This paper was presented as part of the main technical program at IEEE INFOCOM 2011

A QoS Aware Multicore Hash Scheduler for Network Applications

Danhua Guo[1,2]
[1]Cisco Systems, Inc.
San Jose, California, USA
dannguo@cisco.com

Laxmi N. Bhuyan[2]
[2]Department of Computer Science and Engineering
University of California, Riverside
Riverside, California, USA
bhuyan@cs.ucr.edu

***Abstract*—As the line speed of the network evolves at an unprecedented rate, a wide spectrum of network applications call for increasing processing density on network devices. The prevalence of multicore chips ameliorates the stress on processing power, but the QoS guarantee is often ignored. In addition, results of legacy QoS studies are difficult to apply to multicore web servers. Therefore, a multicore scheduler that incorporates QoS concerns is missing. As the network development moves towards cloud computing, we see an increasing importance of QoS guarantees on high performance multicore network appliances.**

In this paper, we propose a proportional share hash based scheduler, PS-HRW, which extends existing optimizations in multicore scheduling with QoS concerns. We address the network QoS requirement by assigning weights to each connection following the classic General Processor Sharing (GPS) theory. Based on our previous multicore scheduling studies, PS-HRW allocates computing resources based on the QoS requirement, such that the workload is balanced at the packet level, and the connection locality is maintained. To provide accurate QoS guarantee, PS-HRW allocates an integral number of cores first and then allocates the residuals using a partitioning theory. However, different from traditional simulation based approach, we target at two popular applications on modern network appliances: Deep Packet Inspection (DPI) and multimedia transcoding. In addition, we generalize the topology of different multicore architectures into a communication matrix and optimize PS-HRW to incorporate cache awareness. Essentially, PS-HRW schedules incoming traffic efficiently by balancing between connection locality, load balancing, core/cache topology and QoS guarantees.

Keywords-Cache Locality; DPI; Load Balancing; Multicore;Multimedia;QoS;Scheduling

I. INTRODUCTION

The evolution of network infrastructure has disclosed high speed Internet at the rate of tens of hundreds of Gigabits per second, shifting the bottleneck of network performance to the processing end. To satisfy the increasing demand in processing power, multicore chips have become the *de facto* platform on modern network appliances. However, existing multicore scheduling optimizations usually sacrifices QoS guarantee for throughput [3, 4, 9, 10, 12, 13, 16, 20]. Such throughput-centric scheduling has long been accepted as the standard evaluation metric of system performance due to the traditional objective of high throughput processing in routers and switches. As the network development moves towards cloud computing, the centralized management of network recourses calls for an increasing concern over network QoS in accordance with users' subjective request. On the other hand, current research results in QoS studies [1, 2, 7, 15, 17, 18] are difficult to apply to multicore network devices in practice. This is because theoretical QoS requires perfect fairness based on different weights of each request. The QoS scheduling has to be implemented at a per packet rate in order to satisfy this condition. Such an implementation inevitably incurs a non-negligible scheduling overhead, and is impractical for real practice. This issue has been addressed by the recent Cisco UCS server blade [6]. An interesting question would be how to apply QoS awareness into multicore scheduling mechanisms and reuse existing research results for throughput-centric multicore scheduling studies.

In this paper, we propose a proportional share scheduler, PS-HRW, based on Highest Random Weight (HRW) and our previous studies on hash based scheduling [9, 10, 11, 12]. The PS-HRW consists of three steps. In the first step, PS-HRW calculates the proportional share of each service request, φ_i, based on the runqueue length of the requests. Use of runqueue length to reflect QoS performance has been widely adopted in related works [3, 12, 22, 23]. Then, it allocates an integral number of cores $\left\lfloor g_i = \frac{\varphi_i}{\sum_j \varphi_j} r \right\rfloor$ based on the QoS requirement of each request using the Adjusted Highest Random Weight (AHRW) scheduler proposed in [10]. We have shown that as a hash based scheduling mechanism, AHRW guarantees workload balancing at the packet level, while maintaining the connection locality at the same time [10]. In the third step, the residual request $g_i - \lfloor g_i \rfloor$ is allocated using a partitioning theory, originally designed to route requests to heterogeneous caches [21]. This theory generates a weight vector for AHRW that follows strictly to the capacity of each PU. AHRW then picks the proper core to host the residual requests. In addition, we generalize the core/cache topology of different multicore architectures into a communication matrix and propose a Hierarchical Cache Aware AHRW scheduler, H-CAHRW, to incorporate cache awareness. When we replace AHRW with H-CAHRW, the property of H-CAHRW guarantees that only

the core that balances connection locality, load balancing and cache/core topology is picked.

We implement PS-HRW on an Intel Xeon web server running L7-filter and FFmpeg applications and compare the performance with Round Robin (RR), Connection Locality scheduling [20] and adaptive partition scheduler [12]. We show that PS-HRW achieves the best balance among many performance aspects including system throughput, and maintains the QoS by reducing the out-of-order departure rate and delay jitter.

The rest of the paper is organized as follows: In Section II, we provide the background of classic studies in QoS scheduling. In Section III, we propose our PS-HRW based on AHRW. In Section IV, we propose H-CAHRW to optimize AHRW with cache awareness to enable deployment on hierarchical multicore architectures. In Section V, we introduce the two network applications, selected to evaluate our scheduling algorithm. In Section VI, we present our experimental results. In Section VII, we discuss related works in QoS scheduling and multicore scheduling. Lastly, in Section VIII, we conclude the paper and point out future research directions.

II. DEFINITION OF QOS SCHEDULING

Modern web servers should be able to provide QoS to each client that subscribes to the required service. Generally, the QoS requirements can be assessed in terms of users' subjective wishes or satisfaction with the quality of the application performance, cost, and so forth. The assessment results are then mapped onto measurable QoS parameters to which the router needs to guarantee. For example, the QoS parameters can be the message response time, the end-to-end delay or the out-of-order departure rate. When performing QoS aware message scheduling, the router needs to relate the resource consumed by each message type with its QoS requirements. Basically, we classify the service provided by a web server into several service classes, each corresponding to a specification of the resource requirement and the QoS parameter. The QoS aware scheduling algorithm aims to provide differentiated service as follows:

- **Fairness**. The system resource is allocated proportionally among the service classes.
- **Independent allocation**. Given sufficient incoming traffic, a service class receives at least as much resources as were assigned to it irrespective of the traffic of other service classes.
- **Work conserving**. Resources not used by some service class may be distributed among other service classes.

