Chapter 16

A Piggybacking Policy for Reducing the Bandwidth Demand of Video Servers

Roberto de A. Façanha* (facanha@telemar-ce.com.br)
Nelson L. S. da Fonseca[&] (nfonseca@dcc.unicamp.br)
** Telecomunicações do Ceará, Telemar, Av. Borges de Melo, 1677, Fortaleza CE, Brazil*
& Institute of Computing, State University of Campinas, Campinas S.P., Brazil, PO Box 6176

Key words: Piggybacking, video-on-demand

Abstract: In video-on-demand systems, users expect to watch a film right after its selection. Nonetheless, such a short response time is feasible only if there is available bandwidth. In line with that, several techniques have been proposed to reduce the huge bandwidth demand on video servers. In this paper, we introduce the Piggybacking policy $S^{\prime}$, which adds a second level of optimization to the *Snapshot* policy. Moreover, we introduce a heuristic to reduce the complexity to generate the tree of superimposed video streams.

1. INTRODUCTION

Video on Demand (VoD) is one of the most promising applications in the future Broadband Integrated Services Digital Networks (B-ISDN). However, since it is extremely demanding on bandwidth, its availability in large scale requires the use of techniques for reducing this demand. Such techniques take into consideration the probability of a set of requests for popular videos (hot videos) getting to the system within a relatively short time interval, making it possible to provide a single video stream for all these requests.

Batching is a technique [1,2,3] which starts a new video stream by grouping all pending requests for a particular video in a certain time window. The main disadvantage of this technique is the delay introduced between the request and the beginning of exhibition of the video, which might lead the user to drop the request. *Piggybacking* is a technique [4,5,6] based on the fact that alterations of up to 5% in

J. N. de Souza et al. (eds.), *Managing QoS in Multimedia Networks and Services*

the display rate of a film are not perceptible to the user. In this way, requests can be promptly initiated and, by varying the display rate of subsequent video streams of the same film, we can superimpose the video streams as soon as they display the same frame. By doing so, we can provide only one video stream for all superimposed streams.

This paper presents the S^2 piggybacking policy. S^2 merges the video streams resulting from the use of the Snapshot policy on set of stream. Furthermore, a heuristic for reducing the complexity of generating merging trees is introduced. The Build Tree heuristic is $O(n^2)$ where n is the number of streams to be merged, whereas a solution via Dynamic programming is $O(n^3)$.

This paper presents the S^2 Piggybacking policy, which was discussed in the seminal paper [7]. The S^2 policy is a generalization of the *Snapshot* Algorithm policy [6]. Furthermore, we propose an heuristic for reducing the complexity of building merging tree.

The remaining of this paper is organized as follows: section 2 presents the *Snapshot* policy. Sections 3 and 4 introduce the S^2 policy and the proposed heuristic, respectively. Section 5 presents some conclusions. Appendix 1 presents the source code for the heuristic.

2. THE *SNAPSHOT* ALGORITHM

Snapshot is a Piggybacking policy [6] which attempts to minimize the number of frames shown for a given set of video streams. The computations performed by *Piggybacking* policies can be regarded as a binary tree (merging tree) in which the leaves correspond to streams, inner nodes are merges and the root is the final merge which forms the resulting stream of the set. Therefore, the number of possible trees generated by a set of streams constitutes the number of *Piggybacking* policies potentially optimal and is given by the $(n\text{-}1)$st Catalan [6,7] number, that is:

$$Catalan(n-1) = \frac{1}{n!} \cdot \frac{(2n-2)!}{(n-1)!}$$

which implies that it is not viable to perform exhaustive search in an attempt to find an optimal strategy and its corresponding binary tree.

