Modeling "Just-in-Time" Communication On the Optimal Resource Utilization in Distributed Real-Time Multimedia Applications

R. Yang*†, R.D. van der Mei*†, D. Roubos*, F.J. Seinstra*, G.M. Koole*, and H.E. Bal*

*Vrije Universiteit Amsterdam, Faculty of Sciences
De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands
Email: ryang@few.vu.nl

†Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ, Amsterdam, The Netherlands

***Abstract*—The applications of Multimedia content analysis (MMCA) operating in real-time environments must run under very strict time constraints, e.g. to analyze video frames at the same time as a camera produces them. To meet these requirements, largescale multimedia applications typically are being executed on Grid systems consisting of large collections of compute clusters. Therefore, first, it is essential to determine the optimal number of compute nodes per cluster, properly dealing with the perceived computation versus communication ratio that depends on several characteristics of the system. This issue is referred to as the problem of "Resource-optimization". Secondly, once the optimal number of resources are available, it is important to assign video frames at the right times to the server, so as to obtain the highest service utilization possible, and to minimize the buffering time for individual video frames at the server side. We refer to this issue as "Just-in-time" problem.**

Motivated by these observations, in this paper we first develop a simple and easy-to-implement method to determine the "optimal" number of parallel compute nodes. The method is based on the classical binary search method for non-linear optimization, and does not depend on the, usually unknown, specifics of the system. Second, we address the Just-in-time problem by introducing an adaptive control method that reacts to the continuously changing circumstances in Grid systems. [1].

I. Introduction

In recent years, the increasing role of multimedia data, in the form of still pictures, audio, speech, video, has lead to a demand for automatic collecting, comparing, processing and features extracting from multimedia data source. One of such applications is the automatic comparison technology used to recognize the forensic video evidence [3] obtained from the surveillance cameras in public locations.

Such applications using computerized technology to process the multimedia data have become a problem worth ever-increasing serious consideration as multimedia applications produce high data rates. The amount of information produced in the world increases by 30% every year. Print, film, magnetic, and optical storage media produced about 5 exabytes (10^{18}) of new information in 2002 [4]. The multimedia archives of a data center (for instance, a large hospital) now has a petabyte-scale (10^{15}) database of stored data. As individual compute clusters cannot satisfy the increasing computational demands, distributed supercomputing on large collections of clusters (Grids) is rapidly becoming indispensable.

Applications in Multimedia Content Analysis (MMCA) often must run under strict time constraints. For example, to avoid delays in queues of people waiting, a biometric authentication system must identify a person's identity within several seconds. Largely autonomous applications, such as the automatic detection of suspect behavior in video data obtained from surveillance cameras, may even need to work under real-time restrictions. In this kind of services-based distributed execution scenario, a client program (typically a local desktop computer) connects to one or more remote *multimedia servers*, each running on a (different) compute cluster. At application run-time, the client application sends video frames captured by a camera to the server, which performs the analysis in a data parallel manner.

For such applications executing on large collections of compute clusters, the resource utilization must be firstly automatically optimized. Efficient methods must be available to determine the optimal number of nodes of the compute cluster. This optimization problem generally depends on priori system information includes the running application using the compute cluster, and the specifics of the computation environment (e.g., network characteristics, CPU power memory, I/O). In this context, it is essential to properly balance the following trade-off: if the number of compute nodes is too low, then the processing power is insufficient to meet strict processing-time requirements of real-time applications; if the number of compute nodes is too high, the parallelization overhead will cause a degradation of the computational performance. This problem is referred to as the resource-optimization problem in this paper. Hence, there is an urgent need for *simple* and *easily implementable*, yet *effective* methods (in terms of the number of evaluation steps), to determine the optimal level of

[1]A partial and preliminary version of this paper has been presented at [1], [2]

parallelism. Also, the method should be *adaptable* to system variation.

Next to finding the optimal number of resources by solving the resource-optimization problem, it is essential to make use of the available resources efficiently by sending video frames at the right times to the server, so as to obtain the highest service utilization possible, and to minimize the service response time for individual video frames. If the employed multimedia server is keeping unoccupied, the analysis results for a video frame can be obtained in the fastest possible way. However, this arrangement is a waste of available compute resources, because this server is not working on previously submitted video data. If all video frames are sent immediately to multimedia server when there is a long queue before they can be processed, then the result data is not "up-to-date". This could be unbearable for some real time applications. To optimize resource utilization, it is essential to tune the transmission of video frames to the occupation of remote multimedia servers. However, due to variations in transmission latencies and other variabilities in the computing environment (e.g., CPU power, memory, I/O), it is difficult to accurately tune the sending of video frames to the variable response time of a multimedia server. In this paper we refer to this issue as the problem of "just-in-time" communication.

To solve the "just-in-time" communication problem, we need prediction methods that react to the continuously changing circumstances in Grid systems. An immediate consequence of a "just-in-time" communication approach is that a multimedia server always analyzes most recently generated ("up-to-date") video frames; the shortest server response delays are introduced due to frame buffering at either the client side or at the server. Clearly, this is an important, even critical requirement in real-time applications.

The remainder of this paper is organized as follows. In Section II we present related work, and indicate the limitation of existing methods. Section III presents the proposed approach in principle. Section IV presents the experimental setup, and describes example applications. In Section V our methods are formulated. Section VI discussed our experimental results. Finally, in Section VII we present our conclusions. In all these sections, the resource-optimization problem and the "just-in-time" communication problem are respectively covered.

II. Limitation of existing methods

Previous work in this field can be categorized into two groups. The first group that is relevant to our resource-optimization problem, belongs to the performance optimization problem of computer systems. In [5], Saavedra-Barrera et al. provide an estimation technique to solve the problem of the high complexity of complete analytical study of computer systems. This approach depends on sufficient priori knowledge of machine characterization and analysis of application program, and the entire system must be stable enough according the model we build. The drawback of the approach is that system variance is almost completely ignored. For applications working on extensive dense data fields (e.g., image data structures) this is a too crude restriction as variations in the hit ratio of caches and system interrupts often have a significant impact on performance [6], [7].

Many other performance estimation techniques that incorporate more detailed behavioral abstractions relating to the major components of a computer system [8], [9], however, need tens, if not hundreds, of platform-specific machine abstractions to obtain truly accurate estimations. Consequently, the essential requirements of simplicity and applicability are not satisfied. To overcome this problem, Seinstra et al. [10] have designed the Abstract Parallel Image Processing Machine (APIPM) model that has been used in a large set of realistic image processing applications to find the optimal number of compute nodes. The main advantage of this model is that predictions are based on the analysis of a small number of rather high level system abstractions (i.e., represented by the APIPM instruction set). The main limitation of this model, however, is that the instruction set and its related performance values are parameterized with a very large number of instruction behavior and workload indicators. As such, the model does not meet our requirements, as obtaining accurate performance values for all possible parameter combinations is both costly and complex.

For our "just-in-time" communication problem, we argue that existing prediction methods (i.e., the adapted mean-based method [11], the adapted median-based method [11], exponential smoothing [12]–[15], and the Robbins-Monro Stochastic Approximation method [16]) are not capable of adhering to the specific requirements of just-in-time communication. One problem of existing methods is that *random peaks* exist in the service processing time[Reference???]. These delays cause accumulative errors in predicting the exact moments of the coming data, resulting in significant deviations from the optimal rhythm in the transmission of frames. Another problem is that existing methods can not deal with *periodic peaks* very well either[Reference???]. We need additional policies to amend these particular problems.

III. Proposed approaches

In practice, running CPU-intensive applications in large-scale distributed computing environments typically consists of two phases: (1) an *initialisation phase* to determine the optimal number of compute nodes L^*, and (2) the *main phase* to actually run the application on the L^* parallel nodes. Here, two proposed approaches are used during respective phase.

A. On resource-optimization problem

First, we propose a simple method to determine the "optimal" level of parallelism, in which the number of evaluation steps is small. Unlike the analytical methods, our parallel program together with the underlying execution platform is treated as a black box from the resource allocator's point of view. This is due to the following requirements that we are looking for a general approach to solve the optimization problem:

- without priori knowledge about the parallel behavior of the adaptable application,

- do not rely upon system specific hardware and/or software characteristics of the applied cluster system.

In this context, experimental observations for realistic, large-scale problems in multimedia content analysis have revealed three important optimization properties.

First, in many situations the optimal number of parallel compute nodes is found to be a power of 2, i.e., of the form 2^m for some $m = 0, 1, \ldots$. This observation is important because it leads to a dramatic reduction of the set of possible solutions. For example, if the number of available compute nodes is m_{max}, the size of the solution space is reduced from m_{max} (i.e., the number of elements in the index set $\{1, \ldots, m_{max}\}$) to $\lfloor \log_2(m_{max}) \rfloor$ (i.e., the number of elements of the set $\{2^0, 2^1, \ldots, 2^K\}$ where $K = \lfloor \log_2(m_{max}) \rfloor$). Here the symbol $\lfloor x \rfloor$ represents the largest integer $\leq x$.

Second, on compute nodes consisting of multiple CPUs (and potentially multiple cores), for a fixed number of compute elements, using more compute nodes and less CPUs per node yields better performance.

Third, if the compute cluster processing time is denoted by $S(L)$, with L the number of compute nodes, then there exists a threshold value L^* such that $S(L)$ decreases fast as a function of L for $L < L^*$, whereas $S(L)$ flattens out, and may even increase, for $L > L^*$. L^* is commonly referred to as the *engineering knee*. Moreover, in practice using too many compute nodes may be very costly. L^* should be the smallest number that match the conditions specified above.

