VOD 시스템에서 최적의 서비스 허용을 위한 콘텐츠 저장 알고리즘

정지찬[O] 고재두 송민석 심정섭

인하대학교 컴퓨터정보공학과

jchjung@inhaian.net, goldedit@gmail.com, mssong@inha.ac.kr, jssim@inha.ac.kr

An algorithm to maximize the service ratio in VOD systems

Jichan Jung[O] Jae-Doo Go Minseok Song Jeong Seop Sim

School of Computer Science and Information Engineering, Inha University

요 약

VOD 서버에 서비스를 요청하는 단말장치의 종류가 다양해짐으로 인해 VOD 서비스 사업자가 제공해야 하는 해상도의 종류 역시 다양해지고 있다. 단말장치가 서비스를 요청할 때 서버는 단말장치에 맞는 해상도로 서비스를 제공해야 하는데 대개의 경우 서버의 용량이 제한되어 있기 때문에 비디오별로 모든 해상도를 저장하고 있을 수는 없다. 단말장치가 서버에 저장되어 있는 해상도를 요청한 경우라면 바로 서비스가 가능하다. 하지만 단말장치가 서버에 저장되어 있지 않은 해상도를 요청해 왔다면 저장되어 있던 파일을 이용해 해상도를 변환한 후 서비스를 해주어야 한다. 만약 서버가 해상도를 변환하는 빈도가 높아 CPU 가용률이 충분하지 않다면 다른 단말장치들의 서비스 요청에 바로 응할 수 없게 된다. 따라서 서버에 저장되는 파일들을 CPU 사용률을 줄일 수 있는 해상도의 파일들로 저장하여 CPU 가용률을 높인다면 보다 많은 단말장치의 요청을 허용할 수 있을 것이다.

본 논문에서는 한정된 저장 용량을 가진 VOD 서버가 단말장치의 서비스 요청들을 최대한 허용하기 위해 저장해야 할 각 비디오의 버전들을 분기한정 기법을 이용하여 찾는 알고리즘을 제시한다.

1. 서 론

최근 다양한 단말장치의 등장과 인터넷 환경의 급속한 변화로 VOD(video on demand) 서비스가 각광을 받고 있다. VOD는 주문형 비디오라고 하며 가입자의 요구에 따라 원하는 프로그램을 취사선택하여 수신 받는 영상 서비스를 말한다. VOD는 IPTV나 기타 멀티미디어 콘텐츠를 이용하는 단말장치에서 활용도가 높은 서비스이다. VOD 서비스가 각광을 받는 이유로 TV 방송 기반에서는 영상물 시청에 시간과 장소의 제한이 컸지만 VOD 서비스에서는 그런 점이 많이 완화 되었다는 점을 들 수 있다. 또한 소비자들이 VOD를 이용한 기록을 토대로 사업자가 서비스를 제공하는 전략을 신속, 정확히 수정하여 소비자들의 기호 만족을 높일 수 있는 점을 들 수 있다.

VOD 서버에 서비스를 요청하는 단말장치의 종류가 다양해짐으로 인해 VOD 서비스 사업자가 제공해야 하는 비디오들의 해상도의 종류 역시 다양해지고 있다 [1,2,3]. 만약 서버에서 제공하는 비디오의 해상도를 단말장치가 수용할 수 없다면 이용에 제한이 발생한다. 예를 들어 단말장치의 다양화 측면에서 볼 때 VOD 제공 서버에 접속해 온 클라이언트들의 단말장치가 노트북, PDA, PMP라고 하면 서버는 이들이 요청한 해상도에 맞는 비디오를 제공할 수 있어야 한다. 또한 네트워크의 대역폭(bandwidth) 측면에서 볼 때 단말장치가 데스크톱 정도의 고성능이고 네트워크의 대역폭이 고화질을 즐기기에 충분하다면 굳이 저화질의 서비스를 이용하지는 않을 것이다. 따라서 서버는 이처럼 다양한 경우에 부합하는 해상도를 제공해 주어야 한다.

서버가 요청받은 해상도의 버전을 가지고 있다면 바로 서비스를 하면 되지만 서버에 요청받은 해상도의 버전이 없을 경우에는 저장되어 있는 파일들 중에서 요청된 해상도보다 상위의 해상도를 이용해 서비스할 버전을 만들어 제공해야 한다. 이때 선택되는 상위의 버전은 요청된 버전에서 가장 가까운 버전이 되도록 한다. 이는 좀 더 상위버전으로 갈수록 원하는 버전을 만들 때 필요한 CPU 사용률이 증가하기 때문이다. 하위의 해상도를 이용할 경우 단순히 패딩(padding)을 하는 것과 같아서 화질의 향상을 기대하기 어렵다. 저장되어 있는 파일을 이용하여 새로운 해상도의 파일을 만들어내는 작업을 트랜스코딩(transcoding)이라고 한다.

