Title

Efficient timing budget management for accuracy improvement in a collaborative object tracking system

Journal

Journal of Vlsi Signal Processing Systems for Signal Image and Video Technology, 42(1)

ISSN

0922-5773

Authors

Ghiasi, S
Bozorgzadeh, E
Nguyen, K
et al.

Publication Date

2006

Peer reviewed

Efficient Timing Budget Management for Accuracy Improvement in a Collaborative Object Tracking System

Soheil Ghiasi[1] *Karlene Nguyen* *Elaheh Bozorgzadeh* *Majid Sarrafzadeh*

Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90095
Tel: (310) 794-5616
Fax: (310) 794-5056
Email: soheil,karlene,elib,majid@cs.ucla.edu

Abstract

This paper presents the idea of managing the comprising computations of an application performed by an embedded networked system. An efficient algorithm for exploiting the timing slack of building blocks of the application is proposed. The slack of blocks can be utilized by replacing them with slower but cheaper, *i.e.* better, *modules and by assigning the computations to the proper resources. Thus, our approach manages the comprising computations and system resources and can indirectly assist the realtime scheduling of computations on system resources. This is performed without compromising the timing constraints of the application and can lead to significant improvements in power dissipation, computation accuracy or other metrics of the application domain. Our algorithm is well-suited for arbitrary tree computations. Moreover, it delivers solutions that are desirably close to the optimal solution. Experimental results for a number of object tracking applications implemented in an networked system with embedded computation resources, exhibit a significant amount of slack utilization.*

1 Introduction

Today's advances in technology has enabled the integration of processing resources, memory blocks and sophisticated I/O interfaces into data acquisition devices. Furthermore, these devices are often capable of communicating with other system devices through wired or wireless networks, and therefore form a network of embedded sensors and computation resources. Such *networked embedded systems* provide the opportunity of processing the perceived information locally at the sensor nodes as opposed to transferring the data to a remote processing station, having the station perform the computation and reading the result back.

[1]The contact author.

These two approaches to data processing, namely locally embedded at the sensor nodes and traditional centralized processor-based computing schemes, introduce many trade offs into the design space. Scalability, energy dissipation and performance of the system can be greatly impacted by the choice of computation scheme deployed in the system. Usually, embedded local computing improves system scalability and energy dissipation. It can also improve system performance for some applications. On the other hand, computation and resource management becomes a critical issue in such scenarios. This paper discusses this problem in a network of embedded devices tracking a moving object. Techniques to automate the computation and resource management in such environments have been proposed.

An instance of such a system consisting of a number of cameras and a controller is depicted in Figure 1. The figure demonstrates an intruder detection and object tracking system that has been built as part of this work [18, 14, 8, 9]. The system consists of multiple IQeye3 cameras [13]. The cameras communicate with the control unit in order to collaborate and distribute their information.

Figure 2 demonstrates the abstract model of the resources existing in the system. A general-purpose processor (IBM PowerPC) and a Xilinx VirtexE FPGA are embedded in each of the cameras. Any of these embedded resources can be used to realize the comprising basic blocks of an application. The control unit is also capable of performing various computations and has more powerful resources compared to the cameras. It is also responsible for realtime scheduling of the application processes on the system resources [16, 7].

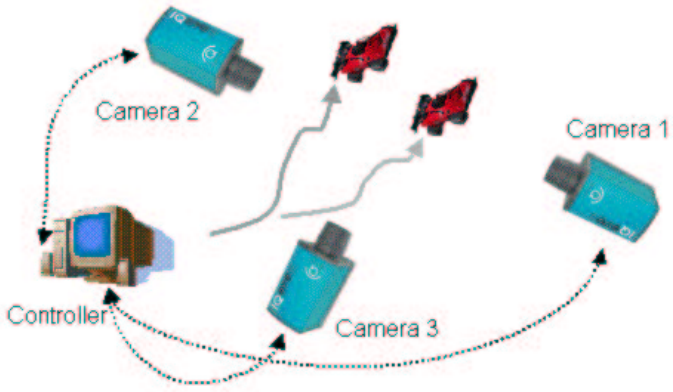

Figure 1: The implemented target tracking system.

An application, such as the aforementioned object tracking application, is composed of various basic blocks that perform different functionalities. To realize an application, each basic block has to be executed on one of system resources. Design libraries, voltage scaling, accuracy-timing trade offs and other design choices lead to different implementations for any

of the comprising blocks of the application. Each of the implementations has its own specifications in terms of timing characteristics, power dissipation, computation accuracy, etc. These specifications, provide a *discrete* range of choices for implementing a basic block.

As highlighted in Figure 2, any of the application processes can be executed locally on the resources embedded in the cameras or it can be shipped to the controller and performed in the traditional manner. In particular, the object tracking application is comprised of two basic computations, namely feature selection and feature tracking. Different versions of these two computations have been implemented for camera and controller processors. Each implementation offers a particular accuracy-latency tradeoff. This is used to study the involved trade offs of embedded vs. traditional processing.

The quality of an implemented application is usually evaluated using one of the standard design metrics such as power dissipation, accuracy or timing. Optimization techniques for improving one of these metrics will usually harm other characteristics of the design. In particular, intensive timing optimization of a design usually imposes additional costs on its implementation. Based on the application domain the system power dissipation, accuracy or response time can be interpreted as the cost. In the conventional implementation process, one of the performance metrics is usually fixed as a constraint and any implementation that does not meet the constraint has no value. Optimization procedures are then used to select the *cheapest* design among all the implementations that meet the constraint.