Based on these observations, our proposed method is aimed at determining L^* as the optimal point of operation. The method takes the idea of the well-known classical binary search method for non-linear optimization, and converges if the relative improvement of $S(L)$ with respect to L (on a log scale) is close enough to 0 (say $5 - 10\%$). To validate the effectiveness of the proposed method, we have performed extensive experimentation on a realistic distributed system (DAS-3 [17], [18]) for both real-time and off-line applications. The results show that our method is indeed highly effective.

B. On "just-in-time" communication problem

A simple execution approach to solve "just-in-time" communication problem, which we refer to as the back-to-back method (BBM), is to perform the sending of a newly generated video frame exactly after a result has been received from the same server (see Figure 1). Using the BBM method, any video frame processed by a multimedia server is guaranteed to be most up-to-date. A drawback of BBM, however, is that the server is idle when it has processed a frame and is waiting for the next one. In a bottleneck situation, the video frame transmission time from the client to the server (Tc_1) and the time to send a result back (Tc_2) may be long. For simplicity, we assume $Tc_1 = Tc_2 = Tc$. Then, the service utilization (SU) using BBM is given by

$$SU = \frac{Ts}{Ts + 2 \cdot Tc},$$

where Ts is denoted as the service processing time of a video frame. Obviously, if the communication time increases, service utilization decreases.

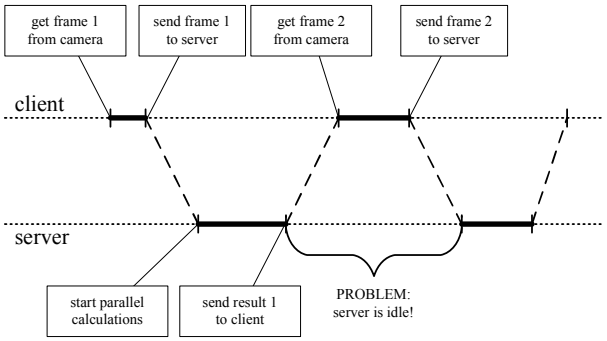


Figure 1. BBM approach for video frame transmission

An alternative approach, referred to as the buffer storage method (BSM), is to establish a buffer at the server side. As long as the buffer is not full, the client is allowed to keep sending frames to the server. When the server is busy, the frames will be stored in the buffer before being processed (see Figure 2). Using BSM, service utilization can reach 100%. However, the drawback is that the data in the buffer may have become outdated *before* the actual video content analysis even takes place, due to the long waiting time. A solution would be to simply remove outdated frames at the server side. This, however, leads to (a lot of) unnecessary traffic between client and server, which should be avoided as resources are scarce.

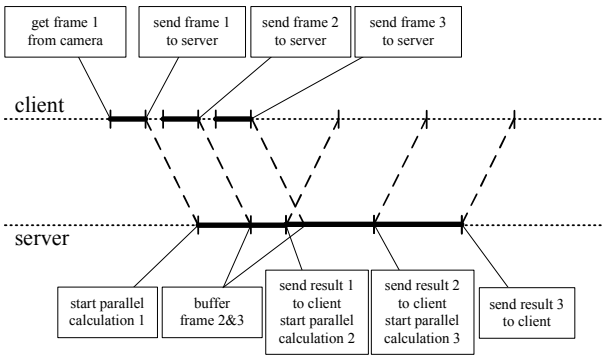


Figure 2. BSM approach for video frame transmission

Given the previous two methods, the optimal strategy would be to send each (i+1)-th frame with a delay after sending the i-th frame. The delay is exactly the processing time of the i-th frame. For instance, if the service processing time of the current frame equals Ts_i, sending the next frame after a period of Ts_i will give an optimal solution. With this strategy, the server gets the most up-to-date frame and the service utilization is unity (see Figure 3). Unfortunately, Ts_i is unknown before the result of the current frame is returned back to the client side. It is therefore essential to have an accurate prediction of the processing time of video frame data.

We have observed that existing predictive methods (i.e.,

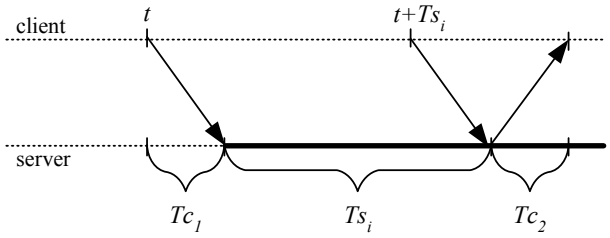


Figure 3. An optimal solution for video frame transmission

the adapted mean-based method [11], the adapted median-based method [11], exponential smoothing [12]–[15], and the Robbins-Monro Stochastic Approximation method [16]), are all capable of generating an accurate trend line based on the processing time of previous frames. However, for our just-in-time communication problem, these methods are not sufficiently optimized for particular cases. The first problem appears, when the processing time of certain frames suddenly become much longer (e.g., a peak) than the expected Ts obtained from a trend line. The sudden change breaks the rhythm of frame transmission and causes accumulative waiting times for all subsequent frames, even when the processing time returns back to the expected Ts (see Figure 4).

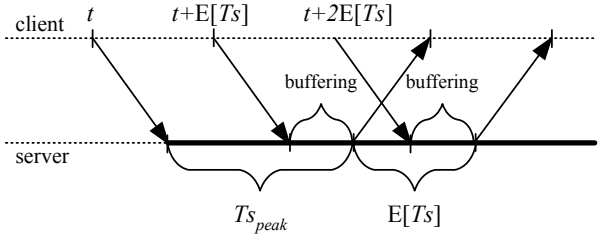


Figure 4. All frames are affected continuously by sudden long process times

Apart from random peaks, a second problem is that one can observe processing times to have periodic peaks. If the service processing time of frame i is predicted as a peak, then the sending of frame $(i+1)$ should be delayed to prevent a long buffering time. None of the prediction methods mentioned above can deal with random peaks very well, nor do these pay attention to periodic characteristics.

In this paper we propose two policies to amend these particular problems. The first, referred to as the *one-before-last-measurement* (BLM) policy, is to restore the rhythm of transmission by removing the extra delay observed at an earlier moment. The second, referred to as the *peak-prediction* (PP) policy, is to find the periodic characteristics of the peaks in processing times and then to predict occurrence of subsequent peaks. Our proposed prediction methods, including the BLM and PP policies, provides good solutions for our just-in-time communication problem.

IV. Experimental Setup

In a Grid environment, resources have different capacities and many fluctuations exist in load and performance of geographically distributed nodes [19]. As the availability of resources and their load continuously vary with time, the repeatability of the experimental results is hard to guarantee under different scenarios in a real Grid environment. Also, the experimental results are very hard to collect and to observe. Hence, it is wise to perform experiments on a testbed that contains the key characteristics of a Grid environment on the one hand, and that can be managed easily on the other hand. To meet these requirements, we perform all of our experiments on the recently installed DAS-3 (the Distributed ASCI Supercomputer 3) Grid test bed [17].

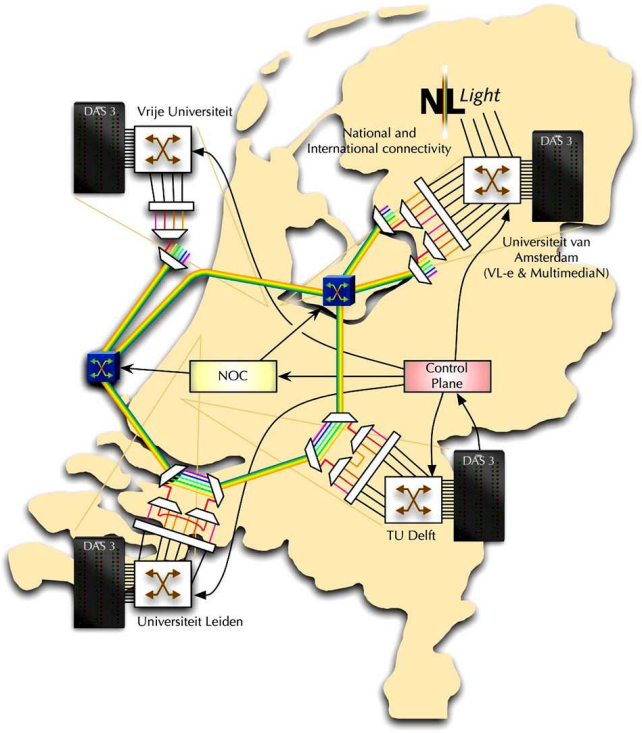


Figure 5. The Distributed ASCI Supercomputer 3 with the StarPlane wide-area optical interconnect.

DAS-3, see Table I and Figure 5, is a five-cluster wide-area distributed system, with individual clusters located at four different universities in The Netherlands: Vrije Universiteit Amsterdam (VU), Leiden University (LU), University of Amsterdam (UvA), and Delft University of Technology (TUD). The MultimediaN Consortium (UvA-MN) also participates with one cluster, located at the University of Amsterdam. As one of its distinguishing features, DAS-3 employs a novel internal wide-area interconnect based on optical 10G links (StarPlane [20]), causing DAS-3 sometimes to be referred to as ”the world’s fastest Grid” [21].

A. Example applications

In our experiments, we use the DAS-3 system to run a real-time multimedia application (referred to as “Aibo”), as well as an off-line application (referred to as “TRECVID”).