The *Snapshot* policy builds an optimal merging tree in the following way: consider a set of n streams of a single video, comprised of L frames, and their positions being given by $f_1, f_2, \ldots f_n$, where $f_1 \geq f_2 \geq \cdots \geq f_n$, at a given instant of time T. The minimum and maximum rates of frames per second of the video streams are denoted $S_{\min}$ and $S_{\max}$, respectively. Let i and j, where

$1 \le i < j \le n$, be two video streams. $P(i,j)$ denotes the merging position (in frames) at which streams i and j show the same frame. Since stream i has speed $S_{\min}$ and stream j has speed $S_{\max}$, the merging position is given by:

$$P(i,j) = f_i + S_{\min} \cdot \left(\frac{f_i - f_j}{S_{\max} - S_{\min}} \right)$$

Let $C(i,j)$ be the cost of a *Piggybacking* policy and $\mathsf{T}(i,j)$ the corresponding binary tree. The cost of a given stream is given by

$$C(i,i) = L - f_i$$

To minimize $C(i,j)$ it is necessary that the following optimality principle be satisfied: *there must exist a stream* k, *with* $i \le k < j$, *such that the left and right subtrees must also be optimal.* These subtrees are denoted by $\mathsf{T}(i,k)$ and $\mathsf{T}(k+1,j)$. The cost of the tree that corresponds to the set of streams $i,\ldots,j$ is given by:

$$C(i,j) = C(i,k) + C(k+1,j) + \max((L - P(i,j)),0) \quad (1)$$

Therefore, the optimal policy for the streams $i,\ldots,j$ contains subtrees $i,\ldots,k^*$ and $k^*+1,\ldots,j$, where

$$k^* = \arg\min_{i \le k < j} C(i,k) + C(k+1,j) - \max((L - P(i,j)),0)$$

In this way, the cost of the set of n streams, $C(1,n)$, can be computed in a bottom-up fashion by means of a dynamic programming algorithm.

The *Snapshot* policy merges video streams in an interval (*Snapshot* interval) given by $I = W/S_{\max}$, where W is the optimal window size for the generalized simple merging policy [6]. At the end of the interval, the optimizing procedures described above are applied.

Sizing merging windows is an important issue. When the window size is large, a higher number of streams can be merged into a single stream, however, the mergings occur towards the end of the video. Conversely, when the window size is small, a reduced number of mergings are most likely to occur close to the beginning of the video. One can easily perceive that neither situation favors the reduction of the number of frames displayed by a set of streams. Aggarwal *et al.* [6] present an analytic method to optimize the size of the merging window to take into account the

anticipated rate of new requests, assuming that these are modeled by a Poisson process.

3. THE S^2 PIGGYBACKING POLICY

The *Snapshot* policy was proposed to minimize the number of displayed frames from the streams contained in an interval I. The optimization involves only the streams that arrive during the last *Snapshot* interval. It is possible to verify that the smaller the interarrival time (higher rates) the smaller the optimal window size is, allowing for one or several optimal windows contained in the maximum catchup window. The possibility of merging two streams separated by at most W_m frames, where W_m is the maximum catchup window size, allows for additional gain on the top of those from the *Snapshot* policy. In line with that, we define a novel policy called S^2 which aims at optimizing the number of displayed frames of a set of streams in a modified maximal merging window, W'_m, $\left(W \le W'_m \le W_m\right)$ and not only in the interval I. The modified maximal window consists of an integer number of optimal windows, i.e., the modified maximal window has $\lfloor W_m / W \rfloor$ optimal windows. The S^2 policy introduces a second level of merges, in which streams resulting from the *Snapshot* intervals are merged.

The S^2 policy works as follows: it initially applies the *Snapshot* algorithm over streams which survive past the *Snapshot* intervals (as originally proposed). It subsequently applies once again the *Snapshot* algorithm on the streams resulting from the first step. We should emphasize that according to the definition of the *Snapshot* policy, the speed $S_{\max}$ is assigned to these resulting streams with the exception of the one generated by the first optimal window contained in W'_m. Another important aspect is that contrary to what happens in the first level of optimization, the point where the optimization procedure is subsequently applied does not have to occur at fixed intervals. These intervals are denoted by I_{S^2} and their duration is determined by the request pattern in each window W'_m (`Figure 1`).

