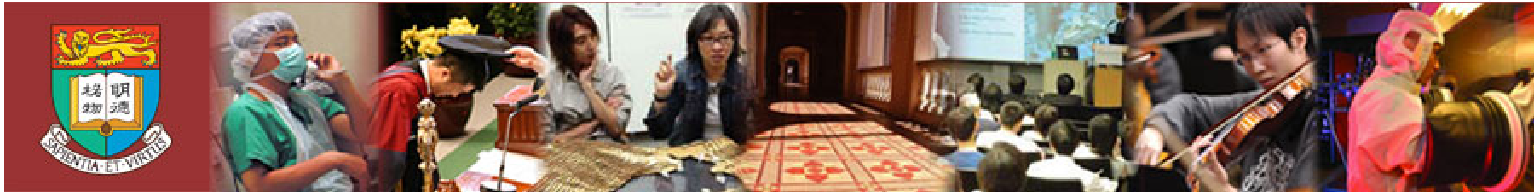
View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by HKU Scholars Hub



Title	Parallelization methodology for video coding - an implementation on the TMS320C80
Author(s)	Leung, KK; Yung, NHC; Cheung, PYS
Citation	Ieee Transactions On Circuits And Systems For Video Technology, 2000, v. 10 n. 8, p. 1413-1425
Issued Date	2000
URL	http://hdl.handle.net/10722/42876
Rights	Creative Commons: Attribution 3.0 Hong Kong License

Parallelization Methodology for Video Coding—An Implementation on the TMS320C80

Kwong-Keung Leung, *Student Member, IEEE*, Nelson H. C. Yung, *Senior Member, IEEE*, and Paul Y. S. Cheung, *Senior Member, IEEE*

***Abstract*—This paper presents a parallelization methodology for video coding based on the philosophy of hiding as much communications by computation as possible. It models the task/data size, processor cache capacity, and communication contention, through a systematic decomposition and scheduling approach. With the aid of Petri-nets and task graphs for representation and analysis, it employs a triple buffering scheme to enable the functions of frame capture, management, and coding to be performed in parallel. The theoretical speedup analysis indicates that this method offers excellent communication hiding, resulting in system efficiency well above 90%. To prove its practicality, a H.261 video encoder has been implemented on a TMS320C80 system using the method. Its performance was measured, from which the speedup and efficiency figures were calculated. The only difference detected between the theoretical and measured data is the program control overhead that has not been accounted for in the theoretical model. Even with this, the measured speedup of the H.261 is 3.67 and 3.76 on four parallel processors (PPs) for QCIF and 352×240 video, respectively, which correspond to frame rate of 30.7 and 9.25 frames per second, and system efficiency of 91.8% and 94%, respectively. This method is particularly efficient for platforms with small number of parallel processors.**

***Index Terms*—Efficiency, H.261, H.263, parallel coding, petri-net, speedup.**

I. Introduction

IN THE PAST decade, the proliferation of video and audio applications has been substantial and widespread, to say the least. Technologies such as DVD, VCD, VoD, digital TV, and video phone, among others, are gradually emerging as consumer products or services, offering multimedia communications, information access and entertainment. A vital link in the success of these applications lies in how the video information is being communicated. To ensure such success, the International Telecommunication Union (ITU) introduced the H.261 recommendation in 1990, which is designed to standardize the video codec for audiovisual services at $p \times 64$ kbits [1]. Three years later, the MPEG-1 coding standard for moving pictures to be stored on digital storage media was announced by the International Organization for Standardization (ISO) [2]. In 1995, built upon the earlier H.261, the H.263 standard was recommended by ITU to deal with video coding for low bitrate communication [3], and the MPEG-2 standard for generic coding of moving pictures and audio was also introduced by ISO in the same year [4]. In 1997, the ISO announced the MPEG-4 for integrating the production, distribution and content access paradigms of digital TV, interactive graphic applications, and World Wide Web [5], and the MPEG-7 for standardizing descriptions of various types of multimedia information to allow fast and efficient searching of such information [6].

