internal nodes) is varied. As can be seen from the figures, the mean deviation *increases* as we increase the tree size. This is because, with larger number of nodes in the tree, there is more contention among different nodes in terms of receiving their shares, leading to greater unfairness.

However, from both Figures 7 and 8, we see that the deviation decreases as we increase the number of threads. The main reason is that, for the same system configuration, as the number

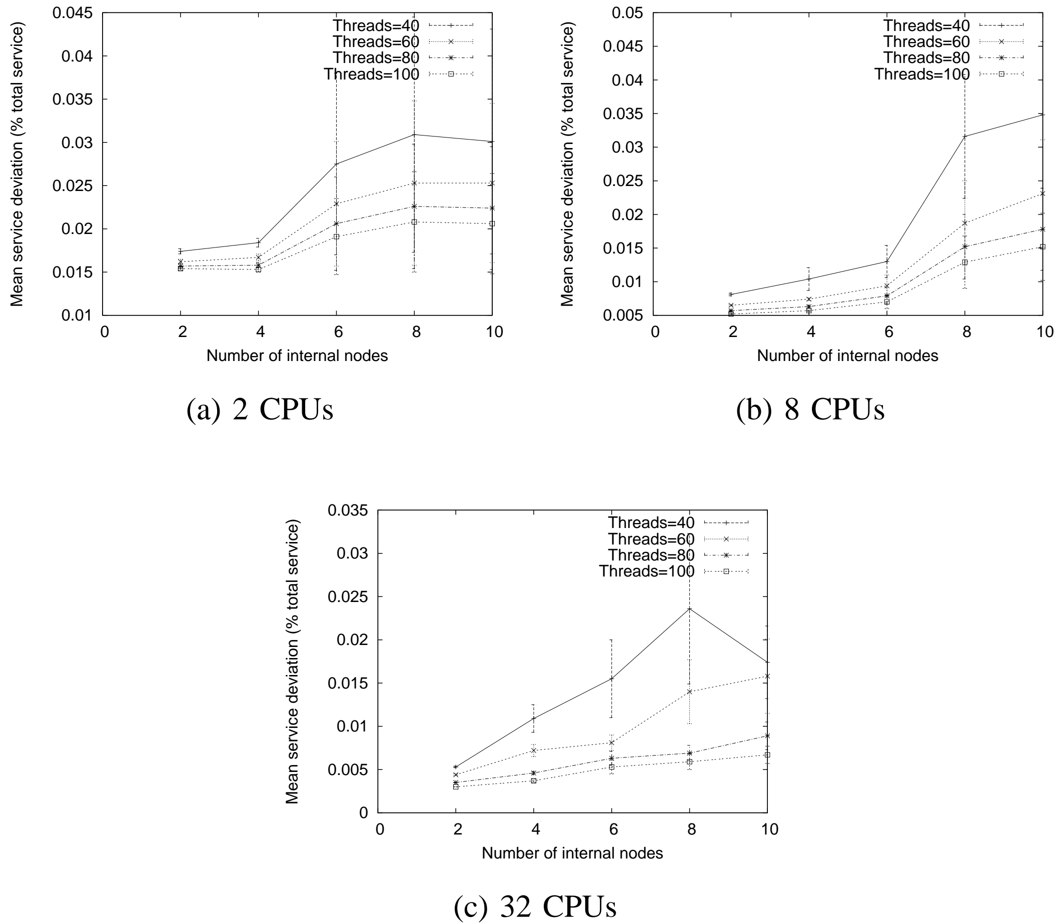


Fig. 8. Effect of tree size on the deviation of H-SFS.

of threads increases, the average share received by each thread also decreases, which in turn leads to smaller deviation. Another way to understand this phenomenon is to observe that with larger number of threads, there is a smaller probability of CPUs remaining idle in the presence of runnable threads.

Overall, our results demonstrate that H-SFS is effective in reducing the deviation of tree nodes from their desired shares. We also show that the performance of H-SFS improves in the presence of large number of CPUs, smaller tree sizes, and large number of runnable threads.