The term "fairness" has colloquial meanings. In QoS, we define "fairness" following the General Processor Sharing (GPS) theory [1, 2, 7, 15, 17, 18]. A GPS server is work conserving and operates at a fixed rate r. By work conserving, we mean that the server must be busy if there are packets waiting in the system. It is characterized by positive real numbers $\varphi_1, \varphi_2, \dots, \varphi_N$. At any given time, a flow is either backlogged or idle. Let $S_i(\tau, t)$ be the amount of session i traffic served in an interval $(\tau, t]$. A session is backlogged at time t if a positive amount of that session's traffic is queued at time t. The a GPS server is defined as one for which

$$\frac{S_i(\tau,t)}{S_j(\tau,t)} \geq \frac{\varphi_i}{\varphi_j}. j = 1,2, \dots N \quad (1)$$

for any session i that is continuously backlogged in the interval $(\tau, t]$.

Summing over all sessions j:

$$\sum_{j=1,2,\dots,N} S_j(\tau, t) = \frac{1}{(t-\tau)\cdot r}$$

$$S_j(\tau, t) \sum_j \varphi_j \geq (t-\tau)\cdot r \cdot \varphi_i$$

And session i is guaranteed a rate of

$$g_i = \frac{\varphi_i}{\sum_j \varphi_j} r \quad (2)$$

We propose a quantitative definition of the QoS requirements for L7-filter and FFmpeg. Previous works [3, 12, 22, 23] have shown that the computation time of each request directly influences the QoS performance of network applications. In L7-filter, connection buffers are feed into the classification engine on a per byte basis for the state machine, the computation time is linear to the size of the connection buffer. In FFmpeg, the transcoding process is also strictly proportional to the size of the message size. Therefore, we can further simplify the QoS requirement for both applications to the size of runqueue in terms of byte for each processing request.

In the HRW scheduling case, let $\varphi_1, \varphi_2, \dots, \varphi_N$ be the runqueue size for each request, and r be the overall number of cores on a server. Then following GPS, each request should be assigned $g_i = \frac{\varphi_i}{\sum_j \varphi_j} r$ number of cores.

In [12], we see an Adaptive Partitioning (AP) algorithm that satisfy this constraint in a cluster of servers. However, that algorithm allocates the proportional share of clusters in a round robin fashion, which fails to consider core/cache topology and load balancing in a multicore system. In addition, the system scheduler handles the fractional allocation in that paper automatically. Therefore, the partitioning algorithm is oblivious to the scheduling decisions for the fractional part. Based on our previous studies in HRW schedulers, we know that HRW can achieve high system throughput by satisfying connection affinity, load balancing and core/cache topology. Our problem now is how to incorporate QoS concerns into HRW.

III. PS-HRW

Objective: Given a service request C_i, with weight φ_i, overall system service rate r, the allocated system resource g_i should be proportional to its weight assignment, $g_i = \frac{\varphi_i}{\sum_j \varphi_j} r$. Meanwhile, system throughput, load balancing, core/cache topology and scheduling overhead should be balanced.

Solution:

We propose a take a 3-step solution to achieve the objective. In the first step, PS-HRW calculates the weight of each service request, φ_i, based on the runqueue length of the request. Then, it allocates an integral number of cores $\left\lfloor g_i = \frac{\varphi_i}{\sum_j \varphi_j} r \right\rfloor$ based on AHRW. In the third step, the residual request $g_i - \lfloor g_i \rfloor$ is allocated using a partitioning theory for routing requests to heterogeneous caches [21]. This theory generates a weight vector for AHRW that follows strictly to the capacity of each PU. The property of AHRW guarantees that only the core that balances connection locality, packet level load balancing is picked.

Note that the choice of scheduler in step 2 and 3 are arbitrary. In Section IV, we propose an optimization to AHRW to consider core/cache topology of different multicore servers, further improving system performance.

A. Weight Calculation

For both L7-filter and FFmpeg, we can obtain the size of each request in terms of bytes by measuring the length of the connection buffer and stream size. The desired weight for request i at time t, $\varphi_i(t)$, can be calculated based on Eq. 3.

$$\varphi_i(t) = size_i(t) \tag{3}$$

Then the proportional allocation of cores for request i, g_i, can be calculated based on Eq. 4.

$$g_i(t) = \frac{\varphi_i(t)}{\sum_{j=1}^{M} \varphi_i(t)} \cdot r = \frac{size_i(t)}{\sum_{j=1}^{M} size_j(t)} \cdot r \tag{4}$$

where M is the number of concurrent requests being processed in the system and r is the number of cores.

B. Integral Core Allocation

For each $g_i(t)$, we use AHRW to allocate the greatest integer number $\lfloor g_i(t) \rfloor$ of cores that is smaller than the required cores $g_i(t)$. Since $g_i(t)$ can be greater than 1, the algorithm will recursively call AHRW to allocate cores one at a time, until the residual value is smaller than 1 ($g_i(t) - \lfloor g_i(t) \rfloor$).

C. Residual Core Allocation

In this step, we use a partitioning theory for routing requests to heterogeneous caches [21]. We can apply that theory to our packet scheduler, where incoming packets function as URL requests and the cores function as caches. Let $p_1, \dots, p_N$ be given arbitrary target probabilities for each of all the N cores. If a core has target probability p_i we desire the fraction p_i of requests to be mapped to it.

In the robust hashing scheme, for a given request $\vec{c}$ we calculate a hash value $h_i = h(\vec{c}, p_i)$ based on AHRW for each of core i. We then schedule the request to the core that has the highest h. This scheme will schedule 1/N of the connection buffers to each core. To deal with target capacities, we introduce multipliers $x_1, \dots, x_N$ and multiply each h_i with the respective x_i , we then map the request to the core that has the largest $Z_i = x_i \cdot h_i$ value. If the multipliers are different, the fractions of requests scheduled to the core will no longer all be the same. Specifically, we want to assign all the cores that has been used in step 2 with the capacity value $1/N'$ and the rest of the cores with $\frac{1}{N'}(g_i(t) - \lfloor g_i(t) \rfloor)$. (Suppose c cores have been allocated in step 2, $N' = c + (N - c) \cdot (g_i(t) - \lfloor g_i(t) \rfloor)$). The intuition behind this algorithm is to use AHRW and the heterogeneous partitioning theory to pick the residual proportional share of cores that balances between connection locality, load balancing and core/cache topology.