Apart from these standards, there are also other video-coding methods [7]–[9] that offer a high compression rate at acceptable visual quality and performance. Whichever method one chooses, the tremendous computational complexity of video coding has pushed single processor technology to its limit. It has been widely accepted that the high complexity of video coding demands multiple high-speed processors, fast cache, and dedicated bus or network to work in parallel. In reality, supercomputers have been frequently used for experimentation or verification of methodologies, while special hardware chips, boards or systems have been built for real-time applications. Both these technologies are either too expensive or dedicated. With the advances in desktop multiprocessor computers and parallel digital signal processor (DSP) technology, there is a real opportunity for practical implementation of programmable real-time video encoder at an affordable price. However, this demands a parallelization strategy that can exploit the potential parallelism and utilize the computing resources efficiently. In particular, this strategy should perform well with small processor number if it is to find applications in desktop systems.

On this issue, there have been a number of implementation examples of the H.261, H.263, MPEG-1, and MPEG-2 on various parallel systems. These approaches may be broadly classified into three categories according to their implementation platforms: supercomputers [10]–[14], network of workstations (NOW) [15]–[17], and dedicated DSP [18]–[21]. In terms of parallelization techniques, spatial [11],[21], temporal [12], [17], or both [10], [13] have been commonly considered. Only a few employed function decomposition on dedicated hardware [20]. From these examples, it can be observed that for the implementations on supercomputers, real-time performance is often achievable on large number of nodes with system efficiency (speedup/nodes) ranging from 32% [10] to 40% [11]. On the other hand, implementations on NOW achieve better efficiency (62%) on small number of nodes (12–16), but the actual frame rate is usually low [3–4.5 frames per second (fps)] [17]. On dedicated DSP, ignoring those simulated cases, the best implementation reported so far was an H.263 on a TMS320C80 system,

Manuscript received October 6, 1998; revised March 16, 2000. This paper was supported by the Texas Instrument Tsukuba Research and Development Center, Japan, by the University Grants Committee, Area of Excellence in Information Technology, Hong Kong, under Grant AOE98/99.EG01, and by the Centre of Urban Planning and Environmental Management, the University of Hong Kong. This paper was recommended by Associate Editor N. Ranganathan.

The authors are with the Department of Electrical and Electronic Engineering, the University of Hong Kong, Pokfulam Road, Hong Kong SAR, China (e-mail: kkleung@eee.hku.hk; nyung@eee.hku.hk; cheung@eee.hku.hk).

Publisher Item Identifier S 1051-8215(00)10627-5.

1051–8215/00$10.00 © 2000 IEEE

achieving 4.26 fps and an efficiency of 81% (MP not consider as a processor) [18]. As not many can have exclusive access to supercomputers for coding purpose, it becomes obvious that the focus should be on a viable parallelization method that works well on NOW or parallel DSP. Besides, many supercomputer implementations have low system efficiency, meaning that a high percentage of the system's time is not doing useful tasks. The goal of this parallelization method must be to bring the efficiency up to 100%. Furthermore, one-off or dedicated approaches limit their expandability and flexibility. As the four coding standards share a basic framework, it would be attractive if the new method is sufficiently general in describing this framework and allows performance analysis to be carried out before any practical implementation.

In this paper, we present a new parallelization method for video coding. It stems from the concept of performing computation and communication in parallel such that communications appear to be hidden by computation. The expected effect of this approach is that computations occupy most of the processor cycles, giving extremely high system efficiency. In essence, this method models the task size, processor cache capacity and communication contention, through a systematic decomposition and scheduling approach, with the aid of Petri-nets and task graphs for representation and analysis. With the task and data size known, typical problems such as cache miss may be avoided by imposing restrictions on the task and data size during decomposition. By considering communication contention on the network, task scheduling, and execution may be modeled more accurately to reflect actual events. The use of Petri-nets and task graphs help to visualize and analyze the model and enable theoretical study and practical refinement. The theoretical speedup analysis of this method indicates that it offers excellent communication hiding, resulting in system efficiency well above 90%. To prove its practicality, a H.261 video encoder has been implemented on a TMS320C80 system according to the model. Its performance was measured, from which the speedup, frame rate, and efficiency were calculated. The only difference detected between the theoretical and measured data is the program control overhead that has not been accounted for in the theoretical model. Even with this, the measured speedup of the H.261 is 3.67 and 3.76 on four parallel processors (PPs) for QCIF and 352 $\times$ 240, respectively, which correspond to frame rate of 30.7 and 9.25 fps, and system efficiency of 91.8% and 94%, respectively [22].