VI. Concluding Remarks

In this paper, we considered the problem of using hierarchical scheduling as a scheduling technique to achieve aggregate resource partitioning among related groups of threads and appli-

cations in a multiprocessor environment. We presented the limitations of existing hierarchical schedulers for multiprocessor systems, by defining a feasibility constraint on the achievable CPU bandwidth partitioning in a multiprocessor hierarchical framework. We then presented H-SMP: a hierarchical CPU scheduling algorithm designed for a symmetric multiprocessor (SMP) platform. This algorithm employs a combination of space- and time-scheduling to achieve desired bandwidth partitioning among the hierarchical tree nodes. This algorithm is also characterized by its ability to incorporate existing proportional-share algorithms as auxiliary schedulers to achieve efficient hierarchical CPU partitioning. We evaluated the properties of this algorithm using H-SFS: an instantiation of H-SMP that employs surplus fair scheduling (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 future work, we intend to implement H-SMP and its instantiations in a real SMP environment and study its efficiency using real-world applications. One of the questions that we would like to answer is if we can improve efficiency and achieve greater space-scheduling by reducing the scheduling decisions across different subtrees. We would also like to evaluate different heuristics to incorporate cache affinity in an SMP environment.

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," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'03)*, October 2003.

[2] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002)*, Dec. 2002.

[3] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin, "Application Performance in the QLinux Multimedia Operating System," in *Proceedings of the Eighth ACM Conference on Multimedia, Los Angeles, CA*, November 2000.

[4] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum, "Cellular disco: resource management using virtual clusters on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 229–262, 2000.

[5] "Intel Dual-Core Server Processor," 2006, http://www.intel.com/business/bss/products/server/dual-core.htm.

[6] "IBM xSeries with Dual-Core Technology," 2006, http://www.intel.com/business/bss/products/server/dual-core.htm.

[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 Transactions on 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," in *Proceedings of the third Symposium on Operating System Design and Implementation (OSDI'99), New Orleans*, February 1999, pp. 45–58.

[9] P. Goyal, X. Guo, and H. Vin, “A Hierarchical CPU Scheduler for Multimedia Operating Systems,” in *Proceedings of Operating System Design and Implementation (OSDI’96), Seattle*, October 1996, pp. 107–122.

[10] J. Bennett and H. Zhang, “Hierarchical Packet Fair Queuing Algorithms,” in *Proceedings of SIGCOMM’96*, August 1996, pp. 143–156.

[11] A. Demers, S. Keshav, and S. Shenker, “Analysis and Simulation of a Fair Queueing Algorithm,” in *Proceedings of ACM SIGCOMM*, September 1989, pp. 1–12.

[12] K. Duda and D. Cheriton, “Borrowed Virtual Time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-Purpose Scheduler,” in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP’99), Kiawah Island Resort, SC*, December 1999, pp. 261–276.

[13] S. J. Golestani, “A Self-Clocked Fair Queueing Scheme for High Speed Applications,” in *Proceedings of INFOCOM’94*, April 1994, pp. 636–646.

[14] P. Goyal, H. M. Vin, and H. Cheng, “Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks,” in *Proceedings of ACM SIGCOMM’96*, August 1996, pp. 157–168.

[15] J. Nieh and M. S. Lam, “ The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications,” in *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP’97), Saint-Malo, France*, December 1997, pp. 184–197.

[16] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng, “Group Ratio Round-Robin: O(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems,” in *Proceedings of the 2005 USENIX Annual Technical Conference*, 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,” MIT, Laboratory for Computer Science, Tech. Rep. TM-528, June 1995.

[19] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, “Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors,” in *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, CA*, October 2000.

[20] C. A. Waldspurger and W. E. Weihl, “Lottery Scheduling: Flexible Proportional-share Resource Management,” in *Proceedings of symposim on Operating System Design and Implementation*, November 1994.

APPENDIX I

PROOF OF PROPERTIES OF GENERALIZED WEIGHT READJUSTMENT

In this appendix, we formally present and prove the properties of the generalized weight readjustment algorithm (Algorithm 1). We first prove the correctness of the algorithm and then present the properties of the weights assigned by the algorithm. Finally, we analyze the time complexity of the algorithm.

In what follows, we consider a set of n sibling nodes in the scheduling hierarchy, numbered from 1 to n. We assume that these nodes have been originally assigned weights $w_1, w_2, \ldots, w_n$, and their thread parallelism values are $\theta_1, \theta_2, \ldots, \theta_n$ respectively. Further assume that the indices 1 to n for the nodes are assigned in the non-increasing order of their weight-parallelism ratio $\left(\frac{w_i}{\theta_i}\right)$, i.e., $\frac{w_i}{\theta_i} \geq \frac{w_j}{\theta_j}, \forall i < j$. The new adjusted weights of these nodes assigned by the generalized weight readjustment algorithm are denoted by $\phi_1, \phi_2, \ldots, \phi_n$ respectively. Finally, let π be the processor assignment of the parent of these nodes. We abstract the parent node's processor assignment as a π-CPU system, which corresponds to the parent node receiving $(\frac{\pi}{p})$ fraction of the total CPU service in a p-CPU system. We use these assumptions in the remainder of this appendix unless stated otherwise.