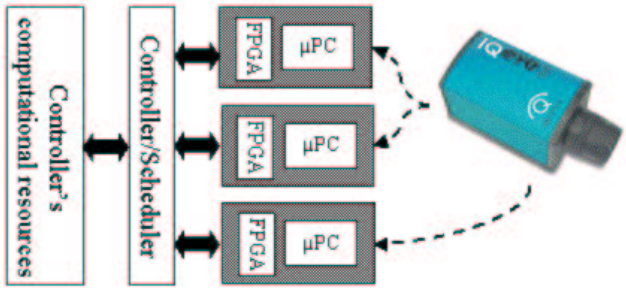

Figure 2: The abstract model of the system resources. The controller has to schedule the tasks either on the camera's embedded resources or its computational units. Each camera has an FPGA and a processor embedded in it.

For a design to meet the timing constraints, it is not always required that all of the comprising basic blocks (or computations) run with the lowest possible delay. In fact, some basic blocks can be slowed down which in turn might lead to significant savings in other design metrics while the design still meets the timing constraints. The process of determining the delay (latency) values for comprising basic blocks of a design is called *delay budgeting*. The objective of this

work is to propose a provably effective delay budgeting scheme for managing the computation latencies and resource utilization such that the design timing constraints are met while using the slower implementations of computations.

The proposed technique determines the delay values for basic blocks of a specific class of applications, namely the applications that can be modeled using a rooted tree. Each computation latency corresponds to a specific implementation of that basic block on a particular resource. Therefore determining the latency value for computations assigns the basic blocks to different system resources, hence manages the resources and computations at the same time. The problem, as formulated in this paper, is NP-hard. Therefore, an optimal polynomial time algorithm to solve it does not exist unless P=NP. Given sufficient amount of time, the proposed approximation algorithm can find solutions as close to the optimal solution as desired.

Section 2 summarizes the previous works on slack management and delay budgeting. Section 3 presents some basic definitions and notations that are used throughout the paper. The problem addressed in this paper is formally described in section 4. Section 5 describes the proposed approximation algorithm and discusses some interesting extensions. Section 6 explains a class of vision applications that can be modeled using a rooted tree. A set of experimental results supporting our algorithm and its solution quality are also reported and finally section 7 concludes the paper.

1.1 An Example

Figure 3.a shows an example of a data flow graph representing an application with four basic blocks. There are two different implementations with different latencies available for realizing each of the blocks. The possible delay values for each node are indicated in Figure 3.a. The figure also illustrates three different implementations of the same application. Assuming that the entire computation should not take more than 8 delay units, Figure 3.b shows an implementation that violates the timing constraint, because it takes 9 delay units to complete. Sections *c* and *d* of the figure depict two valid implementations of the same application with different total delay of the application blocks.

Since implementation of a block with relaxed timing constraints can lead to significant savings in other design metrics, one would like to maximize the total delay of the implemented blocks. Intuitively, the implementation in Figure 3.c is more likely to be *cheaper* than the implementation in Figure 3.d, because the latter has a total delay of 11 units while this quantity for the former design is 14 units.

This example shows that an effective delay budgeting scheme can improve the quality of the final design. This improvement is done through constraint relaxation as much as it does

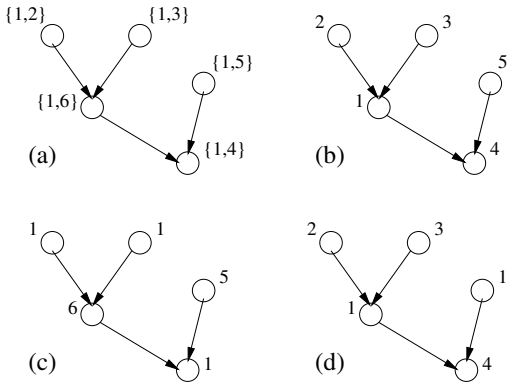

Figure 3: (a) A sample design with different delay choices for each block. (b) An invalid implementation violating the timing constraint of 8 units. (c, d) Valid designs with different timing budgets for basic blocks

not violate the timing constraints of the design. In this paper, we present an ϵ-approximation algorithm that can find solutions as close as desired to the optimal solution.

2 Related Work

The concept of slack in a more general context has been addressed in the synthesis community. The concept of mobility has been proposed which essentially is the difference between the ALAP and ASAP schedule steps for each operation. This concept has been used to prioritize the scheduling of operations [19, 20]. Mobility is also used in *force directed scheduling* [21, 3] aiming to balance the number of operations in each control step. In all these works, the concept of slack/mobility has been used to generate a valid schedule and the objective being minimizing the number of required control steps most of the time. Operation mobility has been a guide for these algorithms to obtain a homogeneous distribution of operations to control steps while pursuing this objective. They are not particularly concerned with the possible ways to exploit the final slack distribution within the schedule for improving other aspects of the design.

In domains other than high-level synthesis, techniques to exploit slack to achieve certain objectives have been proposed. Shin and Choi proposed a scheduling algorithm for hard real-time systems targeting a reduced power solution [22]. Their method yields power reduction by exploiting slack inherent in the system schedule and those arising from variations of execution times. Luo and Jha developed a scheduling technique for distributed real-time embedded systems [15]. Their scheduling technique performs variable-voltage scheduling via efficient slack time re-allocation to reduce average discharge power. Zhu et. al. [24] describe a scheduling technique to reclaim the time unused by a task to reduce the execution speed of future tasks.

Fields et al. propose a slack prediction scheme to schedule operation in different pipelines with different speeds for energy optimization [5] . Zhang et al. [23] propose compiler optimization techniques for exploiting slack in VLIW processor with decreasing the energy consumption as the objective. Chen et al. proposed a slack distribution algorithm for logic circuits [2]. Given a network of logic gates represented with a Directed Acyclic Graph (DAG), this algorithm utilizes the available slack in the network as delay budgets to logic gates. [1] studies the same problem, however they present optimal results for integral budgeting. The authors report improvements over the Zero Slack Algorithm [17], however they have the assumption of continuous possible latency values for different implementations of each gate. Hence, their approach cannot be directly applied to the cases where the possible latencies of each module are discrete values.