A natural generalization of the S^2 policy would be to consider n levels of optimization. Nonetheless, the gain that can be obtained by applying these additional levels are nearly null since streams in these levels are separated by roughly a number of frames which is close to W_m frames. Therefore, the introduction of extra merging levels would not produce effective reductions in the bandwidth demand.

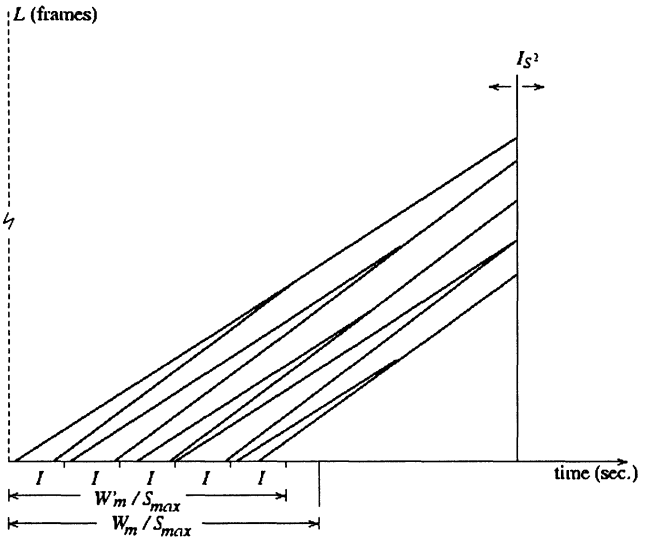

Figure 1. A scenario which shows that the last streams does not necessarily corresponds to the resulting streams of the last optimal window contained in the W'_m.

To evaluate the impact of introducing of a second level of optimization we compare S^2 to the original *Snapshot* policy and to the overall *Snapshot* policy which considers all the request for a given video without dividing time into *Snapshot* intervals. The objective function used in this paper differs from that considered in [6] which takes into account only the number of frames displayed after the interval I. Our objective function considers the total number of frames displayed for a set of streams, that is, it reflects the optimization that takes place in interval I as well as the ones which happen after that interval. So, the cost of exhibiting an individual stream is given by

$$C(i,i) = L$$

for a stream i, and is given by:

$$C(i,j) = Frames_I + (C(i,k) + C(k+1,j) - \max((L - P(i,j)),0))$$

for a set of streams $i,\ldots,j$, where the value of k is determined by Equation 1, and $Frames_I$ is the sum of frames displayed by the m discarded streams $Q(l)$, for $l = 1,\ldots,m$, before the end of the *Snapshot* intervals given by:

$$Frame_I = \sum_{l=1}^{m} Q(l)$$

3.1 Numerical Results

Results presented in this paper were obtaining via discrete-event simulation. We used the method of independent replications to compute confidence intervals with a confidence level of 95%. We assume that requests arrive according to a Poisson process and we use $S_{\min} = 28.5$ and $S_{\max} = 31.5$ frames per second. The graphics show the percentual reduction in the number of frames displayed. Three experiments were performed: i) analysis of the policies under different arrival rates; ii) analysis of the sensibility of the S^2 policy in relation to the window size; iii) analysis of the effect of the duration of a video and of arrival rate over the S^2 policy.

`Figure 2` illustrates the behavior of the policies under different arrival rates, i.e., we vary the mean interarrival time from 15 to 500 seconds. We can see that as the mean interarrival time assumes higher values the benefit of adopting *Piggybacking* are reduced. For high rates, the introduction of a second level of optimization allows gains of up to 8% better than those achieved by the *Snapshot* policy.