전통적인 트랜스코딩에는 동적(dynamic)인 방식과 정

적(static)인 방식이 있다[1,2,3,4,5]. 동적인 방식은 원본을 가지고 있으면서 단말장치들이 요구하는 해상도가 원본이 아닌 경우 매번 만들어서 서비스를 제공하는 방식이다. 정적인 방식은 모든 해상도를 서버가 가지고 있어서 요청이 들어올 경우 단말장치들에게 바로 서비스를 해주는 방식을 말한다. 동적인 방식은 서버 구축에 필요한 저장장치의 용량이 정적인 방식에 비해 작으나 서비스 요청 시 트랜스코딩의 빈도가 높으므로 CPU 사용률이 높다. 정적인 방식은 동적인 방식과 반대의 특성을 가지는데 CPU 사용률은 낮으나 저장장치의 용량이 방대해질 수 있다. 트랜스코딩은 서버, 프록시(proxy), 클라이언트의 단말장치 수준에서 일어날 수 있다. 본 논문에서는 서버 수준에서의 트랜스코딩에 대해 다룬다. 서버 수준에서 다루는 것은 클라이언트 수준에서의 트랜스코딩이 해당 단말장치의 CPU 처리능력에 큰 영향을 받기 때문이다[1,2].

서버에 저장되어 있는 각 비디오들의 해상도의 종류는 CPU 가용률을 최대로 할 수 있는 해상도들로 구성이 되어야 한다. CPU 가용률이 충분하지 않을 경우 서비스를 요청한 단말장치가 대기하는 시간이 길어지기 때문에 원활한 서비스가 되지 않는다. 서버를 구축하는데 필요한 대용량 저장장치의 구성비용이 저장장치의 지속적인 가격하락과 기술증진으로 인해 예전에 비해 상대적으로 낮아졌다. 따라서 단순 용량 추가로 비디오마다 더 많은 해상도를 갖춤으로써 CPU 가용률을 높일 수도 있다. 하지만 고화질의 해상도를 요구하는 클라이언트의 수 역시 상대적으로 증가하고 있어 단순히 저장장치의 규모를 확장하여 서비스의 질을 높이는 것에는 한계가 있다. 따라서 서버의 내부 구성을 최적의 상태로 유지하여 CPU 가용률을 최대가 되도록 하는 방법이 필요하다.

본 논문에서는 한정된 저장 용량을 가진 VOD 서버가 단말장치의 서비스 요청들을 최대한 허용하기 위해 저장해야 할 각 비디오의 버전들을 분기한정 기법을 이용하여 찾는 알고리즘을 제시한다. 단말장치의 서비스 요청을 최대한 허용하기 위해서는 트랜스코딩을 위한 CPU 사용률을 최소화해야 하며 이는 CPU 가용률이 최대가 되어야 함을 의미한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구 및 용어들을 제시하고 3장에서는 CPU 가용률을 최대로 만들어 최적의 서비스가 가능하도록 하는 버전집합을 찾는 알고리즘을 제시하고 4장에서는 실험결과 및 분석을 제시한다.

2. 관련 연구

2.1 최적버전선택 문제

서버 내의 VOD 서비스를 위해 존재하는 모든 비디오 파일의 수를 NV라고 하고 각각의 비디오 파일을 V_i $(i = 1, ..., NV)$라고 하자. 각 비디오 당 저장될 수 있는 최대 해상도 버전의 수를 NR이라고 할 때 각 비디오 파일들이 가지는 해상도 버전을 $V_i^k (k = 1, ..., NR)$로 나타낼 수 있다. 각 비디오 파일별 원본(original version)은 V_i^1이고 이는 최고 화질의 버전을 나타낸다. 서버에 저장되어 있는 모든 비디오 파일은 기본적으로 V_i^1을 반드시 가지고 있어야 한다. 비디오 V_i^k를 요청해 오는 접속확률(access probability)을 P_i^k이라고 하고, $\sum_{i=1}^{NV}\sum_{k=1}^{NR} P_i^k = 1$을 만족한다. 모든 비디오의 접속 확률은 미리 알려져 있다. 각 비디오 파일이 해상도 버전에 따라 구성될 수 있는 집합을 FS_i라고 하고, V_i^k의 수가 NR개일 때 만들어질 수 있는 FS_i의 원소의 개수 $NE = 2^{NR-1}$이 된다. FS_i의 각 원소를 가리킬 때 $FS_{ij} (j = 1, 2, ..., NR)$라고 한다.