The current paper, combines the results of prior summarized versions of this work [11, 12, 10]. It present a technique for utilizing mobility of components in a discrete latency scenario. Our algorithm improves other metrics of the design (such as power dissipation) through slack management without affecting the timing constraints.

3 Preliminaries

An application can be viewed as a set of basic tasks that are to be performed on the input data in some specific order. Each basic task could be a complex operation that is invoked by the application in order to accomplish its functionality. The data dependency among the comprising tasks is usually modeled as a directed acyclic graph (DAG). However for some classes of applications, the corresponding DAG does not have any reconvergent fanouts. In this case, the data dependency among the basic tasks can be represented using a rooted tree. Some instances of such applications from the image processing domain are mentioned in section 6.

Figure 4 illustrates an example of such applications. Each node of the tree corresponds to a module performing one of the comprising basic tasks. Throughout this paper we deal with the applications that can be modeled using a rooted tree. For the same reason, we use the terms *comprising task* of an application and *node* of the tree interchangeably.

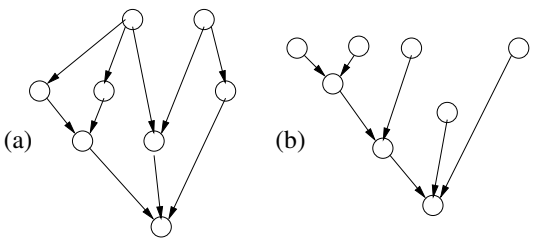

Figure 4: (a)An arbitrary application. (b) A sample application modeled using a rooted tree

A fanin of a node n is a node whose output is used as an input by n. Similarly a fanout of node n is a node that receives as input, the output of node n. We assume that each node has its own propagation delay, i.e. it needs a specific amount of time to generate its output after all its inputs become available. Given the *arrival times* at primary inputs (PI) and the *required times* at primary output (PO), the arrival and required times at the internal nodes of the tree can be computed as follows.

$$r(n) = \min_{x \in FO(n)} (r(x) - d_x) \tag{1}$$

$$a(n) = \max_{x \in FI(n)} (a(x) + d_n) \tag{2}$$

where
$r(n)$ = Required time at the output of node n,
$FO(n)$ = Fanout set of node n,
d_x, d_n = Delay in node x,n,
$a(n)$ = Arrival time at the output of node n,
$FI(n)$ = Fanin set of node n.

In order to compute the arrival times, the nodes are visited in topologically sorted order (from PI to PO). The arrival time for each node is computed using Equation 2. For required times, the nodes are still visited in topologically sorted order, but from PO to PI. Equation 1 is used to set the required time at a node. We also define the *slack* associated with a node n as the difference between the required and arrival time at the node. The arrival time at the PIs are assumed to be 0. The required time at the PO depends on the timing constraints of the application and has to be greater than or equal to the slowest chain of tasks. So all the slacks in the network are non-negative. A critical node is a node whose slack is the minimum slack among all nodes. A critical path is a path from PI to PO which is made of only critical nodes. Any delay budgeting strategy should be careful with existing and newly born critical paths as they are the slowest paths in the circuit and hence they determine the application latency.

4 Problem Formulation

The tree delay budgeting problem can be formulated as below:

- Given an application modeled as a rooted tree with n nodes, a maximum affordable delay D_{max} (also called delay budget) for the entire application and at most m possible latency values for different implementations of each node.

- The objective is to select a delay value for each node and maximize the total delay of the nodes, i.e. we would like to make

$$O.F. = \sum_{i=1}^{n} d_i$$

as large as possible, where d_i is the selected delay value for node i.

- Such that the propagation delay from each primary input to the primary output is not greater than D_{max}. In other words, the entire application can be performed within the delay budget. This constraint guarantees an upper bound for the application execution time.

The idea is that the extra delay on each node can lead to a cheaper implementation for that particular node. Therefore, maximizing the total delay of the nodes seems reasonable in order to decrease the implementation cost. The extra delay on nodes that are slowed down can be utilized to improve the total power dissipation, area or any other design metric that contributes to the *cost*. Intuitively, this problem tries to relax the timing constraints of the nodes that are not critical to the application runtime. The extra delay of such nodes can be exploited to implement them in a *cheaper* fashion. Note that regardless of the user cost function, a less timing-constrained module is always cheaper to implement.

5 An Approximation Algorithm for Delay Budgeting Problem

It is not hard to see that the subset problem [6, 4] can be reduced to a special case of the formulated problem, namely when the application tree is a path of basic blocks. Therefore the formulated delay budgeting problem is NP-hard and a polynomial time optimal algorithm for solving it does not exist unless P=NP [6]. In this section we present an algorithm that can find a solution that is arbitrarily close to the optimal solution of the problem.

Given an application tree T with n nodes and ϵ as the desired approximation accuracy, assume the tree nodes are scheduled using the ASAP algorithm. Therefore each node is assigned to a *level* (figure 5). Furthermore, assume that node i is implemented with its smallest possible delay value, i.e. d_{min_i}. Also, Let $B = D_{max} - D_{cp}$ be the *extra* delay budget, where D_{cp} is the critical path delay of the application.