Theorem 1 - Let $p_1, \dots, p_N$ be given target capacities for each core. Reorder the cores so that $p_1 \le \cdots \le p_N$. Let $x_1 = (N \cdot p_1)^{1/N}$ and let $x_2, \dots, x_N$ be recursively calculated as follows: $x_n = \left[\frac{(N-n+1)(p_n - p_{n-1})}{\prod_{i=1}^{n-1} x_i} + x_{n-1}^{N-n+1} \right]^{\frac{1}{N-n+1}}$. Then the robust hash algorithm, AHRW with multipliers $x_1, \dots, x_N$ will be allocate the fraction p_i of requests to the i^{th} core for $i = 1, \dots, N$.

Proof - Let $x_1, \dots, x_N$ be an arbitrary set of nonnegative multipliers satisfying $x_1 \le x_2 \dots \le x_N$. Let $h_1, \dots, h_N$ be the H-CAHRW hash values associated with each of the N cores. Because the outputs of a hash function can be taken to be independent, uniformly distributed random variables, without loss of generality, we take each h_i to be uniformly distributed over $[0,1]$. Let $Z_i = x_i h_i$ be the i^{th} multiplied hash value. Note that the Z_is are independent and that Z_i is uniformly distributed over $[0, x_n]$. Let $Z_{(i)} = \max(Z_1, \dots, Z_{i-1}, Z_{i+1}, \dots, Z_N)$. Let q_i be the probability that core i has the largest multiplied hash value, that is, $q_i = P(Z_{(i)} \le Z_i)$. Conditioning on $Z_i = x$, we obtain

$$q_i = P\big(Z_{(i)} \le Z_i\big) = \frac{1}{x_i}\int_0^{x_i} P\big(Z_{(i)} \le x\big)dx = \frac{1}{x_i}\int_0^{x_i} \prod_{j \ne i} P\big(Z_j \le x\big)\,dx = \frac{1}{x_i}\int_0^{x_i} \frac{\prod_{j=1}^{N} P(Z_j \le x)}{P(Z_i \le x)}dx = \int_0^{x_i} \frac{1}{x}\prod_{i=1}^{N} P(Z_i \le x)dx = \sum_{j=1}^{N}\int_{x_{j-1}}^{x_j} \frac{1}{x}\prod_{k=1}^{i} P(Z_k \le x)dx\,, \tag{5}$$

where $x_0 = 0$. We must now get an explicit expression for the product in the above expression.

For $x_{j-1} \le x \le x_j$,

$$P(Z_i \le x) = \begin{cases} 1, & i \le j-1 \\ \dfrac{x}{x_j}, & i \ge j \end{cases}$$

Thus, for $x_{j-1} \le x \le x_j$,

$$\prod_{i=1}^{N} P(Z_i \le x) = \Big[\prod_{i=1}^{j-1} P(Z_i \le x)\Big]\Big[\prod_{i=j}^{N} P(Z_i \le x)\Big] = \prod_{i=j}^{N} \frac{x}{x_j} = \frac{x^{N-j+1}}{\prod_{i=j}^{N} x_i} \tag{6}$$

Insert Eq. 5 into Eq. 6 gives

$$\begin{aligned} q_i &= \sum_{j=1}^{N} \int_{x_{j-1}}^{x_j} \frac{x^{N-j}}{\prod_{i=j}^{N} x_i} dx \\ &= \sum_{j=1}^{i} \frac{1}{\prod_{k=j}^{N} x_k} \frac{1}{N-j+1}\left(x_j^{N-j+1} - x_{j-1}^{N-j+1}\right) \\ &= \left(\prod_{k=1}^{N} x_k\right)^{-1} \sum_{j=1}^{i} \frac{\left(\prod_{k=1}^{j-1} x_k\right)\left(x_j^{N-j+1} - x_{j-1}^{N-j+1}\right)}{N-j+1} \end{aligned}$$

We have one degree of freedom. Set $\prod_{k=1}^{N} x_k = 1$. Then

$$q_i = \sum_{j=1}^{i} \frac{(\prod_{k=1}^{j-1} x_k)\left(x_j^{N-j+1} - x_{j-1}^{N-j+1}\right)}{N-j+1} \quad (7)$$

From Eq. 7 we have

$$q_i = q_{i-1} + \frac{(\prod_{j=1}^{i-1} x_j)\left(x_i^{N-i+1} - x_{i-1}^{N-i+1}\right)}{N-i+1} \quad (8)$$

The desired result follows by setting $q_i = p_i, for\ i = 1, \dots, N$,and solving for x_i in Eq. 8.

End of Proof

The robust hash function with multipliers $x_1, \dots, x_N$ in the above theorem is part of CARP [21]. Our algorithm is shown in Table 1.

Discussion

Because the computation in a QoS scheduler is intensive, it can only be calculated and updated at a realistic rate. An additional data structure is required to register the current connection-to-core mapping.

We only apply this algorithm periodically, rather than on a per packet basis. Only when the connection buffer size of a connection φ_i varies by more than a tolerable percentage, say β, should we recalculate the mapping. We choose different value of β to update the weight vector $(\varphi_1, \varphi_2, \dots, \varphi_m)$. A greater value of β reduces the recomputation but loses scheduling accuracy. We also force the scheduler to update the weight vector $\varphi(\varphi_1, \varphi_2, \dots, \varphi_m)$ any time when a connection finishes processing or a new connection arrives at the system.

IV. Hierarchical Cache Aware HRW Scheduler

In the previous section, we used AHRW in the PS-HRW algorithm to allocate cores to connections based on the weight assignment. While AHRW adjusts the weight of HRW to achieve load balancing at the packet level, it fails to take core/cache topology into consideration, which becomes more important in modern multicore chips. Therefore, we introduce H-CAHRW to incorporate cache awareness in the hash function and generalize a hierarchical scheduler for multicore architectures possessing different core/cache topologies.

A. Cache Aware AHRW Scheduler (CA-AHRW)

Since the communication overhead between cores in multicore architectures is heterogeneous, especially with hierarchical memory structure, we should distinguish the inter-core relationship by applying weighted queue lengths in the AHRW hash function.