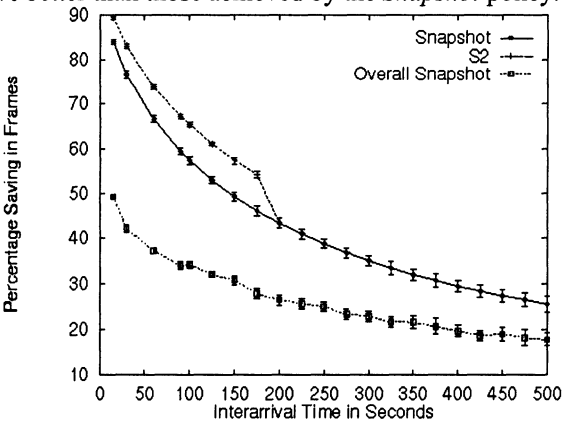

Figure 2. Comparison between the original Snapshot, the S^2 and the global Snapshot policies

`Figure 3` shows the behavior of the S^2 policy as a function of the window size considering mean interarrival times of 30 seconds and a video duration of 2 hours. It can be seen that the S^2 policy is insensitive to the variation of the window size. This is due to the fact that the second level of optimization compensates eventual mergings not performed due to small catchup window. In other words, when the size of the catchup window is small, few merges occur at the first level of optimization which implies in a small reduction on the number of displayed frames. Since there is a second level of mergings, the resulting streams from the first

merging level can still be merged leading to further reduction. As the catchup window gets close to maximum size, most of the gain comes from the first level of optimization.

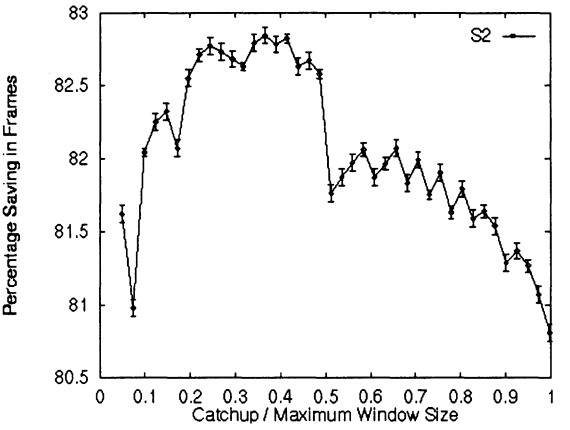

Figure 3. The impact of the window size on the percentage of nondisplayed frames under the S^2 policy

`Figure 4` illustrates the impact of both the video duration and the mean interarrival time on S^2 savings. In this figure, we consider videos with duration from 30 minutes up to 4 hours and mean interarrival time from 15 to 500 seconds. It can be seen that we obtain greater reductions on the number of displayed frames for longer videos in conjunction with higher arrival rates. The reduction on the bandwidth demand varies from 9% for videos of 30 minutes duration and mean interarrival time of 500 seconds. The reduction can be up to 93% for long videos (4 hours of duration) and interval of arrivals of 15 seconds.

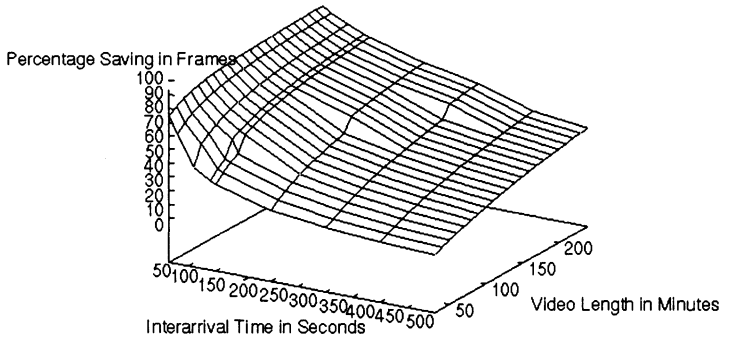

Figure 4. The impact of both request interarrival time and video length on the percentage of nondisplayed frames under the S^2 policy

4. REDUCING THE COMPLEXITY OF GENERATING MERGING TREE

If on one hand, the *Snapshot* policy is an attractive bandwidth reduction technique. On the other hand, the optimal merging (binary) tree of the optimal strategy is constructed by a dynamic programming algorithm whose complexity is $\Theta(n^3)$ [8], where n is the number of streams that survived past the *Snapshot* interval.