This paper is organized as follows. Section II describes the parallelization method and the estimation of computation and communication delays. Section III demonstrates how the method is applied to implementing the H.261 encoder on the TMS320C80. Section IV presents the measurement conditions as well as the detailed results and discussions. This paper is concluded in Section V.

II. Parallelization Methodology

A. Preliminary Considerations

Ideally, parallel processing with N_P number of processors should have a speedup of N_P times in performance. However, this is possible only if the problem can be decomposed into exactly N_P equal workload tasks that can be executed in parallel without any other overhead, and all the processors start and complete their work simultaneously without any idling. If we define system efficiency as the ratio between speedup and N_P, such parallelization is said to have linear speedup and 100% system efficiency. In reality, algorithms usually contain a sequential component and a parallel component, in which only the parallel component can be decomposed into parallel tasks. Moreover, these parallel tasks may have certain dependency that requires communication between them during execution. This communication overhead cannot be completely ignored even if fast network is used. Furthermore, task decomposition and scheduling create constant overhead that can be substantial too. Adding these up, the true performance of a parallel implementation would be determined by how well the sequential component and various overheads can be minimized or hidden.

Let α be the probability that the system is used in a pure sequential mode on one processor. The probability of using all N_P processors in a fully parallel mode is thus $1-\alpha$ [23]. Amdahl's Law states "If the sequential component of an algorithm accounts for α of the program's execution time, then the maximum possible speedup that can be achieved on a parallel system is $1/\alpha$." This can be interpreted as when an algorithm is parallelized by N_P processors, the sequential component execution time remains unchanged, while the execution time of the other components are reduced by N_P time. If T_1 and T_n represent the execution times on 1 and N_P processors, respectively, we have

$$T_n = (1-\alpha)\frac{T_1}{N_P} + \alpha T_1. \tag{1}$$

If $N_P \rightarrow \infty$, then the speedup $T_1/T_n \rightarrow 1/\alpha$. Here, the speedup is upper bounded by $1/\alpha$ no matter how large N_P is. If $\alpha \rightarrow 0$, then $T_1/T_n \rightarrow N_P$, which is the ideal case. In general, α is also called the sequential bottleneck in a program.

For communication, if N_P is large, the delay through the interconnection network grows proportionally. To reduce this effect, one can either reduce the communications between processors, or use high-speed network to shorten the delay. The problem is that the communications may be difficult to reduce, and if this is the case, no matter how fast the network is, network delay could still be substantial. Another possible and yet more attractive approach is to incorporate a dedicated communication unit working in parallel with a computation unit, in each processor. The whole idea is that when computation is in progress, there can be communication in the background. Rather than trying to reduce the absolute communication delay, this approach attempts to hide all or part of the communication.

Furthermore, decomposition and scheduling are crucial to the whole parallelization approach. How the problem is decomposed into tasks determines the granularity of the parallelization. To arrive at an appropriate granularity, task and data dependencies must be known *a priori*. The general rule is that the tasks of a sequential component have strong dependency, whilst the tasks of a parallel component have weak dependency. Similarly, data could be exclusive to a task or shared by others. Once it is allocated, the shared data between processors creates the basic communication needs. When performing scheduling, granularity plays a major role. Scheduling is simpler for

coarse granularity because of the smaller number of tasks and less communication between them. However, task workloads are more difficult to be balanced, and efficiency is expected to be poorer. In addition, if the tasks/data allocated to a processor is larger than its cache size, then the cache misses would cause unexpectedly long task delays. Conversely, fine granularity results in numerous smaller tasks and probably more communications between them. It would usually be easier to balance the workload in each processor, giving better efficiency and smaller cache miss problem. However, more communication means that a match in the cache size and task size is not an issue to be overlooked if one strives for true performance.