In Fig. 1, we have three different communication overhead ratio $w_1, w_2$and w_3. Given these parameters, we can construct a communication matrix to represent the inter-core relationship. Suppose the total number of nodes (PUs) is M, then the $M \times M$ matrix is formed by filling the position (i, j). Obviously, this is a symmetric matrix with the diagonal elements being all 1. We further define an adjustment vector for each node as follows: for node i, its adjustment vector is the i^{th} row vector in the communication matrix. Fig. 1 shows the communication matrix and the derived adjustment vector for each node.

Table 1 PS-HRW Algorithm

Input: $\varphi_i, i = 1, \dots, N$

$\overrightarrow{c_i}$, i = 1, … , N

Output: $setc_i$, i = 1, … , N

Algorithm:

for each $i = 1, \dots, N$

if φ_i varies by more than β **then**

$setc_i$={} // Step 1

$g_i(t) = \frac{\varphi_i(t)}{\sum_{j=1}^{M} \varphi_i(t)} \cdot r = \frac{size_i(t)}{\sum_{j=1}^{M} size_j(t)} \cdot r$

while $g_i(t) > 0$ **then** // Step 2

$setc_i = set_i \cup AHRW(\overrightarrow{c_i})$

$g_i(t) -= 1$

// Step 3

for each core j

if $j \in setc_i$ **then**

$p_j = 1/N'$

else

$p_j = \frac{1}{N'}(g_i(t) + 1)$

Recursively calculate vector $X = (x_1, \dots, x_N)$ using Theorem 1

reset $h(\overrightarrow{c_i})$ to $X \cdot h(\overrightarrow{c_i})$ in AHRW

$setc_i = setc_i \cup AHRW'(\overrightarrow{c_i})$

Now we address how to derive the weighted queue length for each node. Suppose we have an original queue vector (Q_1, Q_2, Q_3, Q_4), which records the real queue length. For each node, by multiplying this queue vector with the adjustment vector from this node, we can obtain the weighted queue vector. Here, we define a new vector multiplication operation following Eq. 9.

$$\vec{A} \otimes \vec{B} = \vec{C}, where\ c_i = a_i \cdot b_i \quad (9)$$

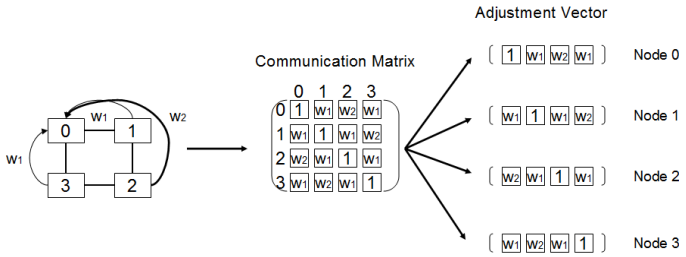

Fig. 1: An example of communication matrix and adjustment vector from a multicore architecture.

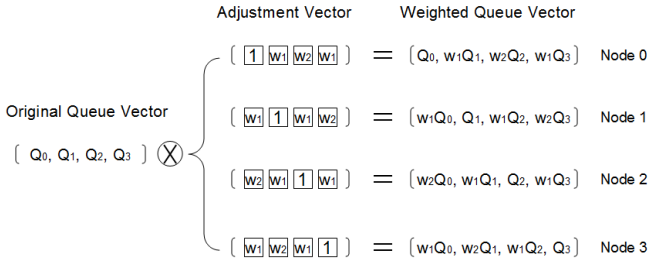

Fig. 2: Weighted queue vector derivation from the original queue vector and the adjustment vector.

Suppose $\vec{A}$ is the original queue vector, $\vec{B}$ is the adjustment vector and $\vec{C}$ is the weighted queue vector. For instance, as shown in Fig. 2, node 1 obtains the weighted queue vector $(w_1Q_0, Q_1, w_1Q_2, w_2Q_3)$, which reflects the core/cache topology and communication heterogeneity in the multicore architecture.

Based on the weighted queue length, we present the CA-AHRW hash function in Eq. 10 and the generalized CA-AHRW hash scheduler in Eq. 11. We define the "homenode" for a flow as the scheduled node by the original HRW hash scheduler, which is responsible for preserving packet departure order and is determined only by the identifier vector $\vec{c}$. Notice that Q'_p is the weighted queue length on node p chosen from the weighted queue vector for the homenode of $\vec{c}$ in Eq. 9.

$$h'(\vec{c}, p, w_p) = w_p \cdot g(\vec{c}, p) = \frac{Q_{min}}{Q'_p} \cdot g(\vec{c}, p) \qquad (10)$$

$$f'(\vec{c}) = p \Leftrightarrow h(\vec{c}, p, w_p) = \max_{k\in[1,N]} h'(\vec{c}, k, w_k) \quad (11)$$

Our generalized CA-AHRW hash scheduler consists of the following four steps:

- Step 1: Construct the communication matrix for the targeting multicore architecture and derive the adjustment vector for each node.
- Step 2: For a given packet of a connection whose homenode is i, obtain the current weighted queue vector for node i.
- Step 3: Apply the weighted queue length in CA-AHRW hash function following Eq. 10.
- Step 4: Choose the node with the maximum weight resulted from the hash function $h'(\vec{c}, k, w_k)$ as the scheduling node according to Eq. 11.

B. Hierarchical CA-AHRW (H-CAHRW) Scheduler

Given a hierarchical multicore architecture, we can apply our CA-AHRW hash scheduler recursively along the traversal of the tree, except that we replace the weighted queue length with the queue length obtained according to Eq. 11 for all non-leaf nodes. For nodes at the same depth of the tree, we can pick an internal node by the CA-AHRW hash scheduler and continue the traversal from that chosen node. The ultimate goal of the hash-tree scheduler is to select a candidate node among the leaf nodes by traversing through the three as shown in Eq. 12,

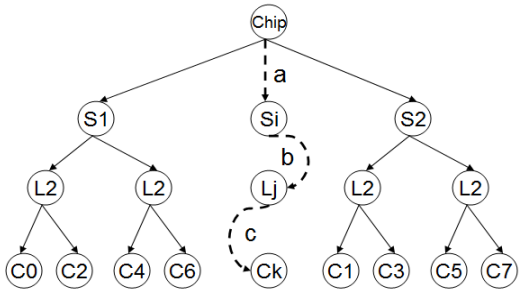

Fig. 3: Hash-tree scheduling process on Intel Xeon E5335 server. The dashed arrows represent a scheduler-selected path: a→b→c.