It was observed that merging trees can be built in a top-down fashion, differently from the dynamic programming algorithms which adopts a bottom-up fashion. In order to reduce it's complexity, we conceived a heuristic, called BuildTree, which is based on divide and conquer strategy. The basic principle of the heuristic consists in successively partitioning the set of streams to be merged in two groups and computing the costs of each subgroup until we obtain the merging tree. The division criterion used takes into consideration the segments of the merging tree (the number of frames of video given by the merging position) between the first stream and an intermediate stream (divisor stream) and between this and the last stream. In other words, we determine the stream that would represent the point of division of the set in potentially optimal subtrees. In this way, for a set of streams $i,\ldots,j$ there exists a stream k, with $i \le k < j$ which minimizes the absolute value of the difference the lengths of the segments $P(i,k)$ and $P(k,j)$, given by:

$$k^* = \arg\min i \le k < j \{|P(i,j) - P(k,j)|\}$$

After determining k^*, the original set of streams $i,\ldots,j$ is divided in the subsets $i,\ldots,k^*$ and $k^*,\ldots,j$, if $P(i,k^*) < P(k^*,j)$, or $i,\ldots,k^*-1$ and $k^*,\ldots,j$, otherwise, and then, the division criterion is re-applied to both. The algorithm considers two particular cases, whose entries have only two or three streams. These cases simplify the analysis in order to obtain a better performance. In the first case, the algorithm computes only the merging position and in the second, the segments $P(i,i+1)$ and $P(i+1,j)$ are compared and the larger value is discarded.

Therefore, the cost for a set of streams $i,\ldots,j$ is obtained through the sum of the lengths of the segments of the merging tree, which represents the cost of the $n-1$ streams except for the resulting stream. So, we must add the length of the resulting stream to this value to obtain the final cost.

In the process of determining the divisor stream (equation 5), $O(n)$ operations are sufficient. We can use the following property to reduce the number of operations carried out: "the divisor stream is the first whose merging segment from the initial stream to the one after the divisor is greater than the merging segment of the stream

following the divisor up to the last stream." But, in the worst case, $k = j-1$, $O(n)$ operations still take place.

THEOREM: THE BUILDTREE HEURISTIC HAS COMPLEXITY $O(n^2)$.
PROOF: See [10].

For a better understanding of the heuristic consider the following example. Suppose that in a given *Snapshot* six streams are initiated and that at the end of it, their positions are: 3338, 3010, 2908, 2316, 1650 and 503. Let i be the first stream; k be the divisor stream and j be the last stream of the set so that the six streams are partitioned into the sets $(1 \leftrightarrow 4)$ and $(5 \leftrightarrow 6)$, $(i = 1, k = 4, j = 6)$, given that the fourth stream is the first stream in which $P(i, k+1) \leq P(k+1, j)$, resulting in the tree in `Figure` 5-a. In sequence we re-apply the algorithm over the subsets $(i = 1, \ldots, j = 4)$ and $(i = 5 \text{ e } j = 6)$. `Figure` 5-b illustrates the final merging tree built by the algorithm (in this case, the same as the tree generated by the dynamic programming algorithm).