예. $NR = 3$ 일 때,

$$FS_i = \{FS_{i1}(=\{V_{i1}\}),\ FS_{i2}(=\{V_{i1}, V_{i3}\}),\ FS_{i3}(=\{V_{i1}, V_{i2}\}),\ FS_{i4}(=\{V_{i1}, V_{i2}, V_{i3}\})\}$$

각 비디오 파일에 대해 S_{ij}^k는 V_i^k가 FS_{ij}에 저장되어 있는지의 여부를 나타낸다. 즉, 해당 버전이 FS_{ij}에 존재하면 $S_{ij}^k = 1$, 해당 버전이 FS_{ij}에 존재하지 않으면 $S_{ij}^k = 0$이다. 어떤 단말장치가 $S_{ij}^k = 1$ 인 파일을 요청한다면 직접적으로 서비스를 제공하고 $S_{ij}^k = 0$ 인 파일을 요청한다면 서버에 저장되어 있는 파일 중 V_i^k에 가장 근접한 상위 해상도의 파일을 선택하여 트랜스코딩 과정을 거친 후 파일을 서비스 한다. C_{ij}^k는 FS_{ij}의 여러 해상도 중에서 V_i^k를 트랜스코딩하기 위해서 필요한 CPU 사용률을 나타낸다. $S_{ij}^k = 1$ 이면 $C_{ij}^k = 0$이 되고 그 이외의 경우에는 CPU 사용률을 가장 적게 할 수 있는 버전을 선택해야 한다. 서버가 이용할 수 있는 총용량을 TS라고 하고 FS_{ij}를 저장하는데 필요한 용량을 STR_{ij}라고 하자. CU_{ij}는 FS_{ij}를 구성하는데 요구되는 CPU 사용률을 말한다. $CU_{ij} = \sum_{k=1}^{NR}(P_{ij}^k \times C_{ij}^k)$로 표현된다. SV_{ij}는 FS_{ij}를 구성할 때 절약되는 CPU 사용률을 말한다. 즉, $SV_{ij} = CU_{i1} - CU_{ij}$이고 $SV_{i1} = 0$이다. 서버는 각각의 요청을 처리할 때의 CPU 가용률을 높이기 위해 용량제한을 만족하면서 각 FS_i에 대해 CPU 사용률을 최소화하는 V_i의 버전들의 집합인 FS_{ij}를 찾아야 한다. 앞으로 이 FS_{ij}를 $SE_i (SE_i = 1, ..., NE)$로 나타낸다. 각각의 FS_i에서 SE_i를 찾는 문제를 최적버전선택 문제라 하고 다음과 같이 정의한다.

정의 1: 최적버전선택 문제

집합 $FS_i (1 \le i \le NV)$가 주어졌을 때, $\sum_{i=1}^{NV} STR_{i,SE_i} \le TS$를 만족하며 CPU 가용률($\sum_{i=1}^{NV} SV_{i,SE_i}$)을 최대가 되도록 하는 집합 $SE_i (SE_i = 1, ..., NE)$를 찾는 문제를 최적버전선택 문제라 한다.

2.2. 다중선택배낭 문제

0-1 배낭문제(0-1 knapsack problem)[6]는 용량이 c인 배낭이 있을 때 하나의 클래스에 있는 아이템들을 이용해 배낭을 채울 때 아이템들의 무게의 합이 용량 c를 초과하지 않고 아이템들의 총 이익이 최대가 되도록 하는 문제이다. 0-1 배낭문제는 잘 알려진 NP-hard 문제이다[6]. 이를 확장하여 용량이 c인 배낭을 각각 여러 개의 아이템으로 구성된 서로 다른 m개의 클래스에서 정확히 하나씩의 아이템을 선택하여 채웠을 때 이들이 이루는 이익의 합이 최대가 되도록 하는 문제를 생각해보자. 각 클래스 $N_i(1 \le i \le m)$의 아이템의 개수를 n_i개라고 하자. 이 n_i개의 아이템 중 j번째 아이템의 이익을 p_{ij}, 무게를 w_{ij}라 하자. x_{ij}는 클래스 N_i에서 j번째 아이템이 선택되면 1이고, 선택되지 않으면 0이라 하자. 이때 다음과 같이 다중선택배낭 문제(Multiple Choice Knapsack Problem)[6]를 정의할 수 있다.