assuming the tree height is L with the root being level 0 and the leaf being level L.

$$f_{tree}(\vec{c}) = p_n \Leftrightarrow \begin{cases} h'(\vec{c}, p_1, w_1) = \max h'(\vec{c}, k_1, w_{k_1}), k_1 \in children\ of\ root \\ h'(\vec{c}, p_2, w_2) = \max h'(\vec{c}, k_2, w_{k_2}), k_2 \in children\ of\ P_1 \\ \ldots \\ h'(\vec{c}, p_L, w_L) = \max h'(\vec{c}, k_L, w_{k_L}), k_L \in children\ of\ P_{n-1} \end{cases} \qquad (12)$$

Fig. 3 shows the deployment of H-CAHRW on an Intel Xeon E5355 based web server. For any given packet, the H-CAHRW scheduler first picks a socket S_i with the maximum weight generated by the CA-AHRW hash function at the socket level (path a). Then we apply the CA-AHRW hash function at the L2 cache level for the selected socket S_i, and pick a L2 cache L_j (path b). Finally, at the core level, CA-AHRW picks the desired core C_k from the selected L2 cache L_j (path c). The properties of connection locality, load balancing and cache-awareness hold true at each level in the tree because the corresponding hash functions at each level are CA-AHRW. In addition, H-CAHRW also provides the following properties.

- **Reduced computation cost**. Suppose the complexity of the original HRW hash function and the adjustment multiplier computation is H. The complexity of CA-AHRW hash scheduler with flat-level linear search is $O(H \cdot N \cdot M)$ for N nodes and M scheduling units. On the other hand, the hash-tree scheduler selects a node by tree traversal. Thus, with the same denotation, the complexity is reduced to $O(H \cdot lgN \cdot M)$. In general, the scheduling overhead is reduced from N to lgN under the same workload and platform.
- **More effective load balancing**. Note that while the hash space remains in the range of $[0,2^{31}-1]$, H-CAHRW reduces the search space from N to lgN, which leads to faster computation for load balancing at each level. In addition, the hash-tree scheduler essentially uses multiple hashes per input key. This behavior not only reduces the possibility of hash collision, but also progressively improves load balancing for the overall system.

Optimized PS-HRW: Replacing the AHRW scheduler in the algorithm described in Table 1 with H-CAHRW, we obtain the optimized Proportional Share H-CAHRW Scheduler, PS-HRW.

V. Selected Network Applications

In this section, we introduce the applications we chose to test our algorithm. A common feature in both application is the hierarchy of "connection" and "packet", which suits our scheduling algorithm.

A. L7-filter

Fig. 4 illustrates the structure of a Linux networking system with L7-filter [11, 13, 16] in an OSI-model context. It sits on

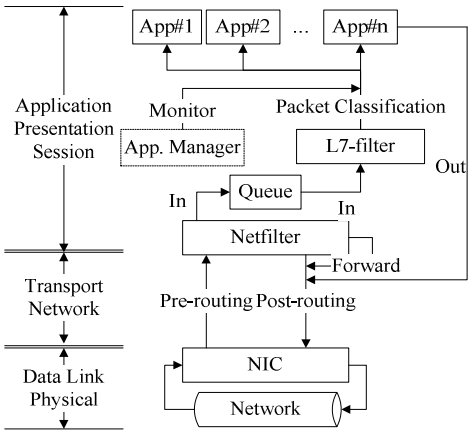

Fig. 4: L7-filter in the OSI Model.

top of the transport layer, monitoring network traffic based on packet payloads. While Netfilter relies on *iptables* to accept/forward/drop incoming packets, L7-filter marks all the accepted packets with their protocol IDs. Potential process/application managers can easily pick up the packet protocol IDs and reshape the traffic for security and management concerns. Cisco's Online Monitoring Engine [5] is an example of such a process/application manager which has been widely adopted in Wall Street firms to provide simultaneous customer feedback based on traffic requests.

In L7-filter, network traffic is classified in the form of "connection buffer". Each connection buffer is made up of up to eight packets of the same connection. Connection buffers are managed in a table, i.e. reassembling buffer, with each distinguished connection registered in a separate entry. Every time a new packet arrives at the system, it is appended to the corresponding buffer entry. The newly reassembled buffer of the current connection is then sent to the matching engine to match against a predefined protocol data set.

Note that due to the NO_MATCH_YET case, there could be multiple connection buffers for the same connection existing in the system at any given time. This is not a problem when the L7-filter program is running on a single core server because the program will be executed sequentially. Several changes to the L7 filter algorithm are necessary for its efficient execution on multi-core processors.

B. FFmpeg

FFmpeg is a powerful multimedia processing tool that can resolve the format discrepancy between multimedia files. The original movie is first decoded into raw frames by appropriate decoder (e.g., MPEG-1). Meanwhile, given the transcoding requirement for this movie in terms of video size, frame rate and bit rate, the FFmpeg controller will specify those parameters used by the encoder. Then, the encoder will start transcoding accordingly.

Fig. 5 illustrates the overview of the scheduling process in FFmpeg transcoding [8]. All the incoming streams are first stored in a global queue in the FIFO order. In the context of

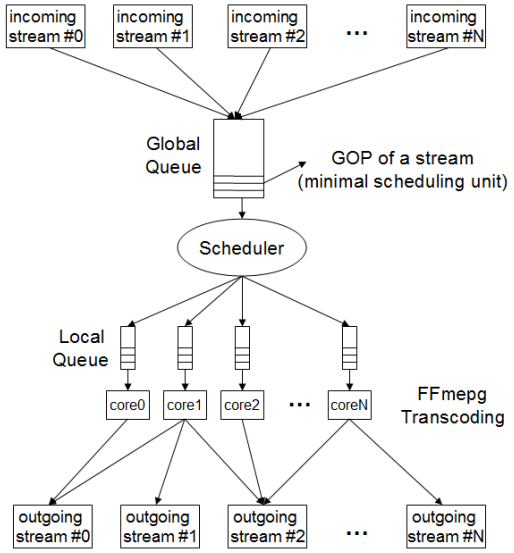

Fig. 5: Overview of the scheduling process in FFmpeg

transcoding, each stream consists of many Groups of Pictures (GOPs), which are the minimal scheduling units and can be scheduled to any core independently. The relationship between stream and GOP can be equivalently understood as that of flow and packet. For each scheduling cycle, the scheduler fetches a GOP from the FIFO global queue, makes the scheduling decision based on the scheduler, and then dispatches the GOP into the local queue of the scheduled core. Compared to L7-filter, FFmpeg has the same hierarchy- we just need to substitute "packet" with "GOP, and "connection" with "stream". Each core runs a transcoding thread, which iteratively fetches a GOP from its local queue and executes the task.