We first show that the generalized weight feasibility constraint

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

is a *necessary* condition for any work-conserving scheduler to achieve proportional-share allocation in a multiprocessor system.

Theorem 1: Consider a π-processor system running n nodes with weights $w_1, w_2, \ldots, w_n$ respectively. Let the nodes have thread parallelism values of $\theta_1, \theta_2, \ldots, \theta_n$ respectively, such that $\sum_{j=1}^{n} \theta_j \geq \pi$. Then, no work-conserving scheduler can divide the CPU bandwidth among the nodes in proportion to their weights if any node violates the generalized weight feasibility constraint, i.e., if for any node i,

$$\frac{w_i}{\sum_{j=1}^{n} w_j} > \frac{\theta_i}{\pi}.$$

Proof: Proof by contradiction.

Suppose there exists a work-conserving scheduler S that can divide the CPU bandwidth among the nodes in proportion to their weights. This implies that the CPU share s_j allocated to node

j is given by

$$s_j = \frac{w_j}{\sum_{l=1}^{n} w_l}, \forall j = 1, \ldots, n. \tag{1}$$

Further, let i be a node that violates the generalized weight feasibility constraint.

Now, since S is work-conserving, no CPU is allowed to be idle if there is an unassigned runnable thread. As $\sum_{j=1}^{n} \theta_j \geq \pi$, and all threads are assumed to be continuously backlogged, all the CPUs are going to be busy at all times. This means that in any time interval $[t_1, t_2)$, the total CPU service in the system is

$$A(t_1, t_2) = \pi \cdot (t_2 - t_1).$$

Also, since θ_i is the maximum number of CPUs i can utilize at any time, its CPU service during $[t_1, t_2)$ is given by

$$A_i(t_1, t_2) \leq \theta_i \cdot (t_2 - t_1).$$

Since $[t_1, t_2)$ is any arbitrary time interval, the CPU share received by node i,

$$\begin{aligned} s_i &\leq \max_{t1<t2} \frac{A_i(t_1, t_2)}{A(t_1, t_2)} \\ &\leq \frac{\theta_i}{\pi} \\ &< \frac{w_i}{\sum_{j=1}^{n} w_j} \end{aligned}$$

This contradicts Equation 1. Hence, proved by contradiction.

■

Algorithm 1 gen_readjust(array $[w_1 ... w_n]$, float π), Returns $[\phi_1 ... \phi_n]$

1: **if** $\left(\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi}\right)$ **then**
2: **gen_readjust**($[w_2 ... w_n]$, $\pi - \theta_1$)
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**

Next, we examine the properties of the generalized weight readjustment algorithm (Algorithm 1). We first show that ordering the nodes in the non-increasing order of their weight-parallelism ratio $\left(\frac{w_i}{\theta_i}\right)$ ensures that infeasible nodes are always placed before feasible nodes. This property has implications on the efficiency of the algorithm, as we show later.

Lemma 1: Consider two nodes i and j with weights w_i and w_j, and node parallelism θ_i and θ_j respectively. If $\frac{w_i}{\theta_i} \leq \frac{w_j}{\theta_j}$, then, node i violates the generalized weight feasibility constraint only if node j violates it.

Proof: If node i violates the generalized weight feasibility constraint, then, we have,

$$\frac{w_i}{\theta_i} > \frac{\sum_{l=1}^{n} w_l}{\pi}.$$

Thus,

$$\begin{aligned} \frac{w_j}{\theta_j} &\geq \frac{w_i}{\theta_i} \\ &> \frac{\sum_{l=1}^{n} w_l}{\pi}, \end{aligned}$$

and hence, node j also violates the constraint.

■

Now we give the correctness proof of the generalized readjustment algorithm.