For all nodes, let cp_i be the critical path delay of the subtree rooted at node i. We also associate a list L_i of delay-gain pairs to node i of the tree. Figure 5 demonstrates an instance of

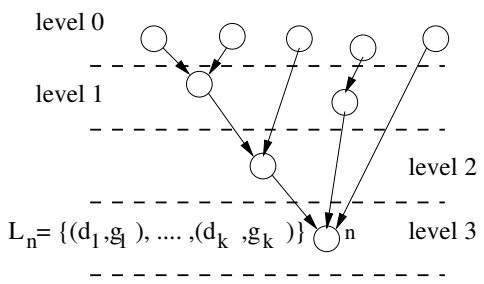

Figure 5: ASAP scheduling of the tree nodes to determine their levels.

this list for node n. Each pair has the form (D, G) and implies that G is the maximum achievable gain by assigning the extra delay D to the subtree rooted at that particular node. Given a particular node in the tree called i, L_i and the extra delay budget B for the subtree rooted at i, let $OPT(i, L_i, B)$ denote the maximum achievable gain by assigning B to that subtree. By definition, for two pairs (d_1, g_1) and (d_2, g_2) in L_i, if $d_1 < d_2$ then $g_1 < g_2$. Therefore, it is apparent that $OPT(i, L_i, B)$ can be easily determined by conducting a binary search on delay values of elements in L_i and comparing them with B. It follows that once this list is generated for the primary output, $OPT(PO, L_{PO}, B)$ will be the best possible solution for the problem.

We initialize L_i with all pairs of the form $(d_i - d_{min_i}, d_i - d_{min_i})$, where d_i's are possible delay values for node i. d_{min_i} is the minimum of all d_i's. Note that by definition L_i contains $(0, 0)$. The initialization process shows the achievable gain by assigning all the extra delay budget to the root node and hence forms a lower bound for the potential gain. Note that assigning the extra delay to the subtree root does not violate the timing constraints.

For updating the lists, we traverse the nodes in the order of their levels[2]. At each level the delay-gain list for the nodes in that level is updated and this process is continued until this list is updated for the primary output. Interestingly, as the subtrees rooted at level 0 nodes contain only those nodes, the initialization step forms the exact lists for them.

Now, we describe an algorithm for updating the lists at each level, assuming that delay-gain lists for nodes in the previous levels are already updated. We define two operations for accomplishing this task and prove their accuracy. First operation called $P(T_1, T_2 \ldots T_n)$ is applicable to cases where the root node in tree T has one possible delay choice equal to 0. Figure 6.a shows an example of this case. Clearly, The extra budget of T has to be applied to the fanins of node r since r has no other option for its implementation other than 0.

Since subtrees $T_1, T_2 \ldots T_n$ have different critical path delays, we might be able to slow down some of the subtrees even more than the extra delay budget. Clearly, the amount of the extra

[2]Note that ASAP is used because it is easy to implement. Any other topological ordering would be fine for our purpose.

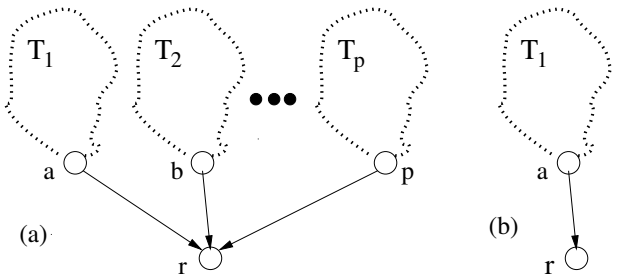

Figure 6: (a) Operation P applies the extra delay budget to all fanins. (b) Operation S divides the extra delay budget between r and the subtree T_1

delay for the subtree i is $cp_{max} - cp_i$, where cp_{max} is the maximum critical path delay among fanins of r. It follows that the delay-gain lists of the fanin nodes can be optimally merged to update this list at node r. This process is outlined by Algorithm 1.

Algorithm 1 Merging fanin delay-gain lists in operation P

Input: $L_a, L_b \ldots L_p$ and $cp_a, cp_b \ldots cp_p$,
Output: L_r

Let cp_{max} be the maximum of $cp_a, cp_b \ldots cp_p$;
Clear L_r;
for all fanin of r **do**
  for all pair $p(d, g) \in L_i$ **do**
    Let $p(d, g) = p(d - (cp_{max} - cp_i), g)$;
  end for
end for
for all extra delay budget $d \in L_i$'s **do**
  Add pair $(d, g_1 + g_2 + \ldots + g_n)$ to L_r where g_i is $OPT(i, L_i, d)$;
end for
Return L_r;

The second operation, $S(T_1, r)$ is applicable when the root node r has only one fanin, namely T_1 (figure 6.b). In this case, the extra budget has to be divided among r and T_1. We combine L_a and the initial value of L_r to update L_r. The combination process, shown in algorithm 2, enumerates all possible cases. Since the number of possible cases can potentially grow exponentially, a trimming procedure is applied to remove some of the pairs from L_r. Section 5.1 proves that the trimming procedure bounds the runtime of the algorithm by a polynomial in n and m while the solutions kept in the list are at most $(1-\epsilon)$ times smaller than the optimal solution.

In order to calculate the delay-gain list at the primary output of a tree, it is necessary to transform a general rooted tree to a tree that can be manipulated by two S and P operations. This can be achieved by replacing each node a of the tree by two nodes a_1 and a_2. We put an edge from a_1 to a_2. All fanins of a become the fanins of a_1 and all fanouts of a become the

Algorithm 2 Updating and trimming the gain-delay list in operation S

Input: L_a, L_r, ϵ, n,
Output: L_r

Let $L_{temp} = \emptyset$;
for all pair $p(d, g) \in L_a$ **do**
  Let $L_{temp} = L_{temp} \cup (L_r + (d, g))$;
  /*where $L_r + (d, g)$ is the list of all pairs in L_r with delay values increased by d and gains increased by g. */
  Trim L_{temp} by the factor ϵ/n;
  /*if there are two pairs with gains a and b in L_{temp} such that $1 - \epsilon/n \leq a/b \leq 1$ then remove the pair with gain b from the list. */
end for
Let $L_r = L_{temp}$;
Return L_r;

fanouts of a_2. Furthermore, we assume that a_1 has no delay and that is the only possible delay value for a_1. It follows that all possible delay values for a are possible for a_2. Figure 7 shows the transformation for a sample tree.