VI. EXPERIMENTAL RESULTS

We implement and evaluate our scheduler on the Intel server, residing two Quad-core Xeon E5355 processors with 2.66 GHz frequency. Each core has a 128KB dedicated L1 cache. Two cores share a 4M L2 and four cores share a socket with 8M L2 cache. We compare our results with RR,

Table 2: Key Features of the Trace Files

Trace Name	# of Pkt.	# of Conn.	Conn. Length	Trace Size
MIT	340K	40K	8.5	286MB
TU	1.32M	110K	12	1GB
NYP	590K	61K	9.6	500MB

Table 3: Resolution Criteria in MPEG Specification and Original Features for Selected Movies

Rate	Frame Size	Frame Rate (fps)	Bit Rate (kbps)
SQCIF	128*96	15	50
QCIF	176*144	15	70
CIF	352*288	26	100
4CIF	704*576	30	200
Original	352*240	29097	920

SM/connection based scheduling [10], Adaptive Partitioning (AP) scheduling [12]. In addition, we also develop a QoS scheduler based purely on HRW, GPS-HRW, instead of H-CAHRW. The performance comparison between GPS-HRW and PS-HRW illustrate the benefits of cache awareness and load balancing on top of QoS in scheduling. We apply the same trace-driven model and trace files for L7-filter described in a previous paper [10].

For L7-filter experiments, we adopt the trace driven model proposed in [10]. Table 2 lists three different trace files we used in our experiments. For FFmpeg experiments, we insert different resolution criteria (Table 3) into the original movie files. On average, there are 20 concurrent streams following measurement results in [12].

A. Selection of Scheduling Frequency

In Fig 6, the impact of scheduling update frequency (β) is illustrated. A greater value of β reduces the accuracy of the QoS, because it requires less update to the scheduler and hence less remapping of the service requests to the core. However, it also reduces the system overhead by triggering less computation. The result shows that a value of 16% gives the best result of performance-overhead trade off. For the rest of the experiments, we choose this value as the parameter.

B. System Throughput

Fig. 7 and 8 show the system throughput of PS-HRW compared to previously proposed schedulers. The system throughput for FFmpeg is measured by GOPs per second. We observe a minimum of 10% degradation in system throughput using GPS-HRW compared to HRW scheduler for both L7-filter and FFmpeg. In addition, GPS-HRW, although more effective than AP, provides less throughput than PS-HRW and HRW based schedulers. On the other hand, PS-HRW improves the system throughput by 11% due to the heuristics in H-CAHRW. Therefore, we can see the importance of core/cache topology and packet level load balancing in the scheduler.

C. Load Balancing

Fig. 9 and Fig. 10 visualize the load balancing results by the range of CPU utilization for each scheduler. The cross ("X") represents the average CPU utilization. We make three observations from these two figures. First, hash based schedulers expose better load balancing than pure connection locality based schemes. PS-HRW performs the best, which is in line with the corresponding system throughput observed in Fig. 7 and 8. Second, AP and PS-HRW achieves better balancing due to the packet level adjustment. Third, compared to PS-HRW, GPS-HRW incurs a higher degree of imbalance. This observation shows that H-CAHRW is more efficient than the original HRW scheduler.

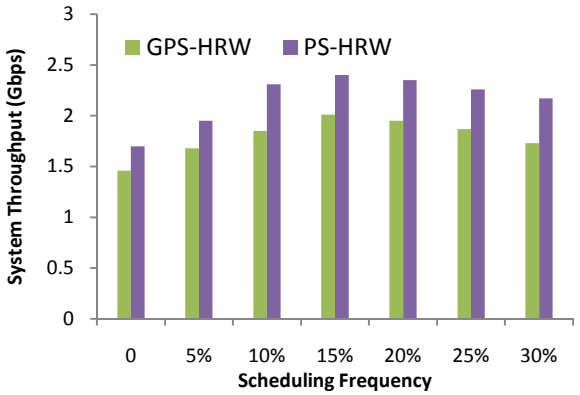

Fig. 6: Impact of scheduling frequency β on system throughput

D. Scheduling Overhead

As one of the major drawbacks of QoS based scheduling, the overhead incurred by the additional computation is a key concern for our design. Because PS-HRW is applied on a connection granularity, it is expected to have a lower overhead compared to other schedulers that schedule on a per packet basis. On the other hand, the heuristic computation for QoS requirements are more complicated than throughput-centric schedulers. We measured the overhead on a per packet basis. Fig. 11 and 12 show the overhead of different schedulers. As expected, both GPS-HRW and PS-HRW incurs additional overhead compared to their baseline scheduler. Because of the calculation used to satisfy the QoS requirements, such overhead is inevitable. However, we also find that this overhead takes less than 9% of the overall system execution time for L7-filter and less than 1% for FFmpeg.

E. Cache Misses

In Fig. 13, we compare the cache performance in terms of L2 data cache miss rate measured by PAPI-3.7.0 [19]. Our first observation is that AP incurs the highest cache misses due to the obliviousness of cache/core locality. This is the most important drawback of most previous works in QoS based scheduling. Because core/cache topology becomes increasingly important in modern multicore chips, the efficient exploration of the cache hierarchy can significantly improve system throughput. Both "conn" and "HRW" cause less cache misses because of the strict connection locality constraint. PS-HRW inherits the cache awareness from H-CAHRW, and incurs the least cache misses.