B. Computation Characteristics of Video-Coding Algorithms

Existing video-compression standards rely on the reduction of temporal and spatial data redundancy existing among digital video data. Temporal data redundancy corresponds to data correlation between pixels across different frames and it is reduced by motion compensation between frames. In a macroblock (MB), pixels are coded as the motion-compensated residues after subtracting by the pixels in the reference frames. Since the residues are usually smaller in magnitude than the pixels themselves, compression is achieved by coding the residue data and the motion vectors. Usually, there is one motion vector per MB, although some standards support more than one motion vector per MB by distinguishing motion vectors between forward and backward directions and between odd and even picture field references.

To determine the motion field, motion estimation (ME) is usually performed. Assume that the current and reference frame data are supplied from a global data server (DS), which can be a video capturing processor or a communication processor, ME is a process of searching for the closest MB from the reference frames within a search area. The search area is a bounded and enlarged area in a spatial position offset from the position of the currently coded MB. Different standards have different sizes. Theoretically, there is no specific ordering of the MBs in a picture regarding ME, it can be parallelized over all the MBs if so desired.

After motion compensation, the frame of residues is divided into a number of blocks of 8×8 pixels for transform coding, quantization, zigzag traversal, and variable-length coding. Similar to ME, there is no specific ordering of the blocks in a picture regarding these coding steps. Therefore, in theory, coding of a frame can be parallelized spatially with granularity of MBs as long as the required input data and reference frame data are available.

The final step in coding a frame is the generation of the output bitstream. This step includes the construction of the bitstream structure by inserting headers of various levels that often use differential coding to reference the previous or neighboring headers. For this reason, the bitstream construction is inherently sequential. In the following discussion, we consider the bitstream construction for a frame to be a single nondivisible task called *header-VLC*.

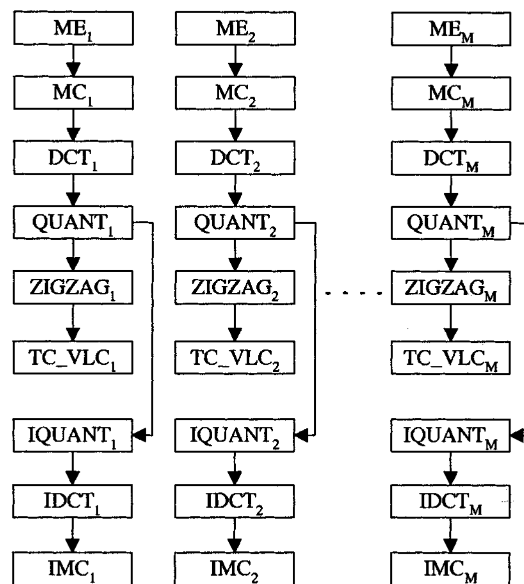


Fig. 1. Task graph for video coding.

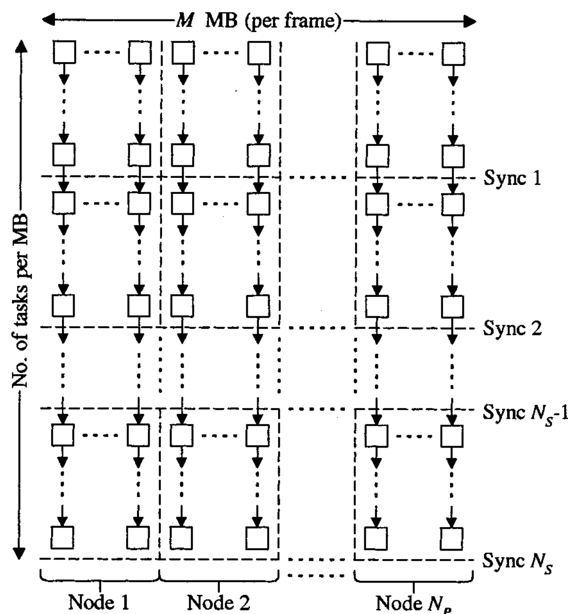


Fig. 2. A task-allocation scheme.

C. Task Decomposition

In this paper, a task is defined as a data unit together with a piece of function or code that operates on the data. The data includes the variable space for the input, output and any side effect involved. For example, the ME of an MB is a task with the input MB, all referenced pixels in the search area, and the resulting motion vector as data. Its function is to find the motion vector giving the smallest sum-absolute-error. The side effect is the error statistic, which can be used for future MB type determination.