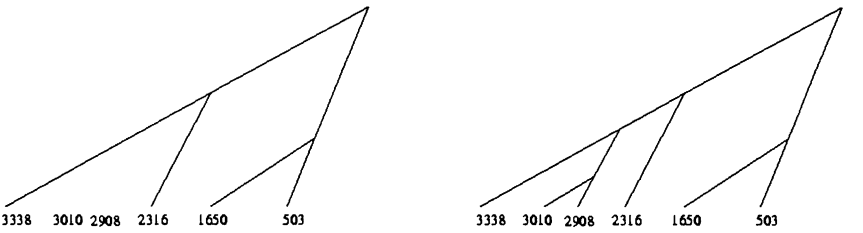

Figure 5. Construction of the merging tree by the heuristic BuildTree

In our experiments, the BuildTree heuristic was implemented using sequential search for the divisor stream in the set $i, \ldots, j$. The use of binary search would of course imply in considerable speed up. In Appendix 1, we show the heuristic as implemented in the C language.

4.1 Numerical Results

The proposed heuristic consists in an approximation of the optimal solution making it necessary to evaluate its precision in relation to the optimal one. With this intent, we execute both algorithms at the end of the *Snapshot* intervals to compare the costs of the generated merging trees. In this experiment, requests arrive according to a Poisson process and the mean interarrival time varies from 15 to 500 seconds.

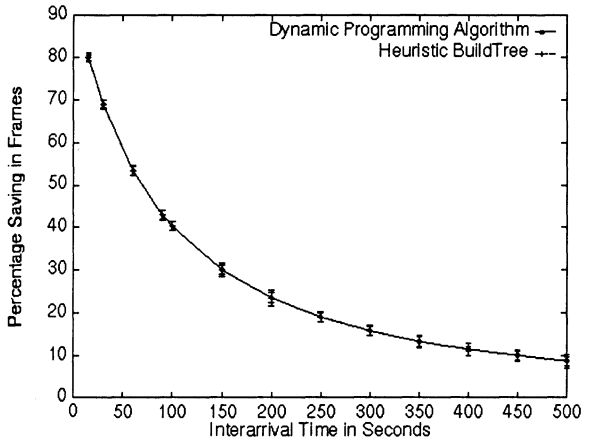

Figure 6. Comparison of the costs computed by the heuristic and by the dynamic programming algorithm

Figure 6 shows the mean percentual gain given by the dynamic programming algorithm and given by the heuristic. It can be seen that the BuildTree heuristic is extremely precise in relation to the optimal solution. This is due to the fact that in the majority of the cases the BuildTree heuristic obtains the optimal solution.

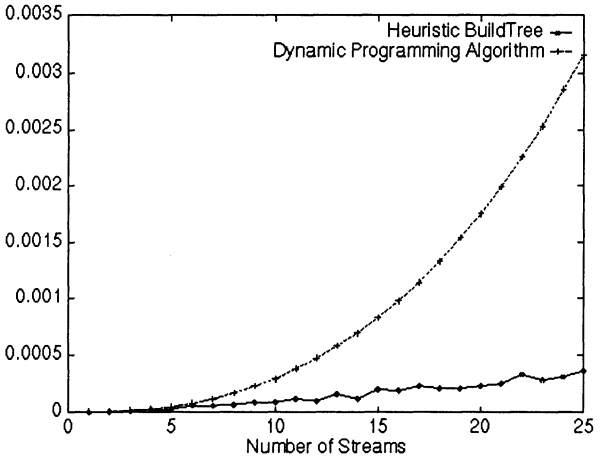

Figure 7. Comparison between the execution times of the heuristic and the dynamic programming algorithm

In *Figure 7*, instances from 1 to 25 streams (observed in other experiments) were fed as input for the computation of the execution times of both algorithms. We observed in all cases that the time in the execution of the algorithm was at most equal to the time of the dynamic programming algorithm.

5. CONCLUSION

This paper introduces the S^2 Piggybacking policy which adds a level of optimization in relation to the *Snapshot* policy. Moreover, we propose an approximate heuristic for reducing the complexity of building merging trees. We verified via simulations that the S^2 policy leads to less displayed frames than the *Snapshot* policy does. Furthermore, we note that the proposed heuristic is reasonably accurate, with a significantly lower complexity.