Theorem 2: The adjusted weights assigned by the generalized weight readjustment algorithm satisfy the generalized weight feasibility constraint, i.e.,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \leq \frac{\theta_i}{\pi}, \forall i \in \{1, \ldots, n\}.$$

Proof: Proof by induction on the number of CPUs (π):

Base Case: For $\pi \leq 1$,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \leq 1, \forall i \in \{1, \ldots, n\},$$

while,

$$\frac{\theta_i}{\pi} \geq \theta_i \geq 1, \forall i \in \{1, \ldots, n\}.$$

Hence, the hypothesis holds.

Inductive Step: Suppose the property holds for all π upto Π.

Now consider π processors s.t. $\Pi < \pi \leq \Pi + 1$. Let $\pi' = \pi - \theta_1$, where θ_1 is the parallelism of node 1.

Note that $\pi' \leq \Pi$.

Based on the weight w_1 of node 1, we have the following cases:

Case (a):

$$\frac{w_1}{\sum_{j=1}^{n} w_j} \leq \frac{\theta_1}{\pi} \tag{2}$$

By step 5 of the algorithm,

$$\phi_i = w_i, \forall i \in \{1, \ldots, n\}.$$

Hence,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} = \frac{w_i}{\sum_{j=1}^{n} w_j}, \forall i \in \{1, \ldots, n\}.$$

Thus, by Relation 2 and Lemma 1,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \leq \frac{\theta_i}{\pi}, \forall i \in \{1, \ldots, n\}.$$

Case (b):

$$\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi}$$

By the inductive hypothesis, **gen_readjust**($[w_2, w_3, \ldots, w_n], \pi'$) would return $[\phi_2, \phi_3, \ldots, \phi_n]$ s.t.

$$\frac{\phi_i}{\sum_{j=2}^{n} \phi_j} \leq \frac{\theta_i}{\pi'}, \forall i \in \{2, \ldots, n\} \tag{3}$$

By steps 2 and 3 of the algorithm,

$$\phi_1 = \frac{\theta_1}{\pi'} \cdot \sum_{j=2}^{n} \phi_j.$$

Thus,

$$\begin{aligned}
\sum_{j=1}^{n} \phi_j &= \frac{\theta_1}{\pi'} \cdot \sum_{j=2}^{n} \phi_j + \sum_{j=2}^{n} \phi_j \\
&= \frac{\theta_1 + \pi'}{\pi'} \cdot \sum_{j=2}^{n} \phi_j \\
&= \frac{\pi}{\pi'} \cdot \sum_{j=2}^{n} \phi_j
\end{aligned}$$

Hence,

$$\begin{aligned}
\frac{\phi_1}{\sum_{j=1}^{n} \phi_j} &= \frac{\frac{\theta_1}{\pi'} \cdot \sum_{j=2}^{n} \phi_j}{\frac{\pi}{\pi'} \cdot \sum_{j=2}^{n} \phi_j} \\
&= \frac{\theta_1}{\pi}
\end{aligned}$$

Finally, $\forall i \in \{2, \ldots, n\}$,

$$\begin{aligned} \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} &= \frac{\phi_i}{\frac{\pi}{\pi'} \cdot \sum_{j=2}^{n} \phi_j} \\ &\leq \frac{\pi'}{\pi} \cdot \frac{\theta_i}{\pi'} \qquad \text{(By Relation 3)} \\ &\leq \frac{\theta_i}{\pi} \end{aligned}$$

Hence, proved by induction.

■

Having given the correctness proof of the generalized weight readjustment algorithm, we now show that the adjusted weights assigned by the algorithm are "closest" to the original weights in the following sense:

1) Nodes that are assigned fewer CPUs than their thread parallelism are assigned their original weights.
2) Nodes with an original CPU demand exceeding their thread parallelism are assigned the maximum number of CPUs they can utilize.

We first present a lemma that helps in proving the above properties.

Lemma 2: The set of adjusted weights ϕ_i assigned by the generalized weight readjustment algorithm satisfy the following condition:

$$\sum_{i=1}^{n} \phi_i \leq \sum_{i=1}^{n} w_i.$$

Proof: Proof by induction on the number of CPUs (π):

Base Case: For $\pi \leq 1$, the property holds trivially, as,

$$[\phi_1, \ldots, \phi_n] = [w_1, \ldots, w_n].$$

Inductive Step: Suppose the property holds for all π upto Π.

Now consider π processors s.t. $\Pi < \pi \leq \Pi + 1$. Let $\pi' = \pi - \theta_1$, where θ_1 is the parallelism of node 1.

Note that $\pi' \leq \Pi$.