The “Aibo” application demonstrates real-time object recognition performed by a Sony Aibo robot dog [22] (see Figure 6).

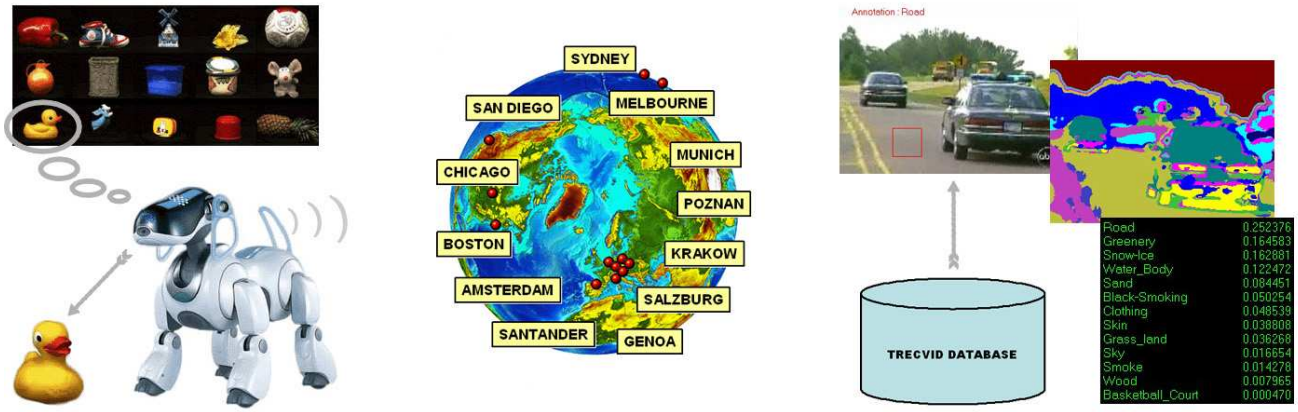


Figure 6. Our example real-time (left) and off-line (right) distributed multimedia applications, which are capable of being executed on a world-wide scale. The real-time application constitutes a visual object recognition task performed by a robot dog (Aibo). The off-line application constitutes our TRECVID system.

Table I
OVERVIEW DAS-3 CLUSTER SITES

Cluster	Nodes	Type	Speed	Memory	Storage	Node HDDs	Network
VU	85 dual	dual-core	2.4 GHz	4 GB	10 TB	85 × 250 GB	Myri-10G and GbE
LU	32 dual	single-core	2.6 GHz	4 GB	10 TB	32 × 400 GB	Myri-10G and GbE
UvA	41 dual	dual-core	2.2 GHz	4 GB	5 TB	41 × 250 GB	Myri-10G and GbE
TUD	68 dual	single-core	2.4 GHz	4 GB	5 TB	68 × 250 GB	GbE (no Myri-10G)
UvA-MN	46 dual	single-core	2.4 GHz	4 GB	3 TB	46 × 1.5 TB	Myri-10G and GbE

Irrespective of the application of a robot, the general problem of object recognition is to determine which, if any, of a given repository of objects, appears in an image or video stream. It is a computationally demanding problem that involves a non-trivial trade-off between specificity of recognition (e.g., discrimination between different faces) and invariance (e.g., to shadows, or to differently colored light sources). Due to the rapid increase in the size of multimedia repositories of 'known' objects [23], state-of-the-art sequential computers no longer can live up to the computational demands, making high-performance computing (potentially at a world-wide scale, see also Figure 6) indispensable.

The "TRECVID" application represents a multimedia computing system that has been applied successfully in the 2004, 2005, and 2006 editions of the international NIST TRECVID benchmark evaluation for content-based video retrieval [24], [25]. The aim of the "TRECVID" application is to find semantic concepts (e.g., vegetation, cars, people, etc.) in hundreds of hours of news broadcasts, a.o., from ABC and CNN. The TRECVID concept detection task is, in general terms, defined as follows: Given the standardized TRECVID video data set, a common shot boundary reference for this data set, and a list of feature definitions, participants must return for each concept a list of at most 2000 shots from the data set, ranked according to the highest possibility of detecting the presence of that semantic concept. Our "TRECVID" application is computationally intensive; for thorough analysis it easily requires about 16 seconds of processing per video frame on the fastest sequential machine at our disposal [26]. Consequently, the required time for participating in the TRECVID evaluation using a single computer easily can take over one year of processing.

Both applications have been implemented using the Parallel-Horus software architecture, that allows programmers to write parallel and distributed multimedia applications in a fully sequential manner [22]. The automatic parallelization and distribution of both applications results in services-based execution: a client program (typically a local desktop machine) connects to one or more *multimedia servers*, each running on a (different) compute cluster. Each multimedia server is executing in a fully data parallel manner, thus resulting in (transparent) *task parallel execution of data parallel services*.

More specifically, in both applications, before any processing takes place, a connection is established between the client application and a multimedia server. As long as the connection is available, the client can send video frames to this server. Every received video frame is scattered by this server into many pieces over the available compute nodes. Normally, each compute node receives one partial video frame for processing. The computations at all compute nodes take place in parallel. When the computations are completed, the partial results are gathered by the communication server again and the final result is returned to the client. In this paper, the time to process a single video frame in this manner is defined as the *service processing time* Ts. The individual values of Ts_i are collected as data source for a trace-driven simulation. In our simulation, the service utilization and total waiting times are calculated by using different prediction methods in combination with our BLM and PP policies.

V. METHOD FORMULATION

This section describes our newly proposed modeling approaches in detail. The approaches are based on the results of extensive experimentation performed on DAS-3 (see Section IV).

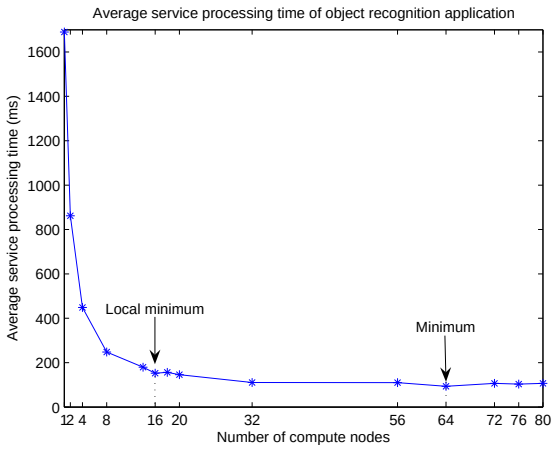
A. On resource-optimization problem

In our example applications, video frames are being processed on a per-cluster basis, using a varying number of compute nodes on each cluster, each consisting of multiple CPUs. The compute cluster (or *service*) processing time is defined as a function $S(L,n)$ of the number of compute nodes $L = 1,\ldots,m_{max}$ and the number of CPUs per node $n = 1,2,\ldots$. Our goal is to minimize the cost function $S(L,n)$ over the set of possible values of (L,n); thus, we are searching for the point $(\hat{L},\hat{n})$ where the function $S(L,n)$ attains its minimum. In this context, it is important to note that the set of possible combinations (L,n) may be very large, and that in practice, finding the optimum $(\hat{L},\hat{n})$ may be very time consuming. Therefore, our goal is to develop a simple but effective heuristic method to obtain a nearly-optimal solution within a short time frame. To this end, we first discuss a number of observations that we collected during our extensive experiments, leading to a dramatic reduction of the set of possible value of (L,n). Subsequently, the method to approach the optimal (L,n) is described in detail.

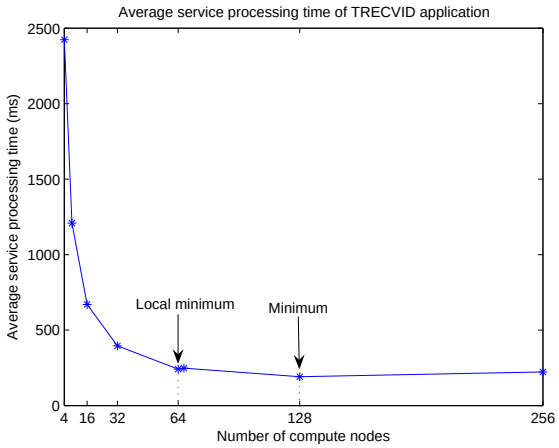
1) Reduction of solution space: Many combinations (L,n) lead to the same total number $T = L \cdot n$ of CPUs. The following observations, made for our particular example applications, rule out many possibilities:

a) The optimal number of CPUs often is a power of 2: In our experiments, we consistently observed that the optimal number of CPUs is found to be a power 2. For example, Figure 7 shows the average processing times for our two example applications: (a) Aibo, and (b) TRECVID. The results show that both *local* and *global* minima are consistently found when the total number of CPUs is a power of 2. This observation leads to a dramatic reduction of the set of possible solutions. Namely, if the number of available compute nodes is L_{max}, the number of available CPUs in each compute node is n_{max}, then the solution set is reduced to $\mathbb{X} := \{(2^p, 2^q), p = 0,\ldots,P, q = 0,\ldots,Q\}$, where $P := \lfloor \log_2(L_{max}) \rfloor$, $Q := \lfloor \log_2(n_{max}) \rfloor$.

b) Using more compute nodes, yet less CPUs per node, is generally better: Another important observation from our experimental results is that for the same total number of CPUs $T = L \cdot n$, using more compute nodes L and fewer CPUs per node n provides better performance. That is, for the same total number of CPUs $T = 2^m$, where the solution set should be $\mathbb{X} := \{(2^p, 2^q), p + q = m\}$, among them, q should be as small as possible. This observation is illustrated by Figure 8, where we consider the case $T = 64$ for three combinations $(L,n) \in \{(64,1),(32,2),(16,4)\}$; the results show that the combination (64,1) has better performance than (32,2) and (16,4). This is explained by the fact that the compute nodes in DAS-3 are linked by a fast local Myrinet interconnect, whereas the CPUs within a single node communicate over a shared memory bus, which is less efficient. Note that the 'burstiness' of the perceived processing times is explained by random operating system interference, and by automatic garbage collection performed by the Java virtual machine.