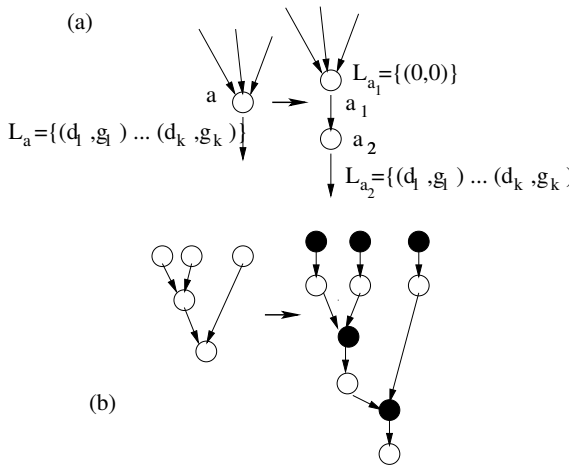

Figure 7: (a) Replacing a node with two to enable S and P operations. (b) Transformation on the entire tree. Black nodes are initialized to have zero delay as the only possible option.

As discussed earlier, a level by level traversing of nodes and applying S and P operations to nodes will update the delay-gain for all nodes. Once this list is updated for the primary output, $OPT(PO, L_{PO}, B)$ reflects the maximum achievable gain.

A minor modification of the described algorithm can monitor the intermediate pairs that contribute to the gain of each pair of L_{PO}. These pairs will provide the proper implementation choice for each node. The details of this implementation should be straightforward and are not explained in this paper.

5.1 Performance Guarantee

Since the trimming step introduces some error into the process, the calculated solution might not be equal to the optimal solution. In this section, we prove some upper bounds for this error and guarantee the solution quality within some limits. We also prove that the algorithm runs in polynomial time.

Theorem 1 *The solution given by the algorithm described in section 5, is at most* $1-\epsilon$ *times smaller than the optimal solution.*

Proof : Operation P does not introduce any error in the delay-gain list since it does not remove any pair from its result. Therefore S is the only operation that might introduce error in the result. There are n nodes in the tree, therefore the number of performed S operations cannot be more than n.

For each pair $(D, G) \in L_r$ that is removed after the operation S (figure 6.b), there is another pair $(D', G') \in L_r$ such that $1 - \epsilon/n \leq G'/G \leq 1$. This implies that $D' \leq D$ and hence G' can be utilized in case the deleted pair was to be chosen.

Now, suppose (D^*, G^*) denotes the optimal delay-gain pair at the primary output for a given problem instance. By induction on the number of consecutive S operations, it can be easily shown that there is a pair $(D', G') \in L_{PO}$ such that $(1-\epsilon/n)^n \leq G'/G^* \leq 1$ and $D' \leq D^*$. It follows that:

$$1 - \epsilon \leq (1 - \epsilon/n)^n \leq G'/G^* \leq OPT(PO, L_{PO}, B)/G^*$$

This completes the proof of the approximation bound ■

Note that the P operations that might happen between two S operations will always keep the pairs within the desired bound and hence, will not hurt the solution quality.

Theorem 2 *The aforementioned algorithm runs in polynomial time in terms of* n.

Proof : To prove that the algorithm runs in polynomial time, we bound the number of elements in L_{PO} (and hence other intermediate lists). Since the trimming step removes the close-enough gain elements from the lists, each two pairs (d_1, g_1) and (d_2, g_2) remaining in a list must satisfy the equation $g_2/g_1 > 1/(1-(\epsilon/n))$ assuming that $g_2 > g_1$. Therefore the number of distinct gain values is at most:

$$\log_{1/(1-\epsilon/n)} nD_{MAX} = \frac{\ln(nD_{MAX})}{-\ln(1-\epsilon/n)} \leq \frac{n\ln(nD_{MAX})}{\epsilon} = \frac{n(\ln n + \ln D_{MAX})}{\epsilon}$$

which is a polynomial in terms of the algorithm inputs and $1/\epsilon$. Thereby the proposed algorithm is a fully polynomial time approximation scheme (FPTAS) ■

5.2 Extensions to Other Objective Functions

The achieved gain by slowing down a node might not be linearly proportional to its delay. For instance a specific implementation of a module which runs two times faster than another implementation, might not be two times more *costly*. In such cases, maximizing the total delay of the nodes will not necessarily lead to the maximum gain.

However, in many cases the amount of potential gain can still be modeled using a function of delay. If gain can be modeled using a polynomial function of the module delay, the above approach and algorithm can be directly applied to approximate the solution to the desired degree. In other words, the above mentioned technique also works for the following objective function:

$$O.F. = \sum_{i=1}^{n} P(d_i)$$

where P is a polynomial function of delay values.

6 Experimental Results

In this section, we present the result of applying our time budgeting algorithm on several applications. The algorithm has been implemented in C and applied to a number of target tracking applications. First, we explain the framework of the performed experiments. Then we report the results of accuracy and delay profiling of different implementations of each application. Finally, we report the simulation results. All experiments support the fact that we approximate the total inserted latency into the applications correctly.