정의 2: 다중선택배낭 문제(MCKP)

다음의 (1), (2), (3)을 만족할 때 $\sum_{i=1}^{m}\sum_{j \in N_i} p_{ij}x_{ij}$를 최대화하는 문제를 다중선택배낭 문제라 한다.

(1) $\sum_{i=1}^{m}\sum_{j \in N_i} w_{ij}x_{ij} \le c,$

(2) $\sum_{j \in N_i} x_{ij} = 1,\ i = 1, ..., m,$

(3) $x_{ij} \in \{0, 1\},\ i = 1, ..., m,\ j \in N_i$

NP-hard 문제인 0-1 배낭문제는 다중선택배낭 문제로 환원(reduce)되기 때문에 다중선택배낭 문제도 NP-hard 문제이다[6]. 0-1 배낭문제를 다중선택배낭 문제로 변형(transformation)하는 방법은 다음과 같다. 0-1 배낭문제의 각 아이템의 이익을 p_j, 무게를 w_j로 나타낼 때, 다중선택배낭 문제에서 각 클래스가 정확히 두 개의 아이템($(p_{i1} = 0,\ w_{i1} = 0)$과 $(p_{i2} = p_j,\ w_{i2} = w_j)$)을 갖도록 한다. $(p_{i1} = 0,\ w_{i1} = 0)$인 아이템만으로 용량이 c인 배낭을 채운 후 더 많은 이익을 가지도록 각 클래스에 남은 아이템을 하나씩 바꿔 나간다. 이를 통해 다항식 시간 안에 임의의 0-1 배낭문제가 다중선택배낭 문제로 쉽게 변형됨을 알 수 있다. 이 과정은 다중선택 배낭 문제의 정의에 어긋나지 않으면서 0-1 배낭문제의 해를 구하는 과정이 된다. 최적버전선택 문제는 서버의 저장용량을 나타내는 TS, 각 비디오별 선택 가능한 버전집합들이 담긴 FS_i, 각 버전집합에서 절약되는 CPU 사용률인 SV_{ij}, 각 버전집합에서의 용량인 STR_{ij}가 각각 배낭의 용량 c, 클래스 N_i, 이익 p_{ij}, 무게 w_{ij}에 해당하는 다중선택배낭 문제이므로 역시 NP-hard 문제이다.

2.3. 분기한정 기법

다중선택배낭 문제를 풀이하는 대표적인 기법으로 분기한정 기법(branch and bound)을 들 수 있다[6]. 분기한정 기법은 최적화 문제를 해결하는 기법 중의 하나로 역추적(backtracking)을 이용하는 기법으로써 역추적과 마찬가지로 상태공간트리를 사용한다[7]. 최적화문제의 해(solution)가 될 수 있는 후보들의 모임을 해공간이라고 한다. 이 해공간은 트리형식으로 나타낼 수 있는데 이 트리를 상태공간트리라고 한다. 즉, 루트(root)에서부터 리프(external node, leaf)까지의 경로 상에 있는 노드들을 선택하게 되면 하나의 해답후보가 되는 트리를 말한다. 상태공간트리에서 각 노드는 문제상태(problem state)라고 부르고, 해의 후보에 해당하는 노드를 해상태(solution state)라고 부른다. 해상태 중에서 문제의 해에 해당하는 노드를 답상태(answer state) 또는 답노드(answer node)라고 부른다.

다중선택배낭문제의 상태공간트리의 노드 수는 $\prod_{i=1}^{m}|N_i|$ $(=|N_1| \times \cdots \times |N_m|)$이다. 하지만 단순 우성의 개념을 이용하면 상태공간트리의 노드 수를 줄일 수 있다.

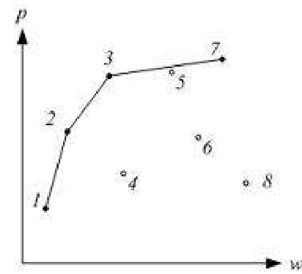

그림 1 무게-이익 좌표 평면에 표시된 아이템들.

정의 3: 단순 우성(simple dominance)

동일한 클래스 N_i 안에 존재하는 두 개의 아이템 j와 k가 다음을 만족하면 아이템 j는 아이템 k에 대해 단순 우성이라고 한다.