In determining the size of a task, several criteria should be considered. Firstly, data size of a task is restricted to be smaller than or equal to the processor cache. Second, the execution time

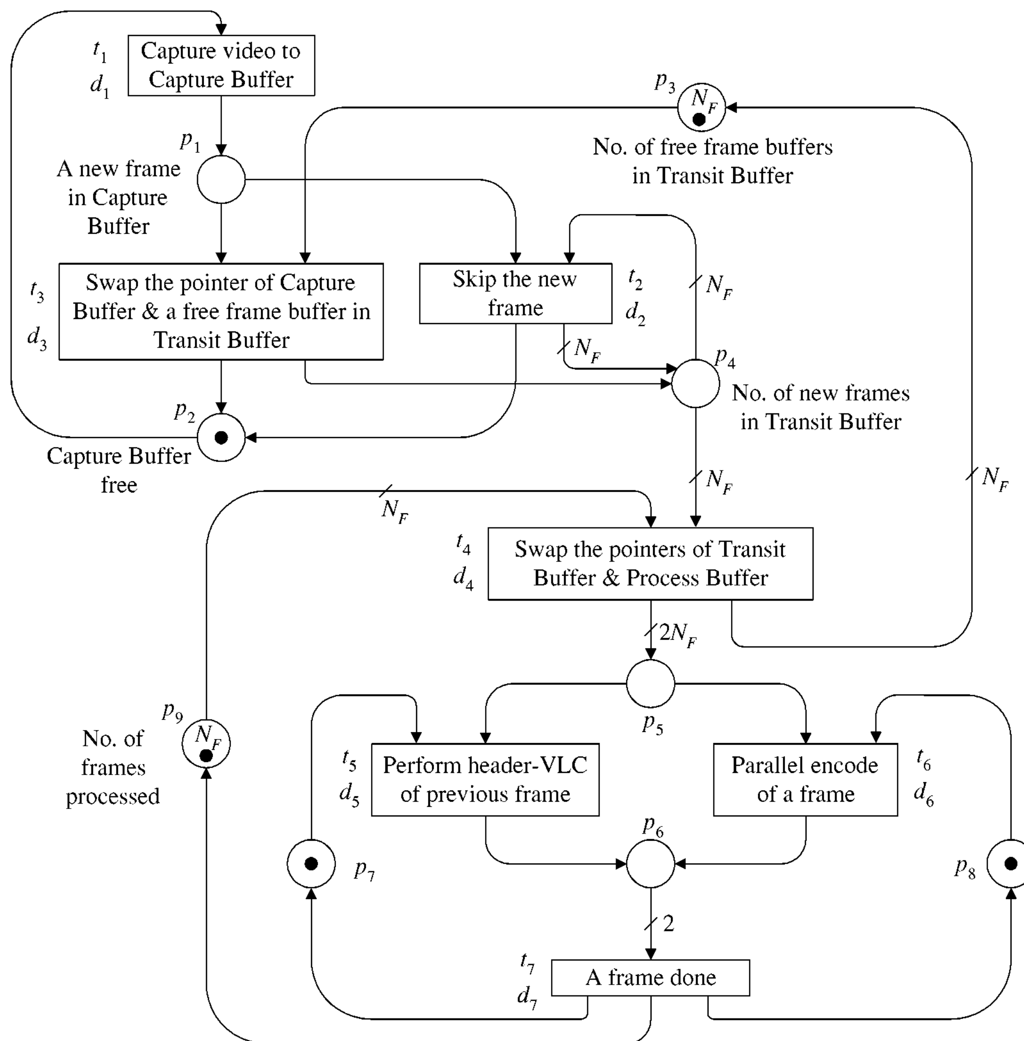


Fig. 3. Petri-net representation of the hiding scheme

of a task at a remote processor should be greater than the data communication time for loading the task and saving the results from the processor. Otherwise, it is better to execute the task locally. Third, a task should possess certain generalized functional meaning, as they have higher possibility of being reused in a class of applications. In general, if not all three criteria can be satisfied in each case, the first two should take precedence of the third.

Usually, it is desirable to have large number of parallel tasks, which can be grouped according to data access locality and temporal locality. To satisfy this, we argue data decomposition is better than functional decomposition because within a function, data accessed usually has high temporal locality. If a function is split into two tasks, the data shared between tasks may cause more communication overhead. Conversely, two tasks with shared data may be merged to form a bigger task to reduce such overhead.