6.1 Experiment Framework

We have implemented a system consisting of multiple cameras with embedded computational resources to track the objects moving in a particular area (Figure 1). As discussed in section 1, the cameras have multiple resources to realize the basic blocks of the application. Furthermore, the controller resources can be used for executing the computations. The particular application of interest is object tracking which is composed of two basic vision algorithms, namely feature selection and feature tracking.

The feature selection algorithm is used to select the *proper* points in an image to track. Sharp corners with significant color or intensity variations are usually considered as good features. Figure 8

Figure 8: Selected features on a sample image. Features are shown by dark squares.

Given two consecutive images taken from a scene, the feature tracking algorithm tries to find the location of the first image features, in the second frame. This will provide information for following the moving object in the scene. This procedure is repeated for the next upcoming frames, i.e., the tracked features in the second image will be tracked in the third frame and so on. Hence, the normal procedure for tracking an object is to run feature selection on the first image and then switch to feature tracking for the next upcoming images. Figure 9 illustrates a sequence of frames of a walking girl. Tracked features are shown with dark squares in the figure.

The tracking algorithm might not be able to track all of the specified features in the second image and hence might *lose* some of the features for tracking in the upcoming images (Figure 9). For example, a sample run of the implemented tracking scheme on 10 consecutive images missed 44 out of the initial 100 features. Moreover, the motion of the objects in the scene can hide some of their features or even create new features visible by a particular camera. Therefore, the tracking system needs to re-execute the feature selection algorithm at some points of time during the application lifetime. This will refresh the feature points provided to tracking algorithm and improves the tracking accuracy.

The appropriate time to re select the features in a scene depends on many practical parameters such as the required accuracy, camera resolution and object's shape and motion. Therefore, any instance of the tracking application would switch from feature tracking to feature selection at a particular moment of time and hence has a different data flow graph (DFG). All of the DFGs representing such applications have the rooted tree form. Figure 10 depicts a few iterations of a sample application. The actual path that the application takes at runtime is not known in advance and depends on the events happening in the scene. However, it is known that the application will take "one" of the paths from the tree root to one of the leaves.

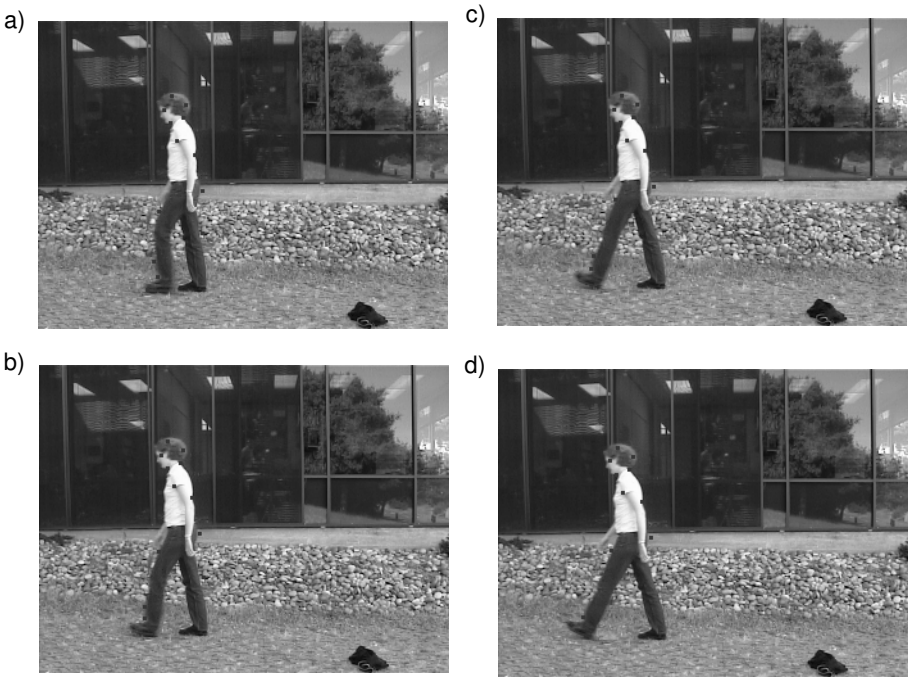

Figure 9: Selected features are tracked in a sequence of four images.

For many practical applications the system's worst case response time should be guaranteed. Therefore, the path delays from the tree root to any of the leaves cannot exceed the total delay budget. Our algorithm manages the computation delays such that the timing constraint is met and the sum of the computation delays are maximized. Each delay value for a computation, corresponds to a particular implementation of that computation on a particular resource. Therefore, the output manages system computations and resources and can assist the task scheduling.

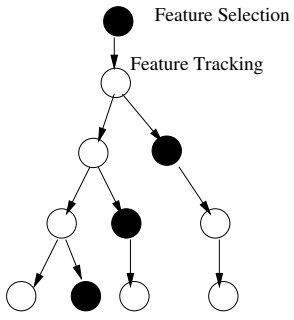

Figure 10: A sample tracking application modeled as a rooted tree. Black nodes represent feature selection algorithm and the white nodes stand for feature tracking.

6.2 Delay Budgeting Results

We have implemented several variations of the feature selection and tracking algorithms. These algorithms are executed either on the IQeye3 camera PowerPC processor or on the controller. These implementations perform the feature selection and tracking tasks with different accuracy levels, hence they exhibit different latencies.

Figure 11 demonstrates the accuracy and latency of various feature tracking implementations. The plot is created for five different implementations of feature tracking algorithm executed on the controller (a Pentium III PC running at 700 MHz). Different versions are named 'a' through 'e', where 'a' corresponds to the slowest (and the most accurate) and 'e' corresponds to the fastest (but the least accurate) implementation.