$w_{ij} \le w_{ik}$ 이고 $p_{ij} \ge p_{ik}$.

단순 우성의 개념은 두 개의 아이템에 대해 더 무거운 아이템이 더 적은 이익을 가질 때 우성인 아이템만을 최적해를 찾는데 이용하자는 것이다. 이는 우성인 아이템을 포함하는 최적해가 존재하기 때문이다[6]. 예를 들어, 그림 1은 같은 클래스 내의 8개의 아이템을 무게-이익 좌표평면에 나타낸 것이다. 여기에서 4번 아이템은 2번 아이템보다 무겁지만 이익은 더 적다. 즉, 4번 대신 2번을 선택하면 더 적은 무게로 높은 이익을 얻을 수 있으므로 최적해를 얻기 위해서 4번 아이템은 선택할 필요가 없다. 그림 1에서 6번, 8번 아이템 역시 마찬가지로 2번 아이템보다 무겁지만 이익은 더 적으므로 선택할 필요가 없다. 그림 1에서는 1번, 2번, 3번, 5번, 7번 아이템이 다른 아이템들에 대해 단순 우성이므로 나머지 4번, 6번, 8번 아이템은 상태공간트리에서 삭제한다면 총 수행 시간을 줄일 수 있다.

역추적이 상태공간트리를 탐색할 때 항상 깊이우선탐색(depth first search)을 사용하는데 반해 분기한정 기법

은 상태공간트리를 탐색하는 방법에 제한이 없다. 본 논문에서는 상태공간트리를 탐색하는 방법으로 너비우선탐색(breadth first search)을 수정한 최선우선탐색(best first search)을 이용한다. 일반적으로 너비우선탐색은 역추적의 깊이우선탐색에 비해 탐색 시 탐색하는 노드의 수에 별다른 차이가 없다. 하지만 최선우선탐색은 너비우선탐색을 할 때 지금까지 선택한 노드들 중에서 얻을 수 있는 이익의 상한값(upper bound)이 가장 큰 노드를 다음에 선택하여 탐색되는 노드의 수를 줄일 수 있다. 다중선택배낭 문제에서 상한값을 구하는 방법에는 그리디 기법(greedy approach), Dyer-Zemel 알고리즘 등 여러 가지가 있다[6].

본 논문에서는 각 노드에서 그리디 기법을 통해 상한값 및 반드시 얻을 수 있는 하한값을 계산한다. 각 노드에서 상한값 및 하한값을 구할 때 비교해야 할 아이템들의 수는 다음의 LP 우성 개념을 이용하여 줄일 수 있다.

정의 4: LP 우성(linear programming dominance)
동일한 클래스 N_i 안에 존재하는 세 개의 아이템 j, k, l이 다음을 만족하면 아이템 j와 l은 아이템 k에 대해 LP 우성이라고 한다.

$$\frac{p_{il}-p_{ik}}{w_{il}-w_{ik}} \geq \frac{p_{ik}-p_{ij}}{w_{ik}-w_{ij}}.$$

LP 우성인 아이템들은 주어진 모든 아이템들을 무게-이익 좌표 평면에 표현했을 때 볼록표면(convex hull)을 형성하게 된다. (그림 1 참조.) LP 우성의 개념을 이용하여 축소된 집합에 대해 상한값을 계산해도 축소되기 이전 집합에 대한 상한값보다 작지 않음이 보장된다[6].

3. 알고리즘

본 논문에서 제시하는 최적버전선택 문제 해결 알고리즘은 크게 두 단계로 구성된다. 먼저 상태공간트리 탐색 과정에서의 계산량을 줄이기 위한 해공간을 축소시킨다. 다음으로 상한값과 하한값을 이용하여 최선우선탐색 전략으로 최적버전을 찾는다.

3.1 해공간 축소

각 FS_i의 원소 수가 NE 개이므로 최적버전선택 문제의 상태공간트리에는 NE^{NV} 개의 노드가 존재하게 된다. 따라서 최악의 경우 대단히 큰 계산량이 필요하다. 만약 FS_i의 원소수를 줄일 수 있다면 최적해를 찾아가는 과정에서 분기되는 노드의 수가 줄어 탐색해야 할 노드의 수가 줄어들 것이다.

본 논문에서는 앞에서 설명한 단순 우성의 개념을 이용하여 상태공간트리의 탐색할 노드의 수를 줄이는 해공간을 축소시킨다. 앞에 설명된 바와 같이 각 FS_i는 다중선택배낭 문제에서 각각의 클래스에 해당하고 SV_{ij}는 이익, STR_{ij}는 무게에 해당되므로 최적버전선택 문제에서의 단순 우성의 개념은 다음과 같이 정의될 수 있다.