(a) Aibo application



(b) TRECVID application

Figure 7. Average service processing time v.s. number of compute nodes.

Based on these observations, the solution set can be reduced drastically. For instance, for a system having 85 nodes and 4 CPUs per node, the reduced solution space is $\mathbb{X} = \{(2^p,1), p = 0\ldots6\} \cup \{(64,2),(64,4)\}$.

In a general form, to determine the optimal number of compute nodes and CPUs per node, the solution space is reduced to the combinations $\mathbb{X} = \{(2^p,1), p = 0\ldots P\} \cup \{(2^P, 2^q), q = 1\ldots Q\}$, where $P := \lfloor \log_2(L_{max}) \rfloor$, $Q := \lfloor \log_2(n_{max}) \rfloor$. For simplicity, we use $(2^{(P+q)},1)$ instead of $(2^P, 2^q)$ for our

notation, although $(2^{(P+q)}, 1)$ does not exist.

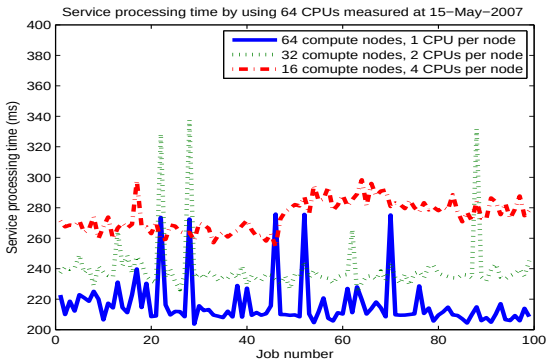
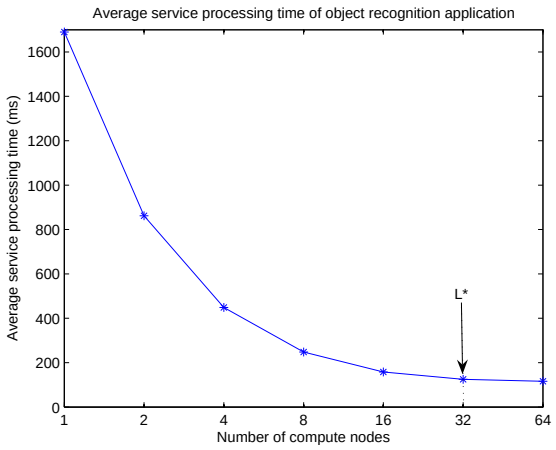


Figure 8. More compute nodes and less CPUs per node is better.

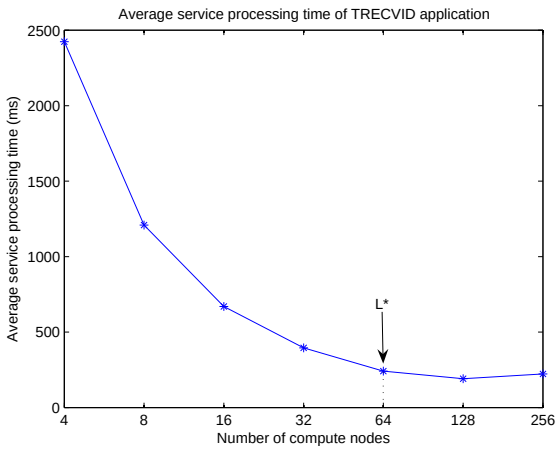
2) Steps to approach the optimal (L, n)*:* From the reduced solution space, we iteratively increase the total number of CPUs to find the optimal (L, n). When the number of applied compute nodes becomes larger, the parallelization overhead increases, and may even become dominant. Our experimental results show that there exists a threshold value m^* such that $S(2^m, 1)$ decreases fast for $m < m^*$, whereas $S(2^m, 1)$ flattens out, and may even increase, for $m > m^*$. As an illustration, Figure 9 shows the average service processing times for the Aibo- and TRECVID-applications for different values of $L = 2^m$. In both cases, we observe that there exists some *saturation point* $L^* = 2^{m^*}$ such that increasing the number of parallel nodes L beyond L^* does not lead to a significant reduction of the service processing times. Throughout, $L^* = 2^{m^*}$ will be referred to as the *engineering knee* and is regarded as the (near-)optimal point of operation.

3) LDS method: To find the engineering knee L^*, we have developed an *Logarithmic dichotomy search* (LDS) method. The LDS method follows the idea of a well-known conventional binary search (CBS) algorithm [27] which aims to find a particular value in a sorted list. Compared to the CBS strategy, the LDS method makes progressively better guesses, and proceeds closer to the optimal value. Let the elements in the solution set $\mathbb{X}$ be denoted by $(e_0, \ldots, e_K)$, with $K = P+Q$, P and Q are defined in Section V-A1b. The LDS strategy selects the median element in the set $\mathbb{X}$, denoted by e_{Mid}. Define ϵ as the desired improvement in the service processing time by increasing the number of compute nodes. If $\frac{S(e_{\text{Mid}})-S(e_{\text{Mid+1}})}{S(e_{\text{Mid}})} > \epsilon$, then we repeat this procedure with a smaller list, and we keep only the elements $(e_{\text{Mid+1}}, \ldots, e_K)$. If $\frac{S(e_{\text{Mid}})-S(e_{\text{Mid+1}})}{S(e_{\text{Mid}})} \leq \epsilon$ then the list in which we search becomes $(e_1, \ldots, e_{\text{Mid}})$. Pursuing this strategy iteratively, it narrows the search by a factor of two each time, and finds the minimum value that satisfies our requirement after $\log_2(K)$ iterations. The pseudo code for our LDS method for the solution space $\mathbb{X}$ is given in Algorithm 1.

It is worth noting that there is still room for improvement. In our implementation, we obtain the runtime information using individual number of compute nodes by sequential measurements. Actually, if there are enough processors, we can do several measurements simultaneously by parallel technique. It is named as parallel LDS strategy. In this paper, we will not give further discussion on this possible improvement.



(a) Aibo application.



(b) TRECVID application

Figure 9. Engineering knee of Aibo and TRECVID applications.

B. On "just-in-time" communication problem

Our real-time multimedia "Aibo" application is run to generate data that are used in our trace-driven simulation for validating our approach.

The notations used here are defined as follows.

- Ts_i: the processing time of the i-th frame.
- Tc_i: the communication time of sending the i-th frame from the client to the server.
- t_i: the time point when the client sends the i-th frame to the server.
- r_i: the time point when the client receives i-th frame from the server.

1) Preliminary:

Low := 0
High := K
While (Low $<$ High) {
Mid := $\left\lfloor \frac{\text{Low+High}}{2} \right\rfloor$
if $S(e_{\text{Mid}}) \leq \frac{S(e_{\text{Mid+1}})}{1-\epsilon}$ {High = Mid;}
else {Low = Mid +1;}
end if;
}
Optimal number of compute nodes := High.

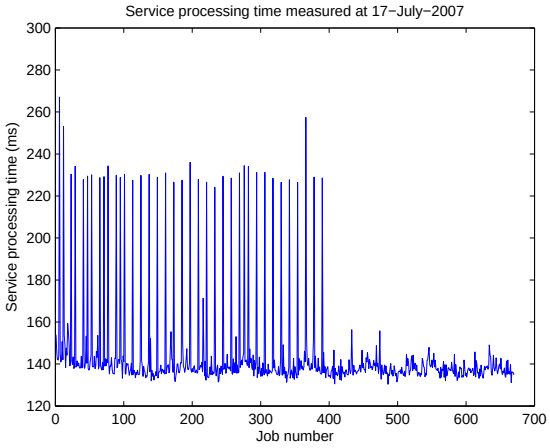
Algorithm 1: Pseudo code of LDS strategy.

a) Trend line: As shown in Figure 3, if we can predict the service processing time of the current frame accurately, then sending the next frame after the predicted time unit should provide an optimal solution. Therefore we investigated several conventional prediction methods (i.e., adapted mean-based methods, adapted median-based methods, exponential smoothing methods, and Robbins-Monro Stochastic Approximation methods) for predicting the service processing time. We found that, based on the earlier service processing times, and by using any of these prediction methods, an accurate trend line can be generated. Figure 10 gives an illustration of the predicted service processing time versus the measured value of running the Aibo application using one compute node, using a single CPU only.