For each algorithm, 20 features are selected using feature selection algorithm. The selected features are passed to each of different implementations and the number of the tracked features is recorded. The number of tracked features has been shown for four consecutive iterations of each tracking ıch other, correspond to

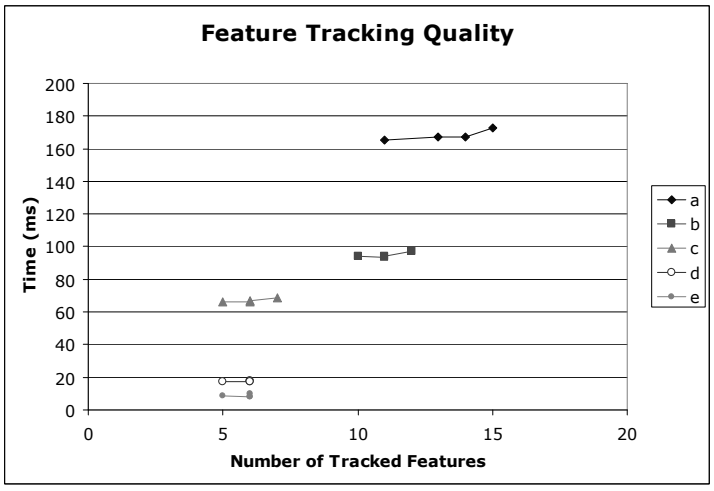

Figure 11: The trade off between accuracy and latency of different feature tracking implementations. Slower implementations track more accurately.

The number of tracked features is a good metric for quantifying the tracking accuracy. Low accuracy tracking algorithms easily lose the features during the tracking process. Therefore, tracking algorithms than can follow a larger number of features are considered more accurate. As it can be seen in Figure 11, faster implementations of the tracking algorithms are less accurate in terms of the tracking quality. We refer to this fact as *accuracy-latency trade off*. In the current set of experiments, we utilize the algorithm presented in section 5 to exploit the

timing slack of each operation and improve the tracking quality.

Table 1 summarizes the delay values for different implementations of features selection and tracking. The delays are measured for a single run of the algorithm on a sample image. All other affecting parameters are fixed, hence we expect that these numbers scale by porting the algorithm to other platforms.

All numbers are reported for executing the computations on camera's embedded processor, except for the first line of each section, which corresponds to running processes on the controller's processor (table 1). While controller's processor is more powerful than processor embedded in the camera, the communication delay for sending the data to the controller and reading the result back slows down this scheme. The significant difference in computation latencies run on camera and the controller highlights this fact.

Algorithm	*Execution Delay* $[ms]$
Feature Selection	
Executed on controller's cpu	3300
Basic Implementation	340
Modified gradient calculation	310
Removed multi-resolution pyramid	295
Feature Tracking	
Executed on controller's cpu	3345
Basic Implementation	360
Removed multi-resolution pyramid	230
Removed smoothing code	180
Modified gradient calculation	110
On demand gradient calculation	95
Tracking 60% of features	80
Tracking 30% of features	60
Simplified tracking for 30% of features	32

Table 1: The implemented algorithms and their corresponding delay values in milliseconds. Each latency is calculated for one iteration of the algorithm under similar conditions. Faster implementations have lower accuracy levels.

Also, a number of tracking application trees have been created. These trees correspond to object tracking applications with different accuracy requirements and they differ in the frequency of feature selection execution. Each tree has about 30 to 40 nodes. Each node is to be implemented with the tasks listed in table 1. Therefore, manual enumeration of all of the cases is not possible and practical. Figure 10 demonstrates one small sample tracking tree.

Each application has a particular worst case delay, i.e. it has to finish within a guaranteed amount of time. We assume that each application has at most a 10% extra delay budget. In

Application	*Minimum possible latency*$[ms]$	*Additional Inserted latency*$[ms]$ $\epsilon = 0.7$	*Additional Inserted latency*$[ms]$ $\epsilon = 0.5$	*Additional Inserted latency*$[ms]$ $\epsilon = 0.3$	*Additional Inserted latency*$[ms]$ $\epsilon = 0.1$	*Additional Inserted latency*$[ms]$ *Optimal solution*
*app*1	949	1227	1227	1257	1283	1283
*app*2	1045	1704	1732	1732	1732	1737
*app*3	1013	1555	1555	1611	1611	1611
*app*4	949	1272	1272	1302	1328	1328
*app*5	1045	1451	1471	1471	1471	1471
*app*6	981	1426	1524	1519	1524	1524
*app*7	1340	3721	3692	3714	3721	3721
*app*8	1045	1800	1830	1810	1830	1830
*app*9	981	1708	1753	1796	1822	1822
*app*10	1077	1821	1825	1825	1825	1825

Table 2: The result of applying the proposed algorithm on 10 different tracking applications. The extra computation delay injected into the application (which can be utilized to improve the tracking accuracy) is mentioned. Last column shows the maximum achievable gain for each application. All simulations were performed assuming that the application can be slowed down at most 10% of its maximum latency.

other words, it has to finish its computation within 10% delay of its fastest possible implementation. The choice of 10% extra delay budget has been made for experimental purposes only. Our approach would be effective for any other choice of delay budget.

The proposed algorithm has been implemented in C and applied to the aforementioned applications. The result of the simulation is reported in Table 2, which summarizes the result of applying our algorithm on 10 different object tracking applications. For each application, the delay of its fastest possible implementation is mentioned. The delay of the implemented applications are at most 10% more than the minimum possible delay. Several simulations have been performed with different approximation factors(ϵ values) and all of them support the fact that the additional inserted computation delay into the application is within the required bound from the optimal solution.