ACKNOWLEDGEMENTS

This work was partially supported by Pronex SAI, CNPq, and FAPESP.

REFERENCES

[1] Dan, D. Sitaram and P. Shahabuddin. "Dynamic Batching Policies for an on-demand Video Server". Multimedia Systems, Vol. 4, pp. 112–121, 1996.

[2] Dan, P. Shahabuddin, D. Sitaram and Don Towsley. "Channel Allocation under Batching and VCR Control in Video-on-Demand Systems". Journal of Parallel and Distributed Computing, Vol. 30, pp. 168–179, 1995.

[3] P. S. Yu, J. L. Wolf and H. Shachnai, "Design and Analysis of a Look-Ahead Scheduling Scheme to Support Pause-Resume for Video-on-Demand Applications". Multimedia Systems, Vol. 3, No. 4, pp. 137–149, 1996.

[4] L. Golubchik, John C. S. Lui and R. Muntz, Reducing I/O Demand in Video-on-Demand Storage Servers, In Proc. of ACM Sigmetrics, pp. 25–36, 1995.

[5] L. Golubchik, John C. S. Lui and R. Muntz, "Adaptive Piggybacking: A Novel Technique for Data Sharing in Video-on-Demand Storage Servers", Multimedia Systems, Vol. 4, No. 3, pp. 140–155, 1996.

[6] C. Aggarwal, J. Wolf and P. S. Yu, "On Optimal Piggyback Merging Policies for Video-on-Demand Systems", In Proc. of ACM SIGMETRICS, Vol. 24, No. 1, pp. 200–209, New York, May, 1996.

[7] Roberto de A. Façanha, N. L S. da Fonseca and P. J. de Rezende. "Reducing Bandwidth Demand on Video Servers", Reduzindo a Demanda de Banda Passante em Servidores de Vídeo (In Portuguese), In Proc. of the Second French-Brazilian Seminar on Distributed Systems: Multimedia Architectures for Telecommunications, Federal University of Ceará, Brazil and University of Versailles, France, pp. 308–319. Fortaleza, Ceará, 1997.

[8] M. Gardner, "Catalan Numbers", Scientific American, pp. 120–124, June, 1976.

[9] T. H. Cormen, C. E. Leiserson and R. L. Rivest, Introduction to Algorithms, The MIT Electrical Engineering and Computer Science Series, The MIT Press, 3rd edition, 1991.

[10] Roberto de A. Façanha, Nelson L. S. da Fonseca and Pedro J. de Rezende, The S^c Piggybacking Policy, Multimedia Tools and Applications, Vol. 8, pp. 371–383, 1999.

APPENDIX

1. CODE OF THE BUILDTREE HEURISTIC

```
int BuildTree(int Posicao[], int i, int j) {
int n = j - i + 1, // Número de fluxos.
    k = i,       // Fluxo divisor.
    Custo = 0,   // Custo da árvore.
    MPik,        // Posição de Mesclagem dos fluxos i e k.
    MPkj;        // Posição de Mesclagem dos fluxos k+1 e j.
switch(n) {
    case 0:
    case 1:return(0);
    case 2:return(MergePosition(Posicao[i], Posicao[j]));
    case 3:MPik = MergePosition(Posicao[i], Posicao[i+1]);
           MPkj = MergePosition(Posicao[i + 1], Posicao[j]);
           return(((MPik  > MPkj) ? MPkj : MPik) + MergePosition(Posicao[i],
                  Posicao[j]));
    default:
           while (MergePosition(Posicao[i], Posicao[k+1]) < MergePosition(Posicao[k+1],
                  Posicao[j]))
               k++;
          Custo  = BuildTree(Posicao, i, k);
          Custo += BuildTree(Posicao, k + 1, j);
          Custo += MergePosition(Posicao[i], Posicao[j]);
          return(Custo);
      }
    }
```

Algorithm 1: Heurística de Construção da Árvore de Mesclagem.