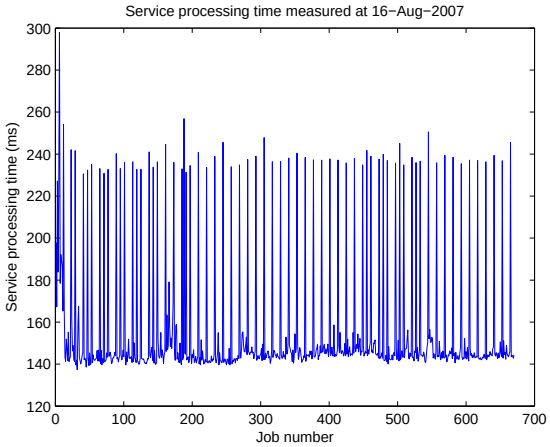
b) Periodicity of the peaks: Another important observation from our experimental results is the occurrence of periodic peaks using large numbers of compute nodes. Because our multimedia application is partially implemented in Java, the *Java garbage collector* [28] has an influence on the service processing time. In case of large service processing times, the effect of garbage collection generally is insignificant and can be ignored. This is the situation as depicted in Figure 10. In contrast, when the service processing time is small compared to the garbage collection time, the periodic peaks are significant. We ran the Aibo application using 64 compute nodes (using one CPU per node) on three different moments in time. From these data sets, we notice that there is a deterministic period of the occurrences of certain specific peaks (see Figure 11).

2) Method: Based on the experimental results, we conclude that an effective prediction method for our application must have the following characteristics: (1) it must be able to generate an accurate trend line of the service processing time, (2) it should be able to deal with outliers in the observed processing time as soon as possible, and (3) it must be able to predict when the next peak occurs. In this section, we discuss the applied prediction methods and our BLM and PP policies in detail.

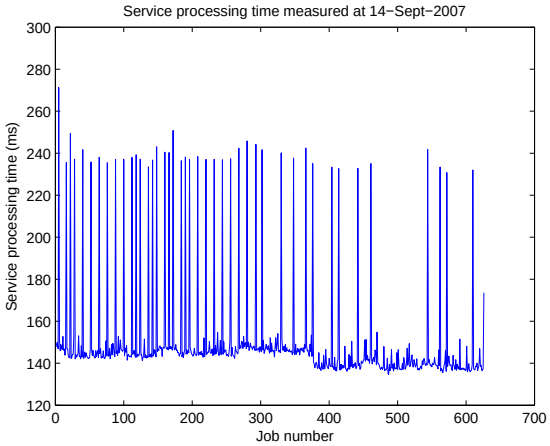
a) Prediction methods: Among existing predictive methods there is a huge difference in the way previously obtained data are handled. In some cases one wants to adapt very



(a) 17-July-2007

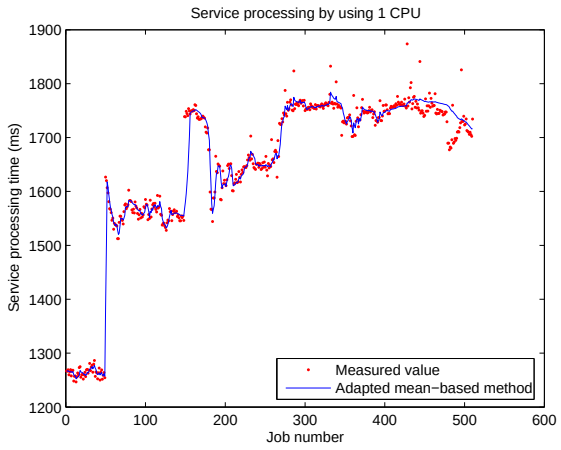


(b) 16-August-2007

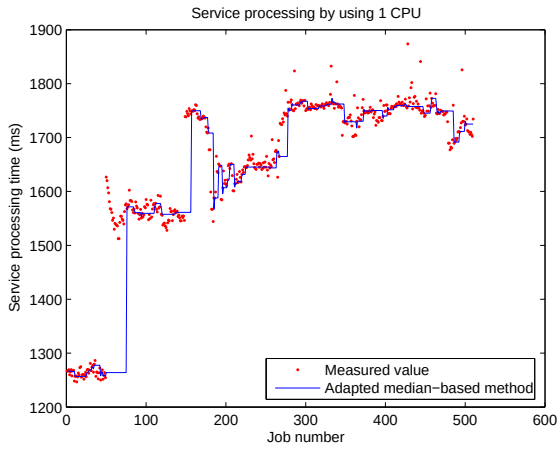


(c) 14-September-2007

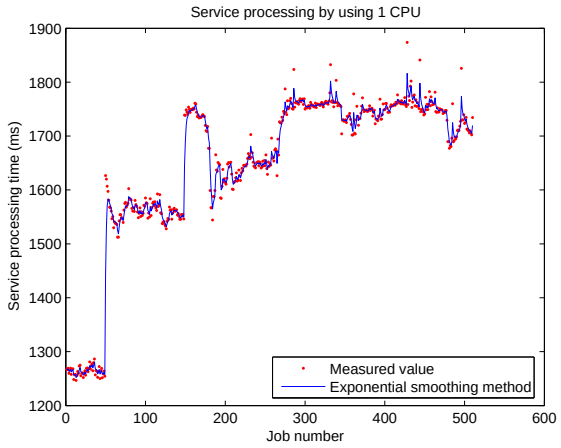
Figure 11. Service processing time taken at different times



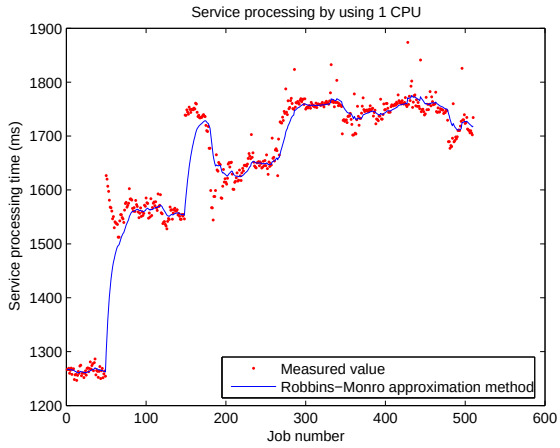
(a) adapted mean-based method



(b) adapted median-based method



(c) exponential smoothing method



(d) Robbins-Monro approximation method

Figure 10. Trend line generated by different prediction methods

quickly to observed changes in the data, while there are also cases in which this behavior is not desired. The adapted mean-based method [11] uses arithmetic averages over some portion of the measurement history to predict the next measurement. In particular, the extent of the history taken into account depends on a parameter K, specifying the number of previous measurements for the arithmetic average. The parameter K is changed by -1, 0, or $+1$ over time based on the prediction error. In our experiments, the initial value of K is set to 20.

Adapted median-based methods [11] use a portion of the measurement history defined by the parameter K to calculate the median which is used for the prediction. The parameter K is adapted in the same way as in the mean-based method above. Note that the prediction of this method is not influenced much by asymmetric outliers (e.g., a peak in the processing time), since this does not affect the median greatly.

In exponential smoothing [12]–[15] earlier measurements are not weighted equally as in the case of a mean-based method, but with exponentially decreasing weights as the measurements get older. More specifically, denote by $w(i)$ the weight for the i-th previous measurement. Then, w is the following function

$$w(i) = \alpha(1-\alpha)^i,$$

with α a parameter determining the rate of decay of the function. In our experiments, we set $\alpha = 0.5$. As in the previous methods, the parameter K determines the number of earlier measurements that we intend to use. In case $K > \{\#\text{ available previous measurements}\}$ and in case $K < \infty$ we made sure, by scaling of the weights, that the sum of the weights used sum up to 1.

The Robbins-Monro approximation method [16] is a stochastic approximation method. If we denote by $\hat{Ts}_i$ the

estimation of the i-th processing time, then the estimation is updated according to the following relation

$$\hat{Ts}_{i+1} = \hat{Ts}_i + \varepsilon_i(Ts_i - \hat{Ts}_i),$$

where ε_i is a parameter possibly depending on i. The intuition behind the update rule is the following. In case the observed processing time is higher than estimated, the prediction for the next processing time is increased by a small amount of the difference, and vice versa. When $\varepsilon_i = 1$ for all i, then the prediction for the next processing time is equal to the last observation. We set $\varepsilon_i = 0.5$ for our experiments.

b) BLM Policy: Our first policy to deal with peaks is called "one-before-last-measurement" (BLM) policy. This policy follows the following steps.

- The i-th job will not be sent until the result of the $(i-k)$-th job becomes available to the client. Because we must take care that the server has enough jobs to process, we can not use the last measurement data as a predictor (also indicated by Harchol-Balter and Downey [29]). Therefore k must be larger or equal to 2. Throughout this paper, we focus on the case that $E[Tc] \leq \frac{E[Ts]}{2}$. Here $E[Ts]$ and $E[Tc]$ represent the expected service processing time and the communication time respectively. In this case, we set $k = 2$. This implies that at most one job is waiting in the buffer at the server side. As a result, the occurrence of cumulative waiting times can be prevented. In the case that $Tc > \frac{E[Ts]}{2}$, we only need to enlarge the value of k. Hence, for $k = 2$, we have the following equation,

$$t_i \geq r_{i-2}. \tag{1}$$

  This equation implies that the i-th video frame is sent after the result of the $(i-2)$-th frame is received by the client. Figure 12 gives an illustration.
- Obviously, if the result of the $(i-1)$-th frame is received, the i-th frame must be sent immediately. Therefore, we have