F. QoS Quality

Lastly, we measure the video quality by the out-of-order departure rate and delay jitter. The former metric describes how many GOPs/connection buffers in a stream depart out of order on average, while the latter is defined the standard deviation of the interdeparture time among GOPs/connection buffers. High jitter is detrimental to the playback quality for FFmpeg and wasting computing resources for L7-filter, which is the main concern of media clients. The results are illustrated Fig. 14. These metrics reflect the playing quality of streams. It is clear that the two metrics follow the same pattern, where RR suffers the most due to its random distribution of scheduling units without stream locality, which results in large number of out-of-order scheduling units, and high jitter. On the contrary, SM is able to maintain a very good video quality by stream locality, although it has throughput and load imbalance deficiency illustrated in Fig. 7-10. On the other hand, despite the throughput advantage using AP, we observe that this group of schedulers sacrifices video quality. Therefore, it is not the

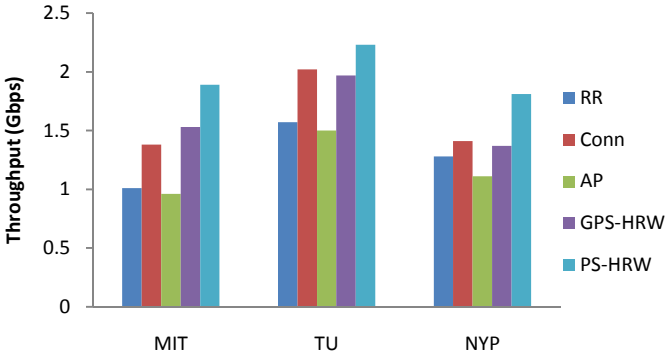

Fig. 7: System throughput of L7-filter for 3 trace files

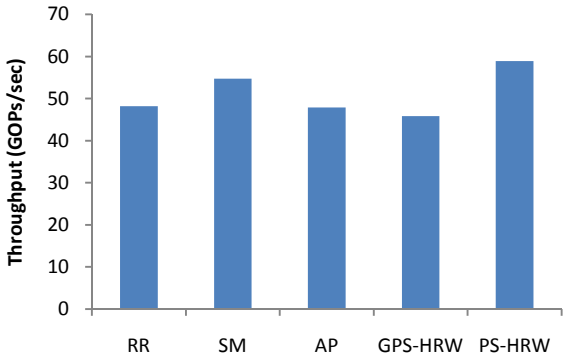

Fig. 8: System throughput of FFmpeg

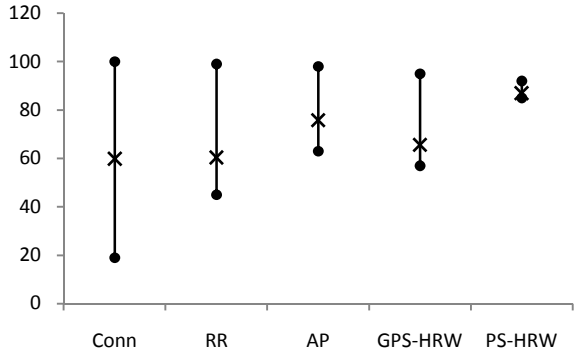

Fig. 9: Load balancing of L7-filter

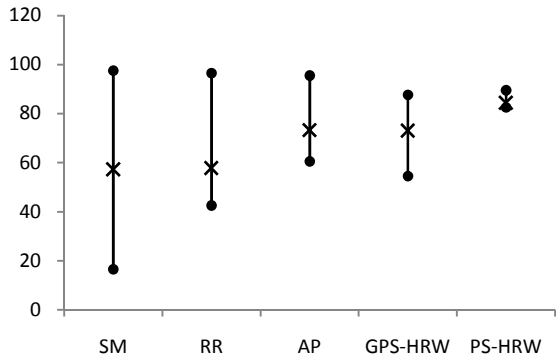

Fig. 10: Load balancing of FFmpeg

best choice when QoS requirement is the first performance priority. PS-HRW, strengthening both throughput and video quality, stands out among all other schedulers.

VII. Related Work

A. QoS Scheduling

QoS scheduling allocates a proportional share of the processing resources to each process according to the weight of the process. General Processor Sharing (GPS) [1, 7, 15, 17] was a theoretically ideal scheduler for single processors that provides QoS guarantee. Later, GPS scheduling was extended to multiple links [2, 18]. While GPS and its extension provide theoretical QoS guarantee to ensure fairness in multicore scheduling based on the weight of each process, it is impractical to implement in real systems due to the complicated computation requirement. In addition, QoS guarantee usually sacrifices connection locality, load balancing and core topology [3, 12]. We have not seen any previous studies that consider this trade off.

B. Multicore Scheduling

In order to explore the increasing processing density on multicore chips, a wealth of research has been reported. Connection locality based scheduling has been proposed and implemented from both academia [9] and industry [20] for its simplicity. An orthogonal direction of research focuses on balancing workload across all the cores [4]. We have previously proposed AHRW [9] to strike a balance between these two important factors. However, these works target at improving system throughput as the sole performance metric without considering QoS requirement.

VIII. Conclusion and Future Work

In this paper, we proposed a Proportional Share HRW based scheduler, PS-HRW. This scheduler balances system performance by provisioning QoS guarantees. Compared with previous works, PS-HRW also inherits connection locality, packet level load balancing, core/cache topology from H-CAHRW. We implemented PS-HRW on a real web server and the results show that PS-HRW provides the best QoS guarantee in terms of out-of-order departure rate and delay jitter. Meanwhile, it provides comparable system throughput compared to existing throughput-centric schedulers. In addition, we also showed that PS-HRW incurs only an additional 2% of scheduling overhead compared to the hash based heuristic scheduling, and causes less last level cache misses, which becomes increasingly important in multicore development.

In the future, we plan to deploy the proposed algorithm on Cisco's UCS blade on both the physical devices and the virtualized router and switches. We believe the trend of QoS aware high performance scheduling has a great future in the era of cloud computing.