정의 5: 최적버전선택 문제에서의 단순 우성
최적버전선택 문제에서 동일한 클래스 FS_i 안에 존재하는 두 개의 버전집합 FS_{ij}와 FS_{ik}가 다음을 만족하면 버전집합 FS_{ij}는 버전집합 FS_{ik}에 대해 단순 우성이라고 한다.

$$STR_{ij} \leq STR_{ik} \text{ 이고 } SV_{ij} \geq SV_{ik}.$$

단순 우성인 버전집합을 찾는 과정은 다음과 같다. 먼저 각 $FS_i(1 \leq i \leq NV)$별로 모든 $FS_{ij}(1 \leq j \leq NE)$를 저장공간($STR_{ij}$)에 대해 오름차순으로 정렬한다. 다음으로 가장 저장 공간이 적은 버전집합부터 시작하여 연속된 두 버전집합 중 무거운 버전집합의 이익이 높아지지 않으면 더 무거운 버전집합을 FS_i에서 삭제한다.

단순 우성 개념을 이용하여 원소수가 축소된 FS_i를 R_FS_i라고 하자. 해공간 축소 과정 후의 상태공간트리에서 최악의 경우에 탐색하게 되는 노드의 수는 $\prod_{i=1}^{NV}|R_FS_i|$ $(=|R_FS_1|\times\cdots\times|R_FS_{NV}|)$이다.

3.2 최적버전선택

본 논문에서는 앞에서 설명한 LP 우성의 개념을 이용하여 상한값과 하한값 계산 시간을 줄인다. 이때, 주어진 $FS_i(1 \leq i \leq NV)$를 이용하여 LP 우성 아이템들을 결정해도 되지만 최적해에 포함될 수 있는 아이템들의 집합인 R_FS_i를 이용하는 것이 더 효율적임을 알 수 있다. 최적버전선택 문제에서의 LP 우성의 개념은 다음과 같다.

정의 6: 최적버전선택 문제에서의 LP 우성
최적버전선택 문제에서 동일한 클래스 R_FS_i 안에 존재하는 세 개의 버전집합 R_FS_{ij}, R_FS_{ik}, R_FS_{il}이 다음을 만족하면 버전집합 R_FS_{ij}와 R_FS_{il}은 버전집합 R_FS_{ik}에 대해 LP 우성이라고 한다.

$$\frac{SV_{il}-SV_{ik}}{STR_{il}-STR_{ik}} \geq \frac{SV_{ik}-SV_{ij}}{STR_{ik}-STR_{ij}}.$$

각 R_FS_i에서 LP 우성인 버전집합들은 앞에서 설명한 대로 R_FS_i의 버전집합들을 무게(STR)-이익(SV) 좌표 평면에 표현한 뒤 볼록표면을 구성하는 버전집합들을 찾으면 된다. 이렇게 구해진 버전집합을 LP_FS_i라 하자.

본 논문에서는 우선순위큐를 이용하여 최선우선탐색 전략을 구현한다. 초기에 LP_FS_1부터 LP_FS_{NV}까지의 버전집합들에 대한 최대가능 이익인 상한값과 반드시 얻을 수 있는 이익인 하한값을 그리디 기법을 이용하여 계산한다. LP_FS_1부터 LP_FS_{NV}의 각각의 버전집합들 중에서 첫 번째 버전집합인 LP_FS_{i1}을 포함시킨 후, 각 LP_FS_i에 남아 있는 버전집합들에서 각 버전집합과 바로 이전 버전집합과의 SV/STR의 비율을 구한다. 모든 LP_FS_i에서 구해진, 이전 버전집합과의 SV/STR 비율을 내림차순으로 정렬한 뒤 이 값이 가장 큰 버전집합을