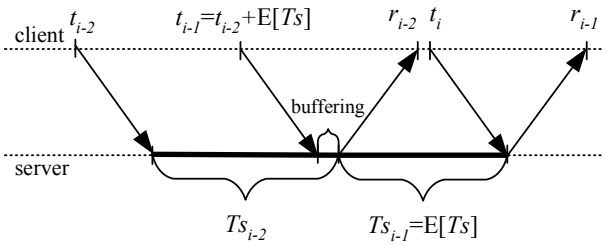
$$t_i \leq r_{i-1}. \tag{2}$$

- The sending time of the i-th frame is also decided by the relationship between the expected service processing time and measured service processing time of the $(i-2)$-th frame Ts_{i-2}. If $Ts_{i-2} > E[Ts]$, then it is optimal to send the i-th frame at $r_{i-2} + E[Ts] - 2 \cdot E[Tc]$. Figure 12(a) gives an example. In case $Ts_{i-2} \leq E[Ts]$, the optimal sending moment is at $t_{i-1} + E[Ts]$. See Figure 12(b). Hence we get the following equation,

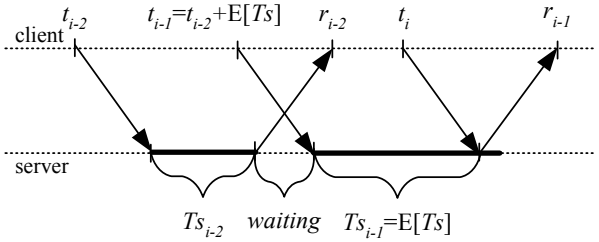
$$t_i = \begin{cases} r_{i-2} + E[Ts] - 2 \cdot E[Tc] & \text{if } Ts_{i-2} > E[Ts], \\ t_{i-1} + E[Ts] & \text{otherwise.} \end{cases} \tag{3}$$

  Note that using the receiving time of the $(i-2)$-th frame to determine the sending time of i-th frame indirectly takes into account the variation of the communication time between the client and the server. Therefore, the assumption $Tc_1 = Tc_2$ is not necessary any longer. Combining Equations 1, 2, and 3, the optimal sending time of i-th frame is given by

$$t_i = min(r_{i-1}, max(r_{i-2}, t_{i-1} + E[Ts], r_{i-2} + E[Ts] - 2E[Tc])). \tag{4}$$



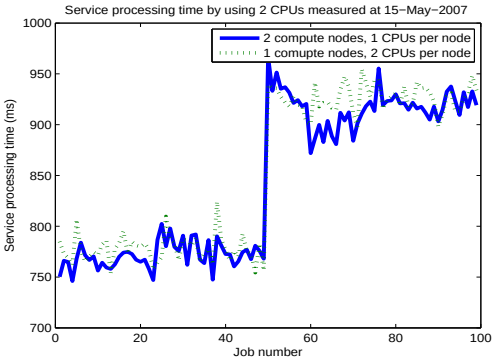
(a) The optimal sending time in case of $TS_{i-2} > E[Ts]$



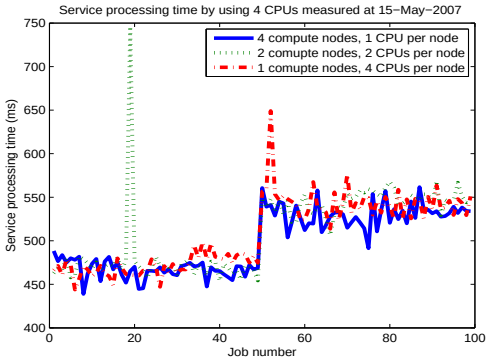
(b) The optimal sending time in case of $TS_{i-2} \leq E[Ts]$

Figure 12. BLM Policy

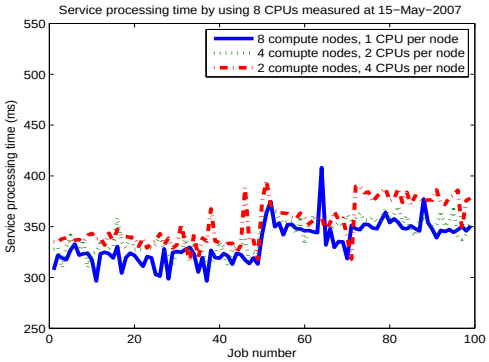
c) PP Policy: Our second method, called peak-policy, tries to predict the next outlier based on historical observations. We define an outlier (i.e., a peak) as significantly different from the average processing time if the observation is much larger than the average (say 1.2 times larger). Based on the occurrences of peaks in the previous observations, we try to predict when the next peak will occur. Motivated by experiments, we observe that there is a deterministic period of the occurrences of peaks. See Figures 11(a), 11(b), and 11(c) for the experimental results. Denote $P = \{i | Ts_i \text{ is peak}\}$ as the set of peaks and denote by p_j the j-th element of P. Let k be an integer number. If $p_j - p_{j-1} = \cdots = p_{j-(k+1)} - p_{j-k}$ then we say that there is a deterministic period of length $d = p_j - p_{j-1}$, and we expect the next peak to occur at job number $j + d$. Note that k defines the number of previous peaks that should have occurred equidistantly with length d such that we consider the peaks as periodical events. The optimal k is not known beforehand. Therefore, we will start with an arbitrary value and adjust it as time evolves. Suppose that $k = 3$, and we observe three peaks each having distance d, then the method predicts that the next peak occurs after processing of d frames. If it turns out that the prediction is wrong, then we increase k by 1, since probably $k = 3$ was too low. In case the prediction is correct, then we decrease k



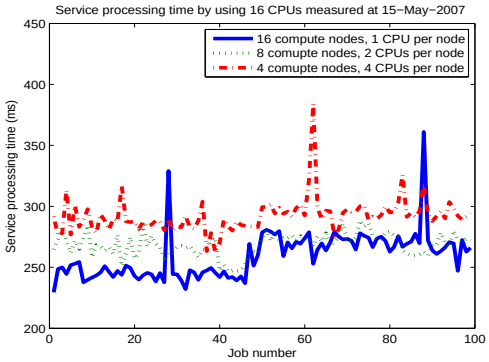
(a) 2 CPUs



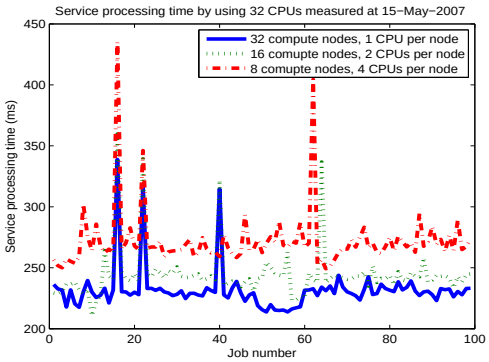
(b) 4 CPUs



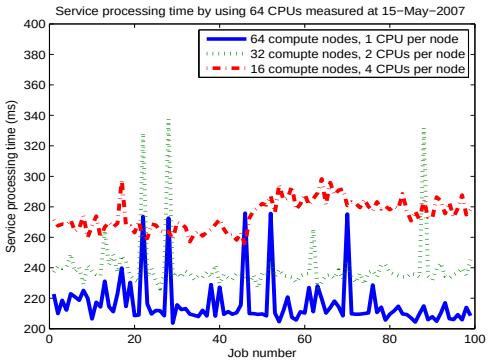
(c) 8 CPUs



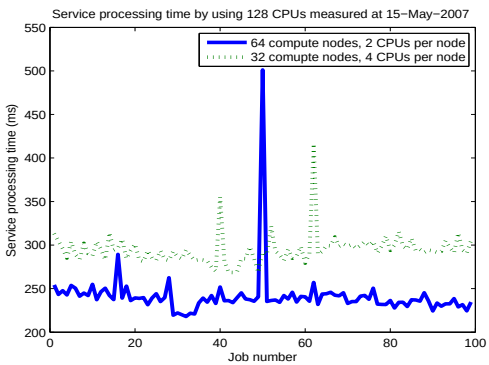
(d) 16 CPUs



(e) 32 CPUs



(f) 64 CPUs



(g) 128 CPUs

Table II
AVERAGE SERVICE PROCESSING TIME OF TRECVID APPLICATION.

(L, n)	(16, 1)	(8, 2)	(4, 4)	(64, 1)	(32, 2)	(16, 4)	(64, 2)	(32, 4)
$S(L, N)$ *ms*	669.28	682.44	736.56	241.62	244.90	263.01	190.70	218.27

by 1, such as to try a smaller number. To prevent meaningless values for k, we restrict k to be in $[3, \infty)$.

By combining the BLM and PP policies with one of the prediction methods to predict service processing times, we obtain our final model to deal with the just-in-time communication problem in real-time applications.

VI. NUMERICAL RESULTS

In this section we present the results of our experiments performed on the DAS-3 system. The detailed experiments results are achieved on the largest cluster at the Vrije Universiteit, that consists of 85 compute nodes with 4 CPUs per node.

A. On resource-optimization problem

Here, we show the numerical results of the average service processing times versus a varying total number of compute nodes. In addition, the simplicity of LDS strategy to determine the optimal number of compute nodes is validated.

First, denote the possible solution space of the compute nodes and the number of CPUs per node as $\mathbb{O}$, where $\mathbb{O} = \{(L, n), L \in [1, \ldots, 85] \text{ and } n \in [1, \ldots, 4]\}$. To show that using more compute nodes and less CPUs per node provides better performance in general, we ran our real-time "Aibo" application on a varying numbers of CPUs (2, 4, 8, 16, 32, 64, and 128 CPUs). We compared the obtained service processing times for a fixed total number of CPUs, while varying the number of CPUs per nodes. The results are shown in Figure 13. In this figure we notice that for small numbers of CPUs (say, ≤ 16), the service processing time is largely independent of the ratio between the total number of employed CPUs and the number of employed CPUs per node. As the number of CPUs increases, it becomes obvious that wider distribution of the CPUs, that is, using less CPUs per node and more compute nodes, provides better performance.