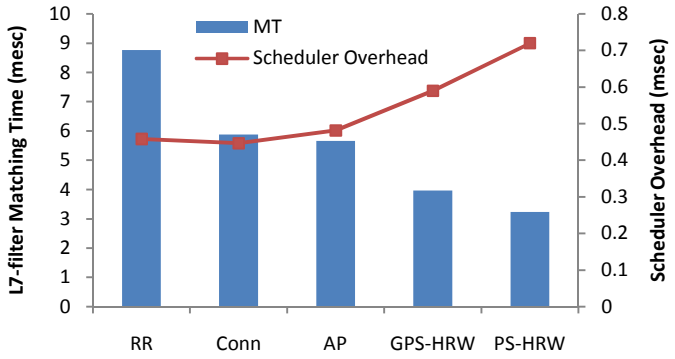

Fig. 11: Scheduling overhead for L7-filter

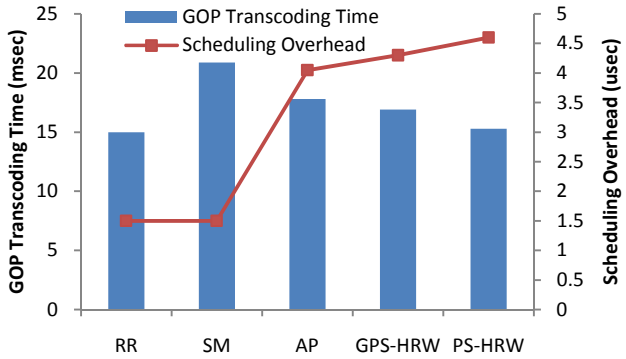

Fig. 12: Scheduling overhead for FFmpeg

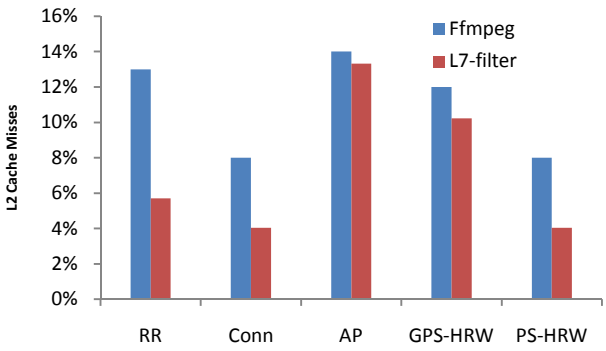

Fig. 13: L2 Cache Miss Rate

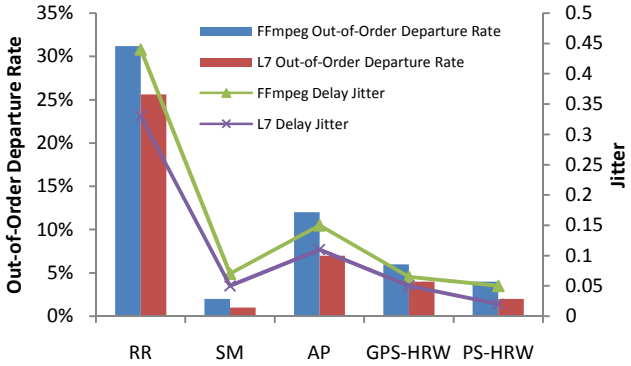

Fig. 14: Out-of-Order Departure and Jitter

ACKNOWLEDGEMENT

The research was supported by NSF grants CNS-0832108 and CSR-0912850.

REFERENCE

[1] J. C. R. Bennett and H. Zhang. WF2Q: Worst-case Fair Weighted Fair Queuing. In Proceedings of the IEEE INFOCOM, San Francisco, March 1996.

[2] J. Blanquer and B. Ozden, Fair queuing for aggregated multiple links, SIGCOMM 2001.

[3] A. Chandra, and P. Shenoy, Hierarchical Scheduling for Symmetric Multiprocessors, IEEE Transaction on Parallel and Distributed Systems, Vol. 19, No. 3, March 2008.

[4] Ho-Lin Chen, et al., On the impact of heterogeneity and back-end scheduling in load balance designs, INFOCOM, 2009.

[5] Cisco SCE 2000 Series Service Control Engine, http://www.cisco.com/en/US/products/ps6151/.

[6] Cisco Unified Computing System, http://www.cisco.com/en/US/netsol/ns976/index.html.

[7] A. Demers, S. Keshav, and S. Shenkar, Analysis and simulation of a fair queuing algorithm, Internet. Res. and Exper., vol. 1, 1990.

[8] FFmpeg, http://ffmpeg.org/.

[9] D. Guo, G. Liao, L. Bhuyan, B. Liu and J. Ding, A scalable multithreaded L7-filter design for multicore servers, ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2008.

[10] D. Guo, G. Liao, L. Bhuyan and B. Liu, An adaptive hash-based multilayer scheduler for L7-filter on a highly threaded hierarchical multicore server, ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2009.

[11] D. Guo and Laxmi Bhuyan, Multithreaded L7-filter for Linux and Multicore Schedulers, http://www.cs.ucr.edu/~dguo/l7.htm.

[12] J. Guo and L. Bhuyan, Load balancing in a cluster-based web server for multimedia applications, IEEE Transactions on Parallel and Distributed Systems, No. 11, Nov. 2006.

[13] N. Hua, H. Song and T. Lakshman, Variable-stride multi-pattern matching for scalable deep packet inspection, IEEE INFOCOM, 2009.

[14] Lukas Kencl, Jean-Yves Le Boudec, Adaptive load sharing for network processors, IEEE Transactions on Networking, No. 2, April 2008.

[15] L. Kleinrock, Queueing SystemVol. 2: Computer applications, New York: Wiley, 1976.

[16] S. Kumar, et al., Algorithms to accelerate multiple regular expressions matching for deep packet inspection, SIGCOMM, 2006.

[17] A. K. Parekh and R. G. Gallager, A generalized processor sharing approach to flow control in integrated services networks: the single-node case, IEEE/ACM Transaction on Networking, Vol. 1, No. 3, June 1993.

[18] A. K. Parekh and R. G. Gallager, A generalized processor sharing approach to flow control--- the multiple node case, Tech. Rep. 2076, Lab. for Inform. and Decision Syst., M.I.T., 1991.

[19] Performance Application Programming Interface (PAPI), http://icl.cs.utk.edu/papi/.

[20] Receive Side Scaling (RSS), http://www.microsoft.com/whdc/device/network/NDIS_RSS.mspx/.

[21] K. W. Ross, Hash routing for collections of shared web caches, IEEE Network, Vol. 11, No. 6, November-December 1997.

[22] E. Amir, S. McCanne, and R. Katz, An active service framework and its application to real-time multimedia transcoding, in Proc. of ACM SIGCOMM, Sept. 1998.

[23] S. Chandra, C. S. Ellis, and A. Vahdat, Differentiated multimedia web services using quality aware transcoding, in Proc. of IEEE INFOCOM , Mar. 2000.