선택해 같은 비디오 번호를 가진 기존 버전집합과 바꾸어 나간다. 이를 통해 어떠한 비디오라도 반드시 하나의 버전집합이 포함됨이 보장된다. 그리디 기법을 통해 0-1 배낭문제가 아닌 일반 배낭문제에서 최적해를 다항식 시간에 찾을 수 있다[6]. 일반 배낭문제는 0-1 배낭문제에서 아이템을 분할하여 일부만 선택할 수 있다는 것만 다른 문제이다. 0-1 배낭문제의 최적이익은 일반 배낭문제의 최적이익보다 클 수 없으므로 일반 배낭문제의 최적이익이 0-1 배낭문제의 최적이익에 대한 상한값이 될 수 있다. 그리디 기법에 의해 버전집합을 선택해 나갈 때, 버전집합이 완전히 포함될 수는 없어 일부만 포함되어야 할 경우가 있다. 이렇듯 분할되어야 하는 버전집합을 LP_FS_{ab}라 하자. LP_FS_1부터 LP_FS_{NV}까지의 버전집합들 중 그리디 기법을 이용하여 LP_FS_{ab}를 포함시키지 않고 얻은 이익(CPU 가용률)을 Y라 하고 LP_FS_{ab}를 분할시켜 계산한 최대이익(실수값 일 수 있음)을 Z라 하자. 이때 Y는 반드시 얻을 수 있는 하한값이 되고 $\lfloor Z \rfloor$는 어떠한 LP_FS_{ij}도 분할하지 않고 얻을 수 있는 상한값이 된다.

이후 각 R_FS_{1j}에서의 상한값과 하한값을 계산하고 우선순위큐를 이용하여 탐색을 진행해 나간다. R_FS_{1j}에서의 상한값은 SV_{1j}에 LP_FS_2부터 LP_FS_{NV}까지의 버전집합들을 이용하여 계산한 최대이익 Z의 $\lfloor Z \rfloor$를 더한 값이 된다. 하한값은 SV_{1j}에 Y를 더한 값이다. 이때 R_FS_{1j}에서 계산한 하한값이 이전의 하한값보다 크다면 R_FS_{1j}에서 계산한 하한값을 새로운 하한값으로 한다. 만약 R_FS_{1j}에서 계산한 상한값이 하한값보다 큰 경우 우선순위큐에 삽입을 한다. 이 과정은$|R_FS_1|$만큼 반복된다.

이후 각 버전집합의 상한값과 하한값은 비슷하게 계산한다. 우선순위큐에서 최대값 삭제 연산을 통해 추출된 버전집합을 R_FS_{ij}라고 하자. 다음으로 계산하여야 할 버전집합은 $R_FS_{i+1,k}(1 \le k \le |R_FS_{i+1}|)$이다. R_FS_{i+1}까지의 버전집합이 선택된 상황이므로 선택된 버전집합들의 이익을 합을 X라 하자. $R_FS_{i+1,k}$에서 상한값과 하한값은 X에 LP_FS_{i+2}에서 LP_FS_{NV}까지의 버전집합들을 그리디 기법을 통해 계산한 Y와 Z를 이용하여 구한다. 상한값은 $X + \lfloor Z \rfloor$가 되고, 하한값은 $X + Y$가 된다. 이때 하한값 $X + Y$가 기존의 하한값보다 크면 하한값을 $X + Y$로 갱신한다. 계산한 상한값이 하한값보다 크면 우선순위큐에 $R_FS_{i+1,k}$를 삽입한다. 만약 하한값이 초기에 설정한 상한값과 같다면 이때의 값이 최적 CPU 가용률이므로 이 값을 생성한 버전들을 출력한다. 이 과정을 우선순위큐에 더 이상 버전집합이 남아있지 않을 때까지 계속한다.

4. 실험 결과 및 분석

본 알고리즘의 실험환경은 다음과 같다.

- 프로세서: Intel Pentium 4 3GHz
- 메인메모리: 1GB
- 운영체제: Linux Fedora Core 7

FFMPEG 프로그램[8]에서 5가지의 샘플 비디오의 원본을 다른 해상도들로 트랜스코딩 시 사용된 CPU 사용률과 변환된 용량을 측정하고 이를 입력으로 쓰인 SV와 STR로 이용하였다.

실험에는 각각 500, 800, 1000, 1500개의 비디오의 원본을 이용하였는데 이는 5가지 샘플 비디오를 100번, 160번, 200번, 300번 반복하여 생성하였고 서로 다른 비디오를 나타낸다. 비디오의 개수가 500, 800, 1000, 1500개 일 때 각각에 주어지는 서버의 총용량으로 409,600MB, 819,200MB, 819,200MB, 1,228,800MB을 설정하였다. 해상도의 종류(NR)는 5로 설정하였고 실험에 쓰이는 접속자들의 접속 간격은 포아송 분포도(*Poisson distribution*)를 따른다고 가정하고 평균 3초로 하였다. 접속 확률은 α = 0.271인 *Zipf* 분포도을 따른다[9]. 하나의 비디오에서 모든 해상도를 선택할 접속 확률은 동일한 것으로 가정하였다. 즉, $\forall i,\ p_i^k = p_i \times \frac{1}{NR} (i = 1, ..., NV)$이다. 서버에 서비스를 요청한 총 클라이언트 중에 정상적으로 서비스를 해준 클라이언트의 비율을 서비스 허용률이라 한다. 실험에서는 본 논문에서 제시한 알고리즘(Version_Selection)과 다음 두 가지 선택 방법의 서비스 허용률을 비교하였다.