Based on the weight w_1 of node 1, we have the following cases:

Case (a):

$$\frac{w_1}{\sum_{j=1}^{n} w_j} \leq \frac{\theta_1}{\pi} \tag{4}$$

By step 5 of the algorithm,

$$\phi_i = w_i, \forall i \in \{1, \ldots, n\}.$$

Hence,

$$\sum_{j=1}^{n} \phi_j = \sum_{j=1}^{n} w_j.$$

Case (b):

$$\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi} \tag{5}$$

By steps 2 and 3 of the algorithm,

$$\begin{aligned} \phi_1 &= \frac{\theta_1}{\pi'} \cdot \textstyle\sum_{j=2}^{n} \phi_j \\ \Rightarrow \textstyle\sum_{j=1}^{n} \phi_j &= \frac{\pi}{\pi'} \cdot \textstyle\sum_{j=2}^{n} \phi_j \end{aligned}$$

Also, by the inductive hypothesis,

$$\sum_{j=2}^{n} \phi_j \leq \sum_{j=2}^{n} w_j$$

Hence,

$$\sum_{j=1}^{n} \phi_j \leq \frac{\pi}{\pi'} \cdot \sum_{j=2}^{n} w_j \tag{6}$$

From Relation 5,

$$\begin{aligned} \frac{w_1}{\sum_{j=2}^{n} w_j} &> \frac{\theta_1}{\pi - \theta_1} \\ \Rightarrow w_1 &> \frac{\theta_1}{\pi - \theta_1} \cdot \textstyle\sum_{j=2}^{n} w_j \end{aligned}$$

Thus,

$$\sum_{j=1}^{n} w_j > \frac{\pi}{\pi'} \cdot \sum_{j=2}^{n} w_j \tag{7}$$

From Relations 6 and 7, we have,

$$\sum_{j=1}^{n} \phi_j < \sum_{j=1}^{n} w_j.$$

Hence, proved by induction.

■

Theorem 3: 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 they can utilize.

These properties can be restated formally as follows:

1) If $S = \{i | i \in \{1, \ldots, n\}, \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} < \frac{\theta_i}{\pi}\}$, then,
$$\phi_i = w_i, \forall i \in S.$$
2) If $\frac{w_i}{\sum_{j=1}^{n} w_j} > \frac{\theta_i}{\pi}$ for a node i, then,
$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} = \frac{\theta_i}{\pi}.$$

Proof:

1) Proof by induction on the number of CPUs (π):

Base Case: For $\pi \leq 1$, the property holds trivially, as,
$$[\phi_1, \ldots, \phi_n] = [w_1, \ldots, w_n].$$

Inductive Step: Suppose the property holds for all π upto Π.

Now consider π processors s.t. $\Pi < \pi \leq \Pi + 1$. Let $\pi' = \pi - \theta_1$, where θ_1 is the parallelism of node 1.

Note that $\pi' \leq \Pi$.

Based on the weight w_1 of node 1, we have the following cases:

Case (a):
$$\frac{w_1}{\sum_{j=1}^{n} w_j} \leq \frac{\theta_1}{\pi} \tag{8}$$

Again, the hypothesis holds trivially by step 5 of the algorithm.

Case (b):
$$\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi}$$

As shown in the proof for Theorem 2,
$$\frac{\phi_1}{\sum_{j=1}^{n} \phi_j} = \frac{\theta_1}{\pi}.$$

Hence, $1 \notin S$.

If $i \in \{2, \ldots, n\}$ s.t. $\frac{\phi_i}{\sum_{j=2}^{n} \phi_j} = \frac{\theta_i}{\pi'}$, then,

$$\begin{aligned} \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} &= \frac{\phi_i}{\frac{\pi}{\pi'} \sum_{j=2}^{n} \phi_j} \\ &= \frac{\pi'}{\pi} \cdot \frac{\theta_i}{\pi'} \\ &= \frac{\theta_i}{\pi} \end{aligned}$$

Hence, $i \notin S$.

Thus, $S = \{i | i \in \{2, \ldots, n\}, \frac{\phi_i}{\sum_{j=2}^{n} \phi_j} < \frac{\theta_i}{\pi'}\}$.

Therefore, $\phi_i = w_i, \forall i \in S$ by inductive hypothesis.

Hence, proved by induction.

2) Proof by contradiction.