We also compared the service processing time for our offline "TRECVID" application, on a varying total number of CPUs (16, 64 and 128 CPUs). The results are tabulated in Tabel II. For this application we have a similar conclusion: more compute nodes and less CPUs per node provides the best performance results.

In Paragraph V-A1a, we mentioned that the optimal number of compute nodes is consistently found to be a power of 2. Combining this result and the observations above, we reduced the original space $\mathbb{O}$ with $85 \times 4 = 340$ possible solutions to the space $\mathbb{X}$ with 9 possible solutions, where $\mathbb{X} = \{(2^i, 1), i \in [0, \ldots, 6]\} \cup (64, 2) \cup (64, 4)$. Based on $\mathbb{X}$, we apply our LDS method to find the minimum value after $\lfloor \log_2 9 \rfloor = 3$ steps. We use Table III to explain the three steps taken in AIBO application when ϵ = 0.1. We continue to approach the optimal number of compute nodes L^* by doubling the total number of compute nodes, until the relative improvement is less than 10%. Here the index of the elements of $\mathbb{X}$ is denoted as $[0, 1, \ldots, 8]$. Then the LDS method is applied. In the first step, we have $Low = 0$ and $High = 8$, and thus

$$Mid = \left\lfloor \frac{Low + High}{2} \right\rfloor = 4.$$

Therefore, we measure the service processing time using $2^4 = 16$ and $2^5 = 32$ compute nodes and 1 CPU per node. The related average service processing times are shown in the first row of Table IV. Because the relative improvement using 32 compute nodes compared to 16 compute nodes is 0.27 ($> \epsilon$), we conclude that 16 compute nodes is not optimal. Therefore, we continue searching for the optimal. In the second step, the index value 5 (= 32 compute nodes) is set as the value of Low. The value of High remains the same. Therefore $Mid = 6$. When calculating the relative improvement using 64 compute nodes and 2 CPUs per node compared to 2^6 compute nodes, we find that the improvement (-0.15) is less than ϵ. Therefore, in the third step, the value of High is reset to 6, and Low remains the same. In this case, $Mid = 5$. The improvement of using 2^6 compute nodes compared to 2^5 is more than ϵ. Thus, Low is reset to 6, such that Low is equal to High, and the whole procedure is finished. The LDS method returns index 6 as the optimal solution. This means, for $\epsilon = 0.1$, the optimal number of CPUs is $2^6 = 64$ compute nodes.

For different ϵ (0.1, 0.2 and 0.3), the (L, n) to be evaluated and the corresponding average service processing time of both applications are reported in Table IV and Table V, respectively. The optimal L^* that we found for both applications for different values of ϵ are listed in Table VI. In this table, we notice that with larger ϵ, the L^* remains the same or decreases.

Table III
THREE STEPS TO APPROACH THE OPTIMAL (L, n).

Step	Low	High	Mid	compare to	relative improvement	Action
1	0	8	4	5	0.27	keep high half
2	5	8	6	7	-0.15	keep low half
3	5	6	5	6	0.15	finish, return index 6

Table IV
AVERAGE SERVICE PROCESSING TIME OF AIBO APPLICATION.

$\epsilon = 0.1$	(L, n)	(16, 1)	(32, 1)	(64, 1)	(64, 2)
	$S(L, N)$ *ms*	152.26	110.64	93.58	108.55
$\epsilon = 0.2$	(L, n)	(16, 1)	(32, 1)	(64, 1)	(64, 2)
	$S(L, N)$ *ms*	152.26	110.64	93.58	108.55
$\epsilon = 0.3$	(L, n)	(4, 1)	(8, 1)	(16, 1)	(32, 1)
	$S(L, N)$ *ms*	448.57	247.72	152.26	110.64

As shown above, we notice that our method is very simple to implement. Besides this, it is very effective because of the

Table V
AVERAGE SERVICE PROCESSING TIME OF TRECVID APPLICATION.

$\epsilon = 0.1$	(L, n)	(16, 1)	(32, 1)	(64, 1)	(64, 2)	(64, 4)
	$S(L, N)$ *ms*	669.28	395.79	241.62	190.70	222.61
$\epsilon = 0.2$	(L, n)	(16, 1)	(32, 1)	(64, 1)	(64, 2)	(64, 4)
	$S(L, N)$ *ms*	669.28	395.79	241.62	190.70	222.61
$\epsilon = 0.3$	(L, n)	(16, 1)	(32, 1)	(64, 1)	(64, 2)	
	$S(L, N)$ *ms*	669.28	395.79	241.62	190.70	

Table VI
THE VALUE OF ENGINEERING KNEE.

ϵ	L^*
0.1	64 (64,1)
0.2	32 (32,1)
0.3	16 (16,1)

(a) Aibo application

ϵ	L^*
0.1	128 (64,2)
0.2	128 (64,2)
0.3	64 (64,1)

(b) TRECVID application

small number of steps required to find the optimal number of compute nodes. In addition, by varying ϵ, we are able to obtain the optimal result related to the desired improvement in the service processing time by increasing the number of compute nodes.

B. On "just-in-time" communication problem

In this section we present the results of our experiments performed on the DAS-3 system. The results are also used as the input for a trace-driven simulation in order to validate our final model for determining the exact transmission moments of video frames. In our experiments, the object recognition application is ran on 64 compute nodes using 1 CPU per node.

First, we apply the BBM method (Figure 1) to our Aibo application. In our experiment, we found that the average service processing time ($E[Ts]$) and the average communication time ($E[Tc]$) between client and server amount to 143.629 *ms* and 11.694 *ms*, respectively. In this case, the server utilization is about 85%, and the average waiting time per frame is 0. Consider that the service utilization using the BBM method is given by $E[Ts]/(E[Ts] + 2 \cdot E[Tc])$. This implies that when Tc is negligible, the BBM method approaches the optimal strategy. However, in a bottleneck situation where $E[Tc]$ is long relative to $E[Ts]$, the BBM method performs badly.

Server utilization can be increased by sending frames with smaller intervals. However, if a sudden change (a peak) in service processing time takes place, all incoming frames are affected. A particularly difficult situation is when a series of long service times occurs, such that the waiting time of frames increases rapidly due to the accumulation of perceived gaps. In our experiments, we used simulation to evaluate the impact of changing the time interval between sending subsequent frames. The time interval is reduced in 5 steps according to Table VII. $E[Ts]$ and $E[Tc]$ in Table VII are adjusted by one of the prediction methods. Since Figure 10 shows that all prediction methods are capable of generating accurate trend lines, in this paper, we only choose one of these (i.e. the exponential smoothing method) as a representative prediction method. In Figure 14, it is shown that the average waiting time increases significantly as the service utilization approaches 100%. Hence, the prediction methods are not sufficient for our just-in-time communication problem.

Table VII
TIME INTERVAL BETWEEN SENDING TWO SEQUENTIAL FRAMES.

Simulation index	Time interval
1	Ts_{BBM}
2	$2E[Tc] + E[Ts]$
3	$1.5E[Tc] + E[Ts]$
4	$E[Tc] + E[Ts]$
5	$0.5E[Tc] + E[Ts]$
6	$0.375E[Tc] + E[Ts]$
7	$0.25E[Tc] + E[Ts]$
8	$E[Ts]$

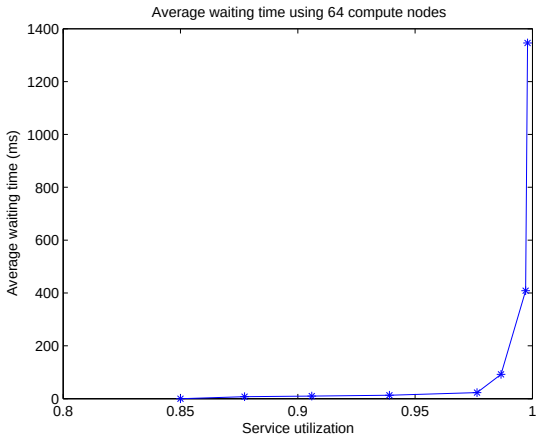


Figure 14. Average waiting time using 64 compute nodes

In our final model, in which one of the prediction methods is combined with the BLM and PP policies, we can achieve high service utilization while keeping the average waiting time low. By using the exponential smoothing method with our policies, we obtain service utilization of about 98%, and an average waiting time per frame of around 7 *ms*. If we define the waiting time percentage (WP) as

$$WP = \frac{\text{total waiting time}}{\text{total waiting time + total service processing time}},$$

then we obtain a WP of around 3.5%. Because of the lower value of WP, we can compare the performance of our final model to the BBM method by looking at the service utilization. Define the gain in service utilization $Gain(SU)$ as follows,

$$Gain(SU) = \frac{\text{service utilization using final model}}{\text{service utilization using BBM method}}. \quad (5)$$

Figure 15 shows the gain of our final model related to the BBM method for different values of $\frac{Tc}{Ts}$.