7 Conclusion

We presented the idea of delay budgeting at the application level by using a fully polynomial ϵ-approximation algorithm. The proposed algorithm works for a particular class of applications that can be modeled using a rooted tree. It is assumed that each block can be implemented in a number of ways and each implementation has a different latency.

The proposed algorithm attempts to exploit the timing slack of each basic block and replaces the basic blocks with slower implementations without violating the timing constraints of the design. The slower implementations of each block are usually less costly in terms of conventional design metrics such as power dissipation, area and accuracy. Thus, the entire procedure leads to savings in the application implementation cost or improvements in its quality.

Experimental results to advocate our algorithm and its solution quality have been reported. The experiments are performed on some object tracking applications. The result for multiple approximation factors and delay budgets support our algorithm performance and solution quality.

Future works include extension of the experiments to verify the simulation results in action. Moreover, the class of considered applications should be generalized to less restricted applications. The present algorithm can handle the applications that can be modeled using a rooted tree. An extension of the target application class to directed acyclic graphs is also among the future directions.

References

[1] E. Bozorgzadeh, S. Ghiasi, A. Takahashi, and M. Sarrafzadeh. "Optimal Integer Delay Budgeting on Directed Acyclic Graphs". In *Design Automation Conference*, June 2003.

[2] C. Chen, X. Yang, and M. Sarrafzadeh. "An Effective Algorithm for Gate-Level Power-Delay Tradeoff Using Two Voltages". In *International Conference on Computer-Aided Design*, November 2000.

[3] R. Cloutier and D. Thomas. "The Combination of Scheduling, Allocation and Mapping in a Single Algorithm". In *Design Automation Conference*, 1990.

[4] T. Cormen, C. Leiserson, and R. Rivest. *"An introduction to algorithms"*. MIT Press, 1990.

[5] B. Fields, R. Bodik, and M.D. Hill. "Slack: Maximizing Performance Under Technological Constraints". In *International Symposium on Computer Architecture*, 2002.

[6] M. Garey and D. Johnson. *"Computers and Intractability, A guide to the theory of NP-Completeness"*. W.H. Freeman and Company, New York, 1979.

[7] S. Ghiasi, H.J. Moon, and M. Sarrafzadeh. "A Networked Reconfigurable System for Collaborative Unsupervised Detection of Events". In *Tehnical Report, Computer Science Dept, UCLA*, July 2003.

[8] S. Ghiasi, H.J. Moon, and M. Sarrafzadeh. "Collaborative and Reconfigurable Object Tracking". In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2003.

[9] S. Ghiasi, H.J. Moon, and M. Sarrafzadeh. "Improving Performance and Quality thru Hardware Reconfiguration: Potentials and Adaptive Object Tracking Case Study". In *Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, October 2003.

[10] S. Ghiasi, K. Nguyen, E. Bozorgzadeh, and M. Sarrafzadeh. "On Computation and Resource Management in an FPGA-based Computing Environment". In *International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, February 2003.

[11] S. Ghiasi, K. Nguyen, E. Bozorgzadeh, and M. Sarrafzadeh. "On Computation and Resource Management in Networked Embedded Systems". In *International Conference on Parallel and Distributed Computing and Systems*, November 2003.

[12] S. Ghiasi, K. Nguyen, and M. Sarrafzadeh. "Profiling Accuracy-Latency Characteristics of Collaborative Object Tracking Applications". In *International Conference on Parallel and Distributed Computing and Systems*, November 2003.

[13] IQinVision. "Product manuals and online documentation". *http://www.iqinvision.com.*

[14] R. Kumar, S. Ghiasi, and M. Srivastava. "Dynamic Adaptation of Networked Reconfigurable Systems". In *Workshop on Software Support for Reconfigurable Systems*, 2003.

[15] J. Luo and N. Jha. "Battery-Aware Static Scheduling for Distributed Real-Time Embedded Systems". In *Design Automation Conference*, June 2001.

[16] A. Nahapetian, S. Ghiasi, and M. Sarrafzadeh. "Scheduling on Heterogeneous Resources with Heterogeneous Reconfiguration Costs". In *International Conference on Parallel and Distributed Computing and Systems*, November 2003.

[17] R. Nair, C. Berman, P. Hauge, and E. Yoffa. "Generation of Performance Constraints for Layout". *IEEE Transactions on Computer Aided Design*, 8:860–874, 1989.

[18] K. Nguyen, G. Yueng, S. Ghiasi, and M. Sarrafzadeh. "A General Framework for Tracking Objects in a Multi-Camera Environment". In *International Workshop on Digital and Computational Video*, November 2002.

[19] B. Pangrle and D. Gajski. "Design Tools for Intelligent Silicon Compilation". *IEEE Transactions on Computer Aided Design*, 6:1098–1112, November 1987.

[20] A. Parker, J. Pizarro, and M. Mlinar. "MAHA: A Program for Datapath Synthesis". In *Design Automation Conference*, 1986.

[21] P. Paulin and J. Knight. "Force Directed Scheduling for Behavioral Synthesis of ASICs". *IEEE Transactions on Computer Aided Design*, 8:661–679, 1989.

[22] Y. Shin and K. Choi. "Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems". In *Design Automation Conference*, June 1999.

[23] W. Zhang, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, D. Duarte, and Y.Tsai. "Exploiting VLIW Schedule Slacks for Dynamic and Leakage Energy Reduction". In *International Symposium on Microarchitecture*, 2001.

[24] D. Zhu, R. Melhem, and B. Childers. "Scheduling with Dynamic Voltage/Speed Adjustment using Slack Reclamation in Multi-Processor Real-Time Systems". In *IEEE Real Time Systems Symposium*, 2001.