Suppose for the node i,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \neq \frac{\theta_i}{\pi}.$$

Then, by Theorem 2,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} < \frac{\theta_i}{\pi}. \tag{9}$$

Relation 9 implies that the node i belongs to the set S defined above for Property 1. Hence, by Property 1,

$$\phi_i = w_i. \tag{10}$$

From Equation 10 and Lemma 2, we have

$$\begin{aligned} \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} &\geq \frac{w_i}{\sum_{j=1}^{n} w_j} \\ &> \frac{\theta_i}{\pi} \end{aligned}$$

This contradicts Relation 9.

Hence, proved by contradiction.

■

Corollary 0: Any pair of nodes that get fewer CPUs than their parallelism, get shares in proportion to their original weights.

Formally, if $\frac{\phi_i}{\sum_{l=1}^{n}\phi_l} < \frac{\theta_i}{\pi}$ and $\frac{\phi_j}{\sum_{l=1}^{n}\phi_l} < \frac{\theta_j}{\pi}$, for any $i, j \in \{1, \ldots, n\}$, then,

$$\frac{\phi_i}{\phi_j} = \frac{w_i}{w_j}.$$

Algorithm 2 hier_readjust(tree_node *node*)

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

Now, we analyze the time complexity of the generalized weight readjustment algorithm, and use it to analyze the time complexity of the hierarchical weight readjustment algorithm (Algorithm 2).

Theorem 4: The worst-case time complexity $T(n, \pi)$ of the generalized weight readjustment algorithm for n nodes and π processors is $O(\pi)$.

Proof:

If $\pi \leq 1$, then, the algorithm terminates at step 5, as all nodes satisfy the weight feasibility constraint in this case.

This means

$$T(n, \pi) = O(1), \text{if } 0 < \pi \leq 1.$$

Otherwise, the algorithm makes a recursive call at step 2 and terminates at step 3.

The recursive call is made with π set to $\pi - \theta_1$. Since, $\theta_1 \geq 1$, we have,

$$T(n, \pi) \leq T(n, \pi - 1) + O(1).$$

Solving the recurrence relation, we get,

$$T(n, \pi) = O(\pi).$$

■

We now present a lemma that allows us to extend the time complexity analysis of the generalized weight readjustment algorithm (that is used for a set of sibling nodes) to the time complexity analysis of the hierarchical weight readjustment algorithm (that is applied to the

whole scheduling tree). This lemma states that the total number of processors assigned to the nodes at each level of the tree remains constant.

Lemma 3: At any level l of the scheduling hierarchy,

$$\sum_{j \in S_l} \pi_j = p,$$

where, S_l is the set of nodes at l^{th} level of the tree, and p is the number of CPUs in the system.

Proof: Proof by induction on the tree level (l).

Base Case: At $l = 0$, i.e., at the root of the tree, the property holds by definition as $\pi_{root} = p$.

Inductive Step: Suppose the property holds for all levels l upto L.

Now consider level $L + 1$. Then, by the definition of processor assignment,

$$\begin{aligned} \textstyle\sum_{j \in S_{L+1}} \pi_j &= \textstyle\sum_{j \in S_{L+1}} \left(\frac{w_j}{\sum_{k \in C_{P(j)}} w_k} \cdot \pi_{P(j)} \right) && \text{where, } P(j) \text{ is the parent of node } j \\ &= \textstyle\sum_{i \in S_L} \sum_{j \in C_i} \left(\frac{w_j}{\sum_{k \in C_i} w_k} \cdot \pi_i \right) && \text{where, } C_i \text{ is the set of node } i\text{'s children} \\ &= \textstyle\sum_{i \in S_L} \pi_i \\ &= p && \text{by the inductive hypothesis} \end{aligned}$$

Hence, proved by induction.

■

Theorem 5: 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 $O(p \cdot h)$.
Proof: The time complexity of the hierarchical weight readjustment algorithm is determined by the time complexity of applying the generalized weight readjustment algorithm to all the nodes in the scheduling tree.

Thus, using Theorem 4, the time complexity of the hierarchical algorithm $T(n, h, p)$ is given by

$$\begin{aligned} T(n, h, p) &= \textstyle\sum_{i=1}^{n} O(\pi_i) && \text{where } \pi_i \text{ is the processor assignment of node } i \\ &= \textstyle\sum_{l=1}^{h-1} O(p) && \text{by Lemma 3} \\ &= O(p \cdot h) \end{aligned}$$

■

Thus, the time complexity of the hierarchical weight readjustment algorithm is dependent 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.