- 인기도 우선 선택 방식: NV개의 모든 비디오에 대한 원본들을 저장한 뒤, 비디오별 인기도 순에 따라 저장할 공간이 부족해질 때까지 저장해 나가는 방식이다.
- 무작위 선택 방식: 인기도 우선 선택과 마찬가지로 NV개의 모든 비디오의 원본들을 저장한 뒤 각 비디오들에 대한 버전집합을 무작위로 선택해 저장을 한다. 인기도 우선 선택과 마찬가지로 저장장치의 용량을 모두 소진 시킬 때까지 이 과정을 반복한다.

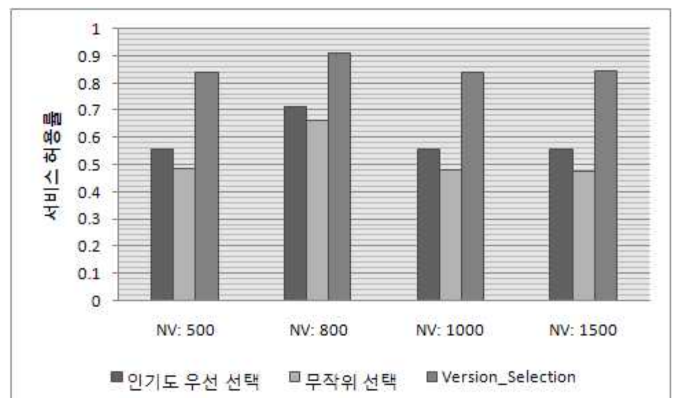

그림 2 비디오 개수에 따른 서비스 허용률

그림 2는 저장장치의 비디오의 개수에 따른 서비스 허용률을 나타낸다. 실험에서 수행한 4 가지의 경우 모두 본 논문에서 제시된 콘텐츠 저장방법이 가장 좋은 서비스 허용률을 보임을 알 수 있었다. 모든 경우에서 무작위 선택보다 많게는 35%, 인기도 우선 선택에 비해 많

게는 28% 정도로 허용률이 높게 나왔다. 실제 최적버전들을 찾아내기 위한 수행시간은 비디오 개수 500 개, 800 개, 1000 개, 1500개에 대해 각각 0.78초, 12.77초, 17.83초, 0.82초가 걸렸으며 메모리는 각각 3.5MB, 4.1MB, 4.1MB, 4.6MB가 사용되었다.

참고문헌

[1] R. Mohan, J. Smith, and C. Li. Adapting multimedia internet content for universal access. *IEEE Transactions on Multi*media, Vol.1, No.1, 104-114, March, 1999.
[2] B. Shen, S. Lee, and S. Basu. Caching strategies in transcoding enabled proxy systems for streaming media distribution networks. *IEEE Transactions on Multimedia*, Volume 6, Issue 2, 375-386, April 2004.
[3] M. Song and H. Shin. Replication and retrieval strategies for resource effective admission control in multi-resolution video servers. *Multimedia Tools and Applications Journal*, Volume 28, Issue 3, 89-114, March 2006.
[4] I. Shin and K. Koh. Hybrid transcoding for QoS adaptive video on demand services. *IEEE Transactions on Consumer Electronics*, Volume 50, Issue 2, 732-736, May 2004.
[5] W. Lum and F. Lau. On balancing between transcoding overhead and spatial consumption in content adaptation. *In Proceedings of the ACM MOBICOM*, 239-250, September 2002.
[6] Hans Kellerer and Ulrich Pferschy and David Pisinger, Knapsack Problems. *Springer*, 2004.
[7] Richard Neapolitan and Kumarss Naimipour, Foundations of Algorithms. *Jones and Bartlett Computer Science*, 2004.
[8] http://ffmpeg.mplayerhq.hu/.
[9] A. Dan, D. Sitaram, and P. Shahabuddin. Dynamic batching policies for an on-demand video server. *ACM/Springer Multimedia Systems Journal*, Volume 4, Issue 3, 112-121, 1996.