In this figure, we notice that the gain in utilization is almost linear in $\frac{Tc}{Ts}$. This can be explained by the fact that the service utilization in the final model is very close to 1 and

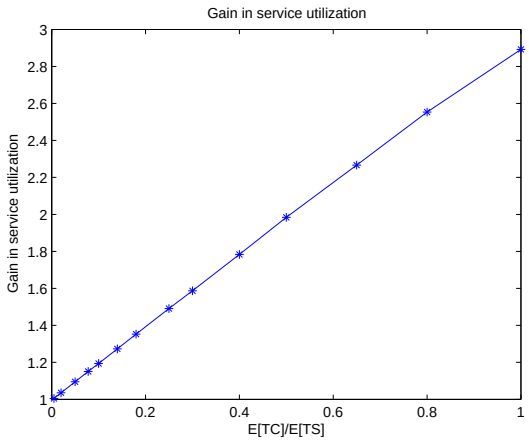


Figure 15. Gain in the service utilization

the service utilization belonging to the simple strategy can be approximated by $E[Ts]/(E[Ts]+2\cdot E[Tc])$. Hence, based on Equation 5, we have

$$\text{Gain(SU)} \approx \frac{1}{Ts/(Ts+2\cdot Tc)} = 1+2\frac{Tc}{Ts}.$$

Therefore, the gain in the service utilization is nearly increasing linearly with Tc/Ts.

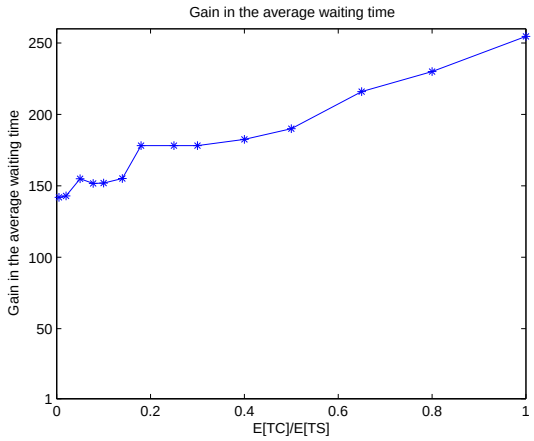


Figure 16. Gain in the average waiting time

The last comparison is done to evaluate the benefit brought by our policies. For the prediction method of exponential smoothing, we compare the performance of our final model to the prediction method by looking at the average waiting time. Define the gain in the average waiting time $Gain(w)$ as follows,

$$\text{Gain(w)} = \frac{\text{average waiting time using the prediction method}}{\text{average waiting time using final model}}.$$

The results of this comparison are shown in Figure 16. The reason why the final model can gain so much, can be explained by the following example. Assume that during processing, only one peak takes place and that, after that peak, there are still 100 frames to be processed. In this situation the use of prediction methods causes all following 100 frames to be delayed by the peak. But using our final model, there is only 1 following frame affected by the peak. Thereafter, the sending times of the next 99 frames are corrected. Thus no error accumulation occurs. Therefore, we conclude that our final model, incorporating BLM and PP, are indispensable and effective for just-in-time communication.

VII. Conclusions

In this paper we explored firstly the relation between the service processing time of distributed multimedia applications and the number of compute nodes for a varying number of CPUs. We observed that there exists a threshold value L^* (referred to as the *engineering knee*) such that the service processing time decreases fast as a function of L for $L < L^*$, whereas the service processing time flattens out, and may even increase, for $L > L^*$. To find L^*, first we reduce the possible solution set. Then we apply our LDS method to find L^*. Extensive validation has shown that our method is fast and effective.

Specifically, we have found that our method can find optimal resource utilization for an average sized cluster system in no more than three evaluation steps. As a result, we conclude that our method adheres to all requirements as stated in the introduction: it is simple, easily implementable, and effective. In addition, our method takes into account system variation. Even though our focus was on the MMCA domain, our approach is general enough to be applicable in other domains as well.

For "just-in-time" communication problem, we have explored that we have to trade off between high service utilization and short service response time. Using a BBM method, the waiting time is zero. However, service utilization decreases when the communication time between client and server increases. By applying existing prediction methods to this problem, service utilization can be increased. However, at the same time, the average waiting time of video frames increases even faster. This can be explained by the fact that existing prediction methods do not pay attention to peaks in the service processing time. For this reason, we have developed two innovative policies, BLM and PP. Using the first policy, cumulative waiting times are avoided by postponing transmission of a new job when a peak is detected. The second policy is used to predict possible peaks. If we can predict the moment when a peak occurs, then we can send new jobs at the right time. Combining these two policies with any of the existing prediction methods described in this paper, we achieve our final model to solve the just-in-time communication problem.

Our final model is validated in our experiments. Moreover, we have extensively investigated the gain of our final model related to the BBM method, as well as the prediction methods without incorporating our newly developed policies. From our experimental results we conclude that our final model significantly outperforms the other methods. Specifically, we have observed that, in comparison to other methods, our final

model improves server utilization from 85% to 98%, and reduces the average waiting time per frame by factor 250.

REFERENCES

[1] R. Yang, R. van der Mei, D. Roubos, F. Seinstra, and G. Koole, "On the Optimization of Resource Utilization in Distributed Multimedia Applications," in *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, 2008, pp. 358–365.

[2] R. Yang, R. van der Mei, D. Roubos, F. Seinstra, G. Koole, and H. Bal, "Performance model for "Just-in-Time" Problem in Real-Time Multimedia Applications," pp. 518–525, 2008.

[3] C. G. Snoek, M. Worring, J.-M. Geusebroek, D. C. Koelma, F. J. Seinstra, and A. W. Smeulders, "The semantic pathfinder: Using an authoring metaphor for generic multimedia indexing," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 28, no. 10, pp. 1678–1689, 2006.

[4] online, "http://privacy.cs.cmu.edu/dataprivacy/projects/explosion/index.html," 2009.

[5] R. Saavedra-Barrera, A. Smith, and E. Miya, "Machine characterization based on an abstract high-level language machine," *IEEE Trans. Computers*, vol. 38, no. 12, pp. 1659–1679, 1989.

[6] C. Grelck, "Array Padding in the Functional Language SAC," *in Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, vol. 5, pp. 2553–2560, 2000.

[7] K. Schutte and G. van Kempen, "Optimal cache usage for separable image processing algorithms on general purpose workstations," *IEEE Trans. Signal Process*, vol. 59, no. 1, pp. 113–122, 1997.

[8] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[9] B. Maggs, L. Matheson, and R. Tarjan, "Models of parallel computation: a survey and synthesis," *in Proc. International Conference on System Sciences*, vol. 2, pp. 61–70, 1995.

[10] F. Seinstra, D. Koelma, and J. Geusebroek, "A software architecture for user transparent parallel image processing," *Parallel Computing*, vol. 28, no. 7-8, pp. 967–993, 2002.

[11] R. Wolski, "Forecasting network performance to support dynamic scheduling usingthe network weather service," *in Proc. International Conference on High Performance Computing (HiPC)*, pp. 316–325, 1997.

[12] R. Brown, *Statistical forecasting for inventory control*. McGraw-Hill New York, 1959.

[13] ——, *Smoothing, Forecasting and Prediction of Discrete Time Series*. Prentice-Hall, 1963.

[14] C. Holt, "Forecasting Trends and Seasonals by Exponentially Weighted Moving Averages," *ONR Memorandum*, vol. 52, 1957.

[15] P. Winters, "Forecasting Sales by Exponentially Weighted Moving Averages," *Management Science*, vol. 6, no. 3, pp. 324–342, 1960.

[16] H. Kushner and G. Yin, *Stochastic Approximation and Recursive Algorithms and Applications*. Springer-Verlag, 2003.

[17] online, "http://www.cs.vu.nl/das3/," 2007.

[18] ——, "http://www.asci.tudelft.nl," 2007.

[19] M. Dobber, G. Koole, and R. van der Mei, "Dynamic Load Balancing for a Grid Application," *in Proc. International Conference on High Performance Computing (HiPC)*, vol. 1, pp. 342–352, 2004.

[20] online, "http://www.starplane.org/," 2007.

[21] M. Feldman, "Grid Envy," *ClusterVision News*, pp. 6–7, 2006.

[22] F. Seinstra, J. Geusebroek, D. Koelma, C. Snoek, M. Worring, and A. Smeulders, "High-Performance Distributed Image and Video Content Analysis with Parallel-Horus," *IEEE Multimedia*, vol. 14, no. 4, pp. —, 2007.

[23] J. Geusebroek, G. Burghouts, and A. Smeulders, "The Amsterdam library of object images," *International Journal of Computer Vision*, vol. 61, no. 1, pp. 103–112, 2005.

[24] A. Hauptmann, R. Baron, M. Chen, M. Christel, P. Duygulu, C. Huang, R. Jin, W. Lin, T. Ng, N. Moraveji *et al.*, "Informedia at TRECVID 2003: Analyzing and searching broadcast news video," in *Proc. of TRECVID*, 2003.

[25] C. Snoek, J. van Gemert, J. Geusebroek, B. Huurnink, D. Koelma, G. Nguyen, O. De Rooij, F. Seinstra, A. Smeulders, C. Veenman *et al.*, "The MediaMill TRECVID 2005 semantic video search engine," in *Proceedings of the 3rd TRECVID Workshop*, 2005.

[26] F. Seinstra, C. Snoek, D. Koelma, J. Geusebroek, and M. Worring, "User transparent parallel processing of the 2004 NIST TRECVID data set," in *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS 2005)*.

[27] D. Knuth, *The art of computer programming, volume 3: sorting and searching*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1998.

[28] online, "http://www.artima.com/underthehood/gc.html," 2007.

[29] M. Harchol-Balter and A. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Trans. Computer Systems*, vol. 15, no. 3, pp. 253–285, 1997.