APPENDIX II
PROOF OF PROPERTIES OF HIERARCHICAL MULTIPROCESSOR SCHEDULING

In this appendix, we present the proof of the properties satisfied by H-SMP: the hierarchical multiprocessor scheduling algorithm. We consider a system model consisting of a fixed scheduling hierarchy, with no arrivals and departures of threads and no weight changes. Further, 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 non-trivial case, we would also assume that the number of threads $n \geq p$. In such a system model, H-SMP satisfies the following property.

Theorem 6: After every scheduling instant, for any node i in the scheduling tree, H-SMP ensures that

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

Proof: Proof by induction on the scheduling tree level (l).

Base Case: For $l = 0$, the property holds trivially, as $r_{root} = p = \pi_{root}$.

Inductive Step: Suppose the property holds for all nodes upto level L of the scheduling tree.

Now, consider a node P at level L of the tree with processor assignment and assignment equal to π_P and r_P respectively. Further, consider the set of P's children nodes C_P. These children nodes lie at level $L + 1$ of the tree.

Claim 1: No member of C_P is in the deficit set.

Proof of Claim 1: Let us assume that Claim 1 is false. Then, there exists a node $i \in C_P$ s.t. i is in the deficit set, i.e.,

$$r_i < \lfloor \pi_i \rfloor.$$

Now, for any set of sibling nodes, at any scheduling instant, H-SMP first schedules nodes that are in the deficit set, i.e., those nodes i for which $r_i < \lfloor \pi_i \rfloor$. And since the deficit set is non-empty by our assumption, no nodes could have been scheduled from the low-threshold set for C_P. Thus, the number of CPUs assigned to a node $j \in C_P, j \neq i$ would be

$$r_j \leq \lfloor \pi_j \rfloor.$$

Therefore, the total number of CPUs assigned to nodes in C_P is given by

$$\begin{aligned}\sum_{j\in C_P} r_j &< \sum_{j\in C_P} \lfloor \pi_j \rfloor \\ &< \lfloor \sum_{j\in C_P} \pi_j \rfloor \\ &< \lfloor \pi_P \rfloor \\ &< r_P, \text{(by the inductive hypothesis)}\end{aligned}$$

This contradicts the relation

$$r_P = \sum_{j\in C_P} r_j, \tag{11}$$

which holds by definition.

Therefore, Claim 1 is true.

Claim 2: No node $i \in C_P$ has $r_i > \lceil \pi_i \rceil$.

Proof of Claim 2: Let us assume that Claim 2 is false. Then, there exists a node $i \in C_P$ s.t.

$$r_i > \lceil \pi_i \rceil. \tag{12}$$

Now, for any set of sibling nodes, at any scheduling instant, H-SMP first schedules nodes that are in the deficit set. By Claim 1, the deficit set for C_P is empty. In that case, H-SMP schedules nodes that are in the low-threshold set, i.e., those nodes j for which

$$\lfloor \pi_j \rfloor = r_j < \lceil \pi_j \rceil.$$

The existence of node $i \in C_P$ satisfying Relation 12 implies that the low-threshold set is empty (otherwise these nodes would still be available for scheduling). This means that the number of CPUs assigned to a node $j \in C_P, j \neq i$ would be

$$r_j \geq \lceil \pi_j \rceil.$$

Therefore, the total number of CPUs assigned to nodes in C_P is given by

$$\begin{aligned}\sum_{j\in C_P} r_j &> \sum_{j\in C_P} \lceil \pi_j \rceil \\ &> \lceil \sum_{j\in C_P} \pi_j \rceil \\ &> \lceil \pi_P \rceil \\ &> r_P \text{ (by the inductive hypothesis)}\end{aligned}$$

This contradicts Equation 11.

Thus, Claim 2 is true.

By Claim 1,

$$r_i \geq \lfloor \pi_i \rfloor, \forall i \in C_P.$$

By Claim 2,

$$r_i \leq \lceil \pi_i \rceil, \forall i \in C_P.$$

Therefore,

$$\lfloor \pi_i \rfloor \leq r_i \leq \lceil \pi_i \rceil, \forall i \in C_P.$$

Now, since P is an arbitrary node at level L, we can assert that

$$\lfloor \pi_i \rfloor \leq r_i \leq \lceil \pi_i \rceil, \forall i \in S_{L+1},$$

where, S_{L+1} is the set of nodes at level $L+1$ of the scheduling tree.

Hence, proved by induction.

■

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

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