The above considerations were applied to video coding and Fig. 1 depicts a general task graph. It consists of nine tasks per MB in each column. The *arc* between two tasks represents the precedence relation between them. If a frame consists of M MB, there are $9M$ tasks per frame, where there is no precedence constraint between tasks of different MBs.

D. Task Memory Access and Allocation

The mapping of tasks to the processors has two constraints. The first is that the precedence relation established in the task graph must be followed. If two tasks with precedence relation are mapped to different processors, there must be at least a synchronization point in time between the two processors such that the precedence relationship is satisfied. It should be noted that the synchronization point introduces processor idling time, as the processors that complete their execution faster than the others would have to wait for those that are still executing. To avoid unnecessary synchronization, tasks should be allocated such that there is a minimum number of synchronization points. The second is the matching of task data to processor memory. This constraint is not a necessary condition, but it allows the use of triple buffering to hide communication overhead. To do that, the basic requirement for triple buffering is that the local memory of a processor can at least hold two tasks simultaneously. When a task is being executed, the result of the last task is saved to the global buffer while the next task is being loaded. If the saving and loading of data complete earlier than the computation of the current task, the processor can proceed to the next task without idling. Strictly, the first constraint is for cor-

rectness and therefore must be satisfied. The second constraint is for achieving better performance, which can be sacrificed if necessary.

Mathematically, a sequence of tasks allocated to a processor should be constructed on the condition that every pair of successive tasks have the union of their data access smaller than or equal to the processor memory size. Let $S = \{T_i | i = 1, 2, \cdots, N_T\}$ be a sequence of tasks allocated to a processor, N_T be the number of tasks in S, $M(T_i)$ be the data accessed by T_i, and C be the size of processor memory. If the tasks in S are executed in the order of increasing i, the condition is that

$$|M(T_i) \cup M(T_{i+1})| \leq C, \qquad i = 1, 2, \cdots, N_T - 1. \quad (2)$$

To illustrate these points, Fig. 2 depicts one possible task allocation scheme, in which the M MBs are decomposed into N_P subsets of M/N_P MBs each when allocated to each processor. Under this scheme, each processor can perform a sequence of tasks before a synchronization point and start another sequence of tasks after a synchronization point.

E. Communication Hiding

Let N_F be the number of frames to be coded as a unit. For MPEG-1/2, N_F is the number of B-frames between successive I- or P-frames, plus the trailing I- or P-frame. For H.261 and H.263, N_F is 1 and 2, respectively, which H.263 allows two frames to be coded as a unit called a PB-frame. The Petri-net representation in Fig. 3 depicts the buffering scheme. In the net, a circle represents a *place* labeled as p_i [24]. The rectangular boxes represent *transitions* labeled as t_i. If a transition is associated with a time delay, it is labeled as d_i, else it is represented by a *bar*. The *arcs* between places and transitions carry a weight of unity unless specified. A transition is enabled and fired if all of its inlet places have the number of tokens specified in the corresponding arc. Once fired, the tokens enabling the transition are consumed while new tokens are generated in the outlet places. The solid black circle and number inside a place represent the number of tokens in that place. The tokens currently shown in Fig. 3 represent the initial marking, where only t_1 is enabled for capturing a frame.

In this scheme, there are three sets of logical buffers. The first logical buffer is called the *capture buffer* (CB) which is used to hold the frame currently being captured. Its size is equivalent to a single frame buffer. The second logical buffer is called the *transit buffer* (TB), which has N_F frame buffers. Its purpose is to keep a set of N_F recently captured frames. The third logical buffer is called the *process buffer* (PB), which is the same as the TB and it is used to hold the N_F frames that are being coded currently. The utilization of these buffers is such that the physical frame memory corresponding to a logical frame buffer is not fixed. For instance, when a new frame is captured into the CB, its buffer pointer is swapped with the pointer of a free frame buffer in the TB. This allows the capturing hardware to access the free buffer and load the next frame. This continues until N_F frames are stored into the TB. Then the N_F pointers of the PB are swapped with those of the TB such that the physical memory associated with the TB is now associated with the PB instead,

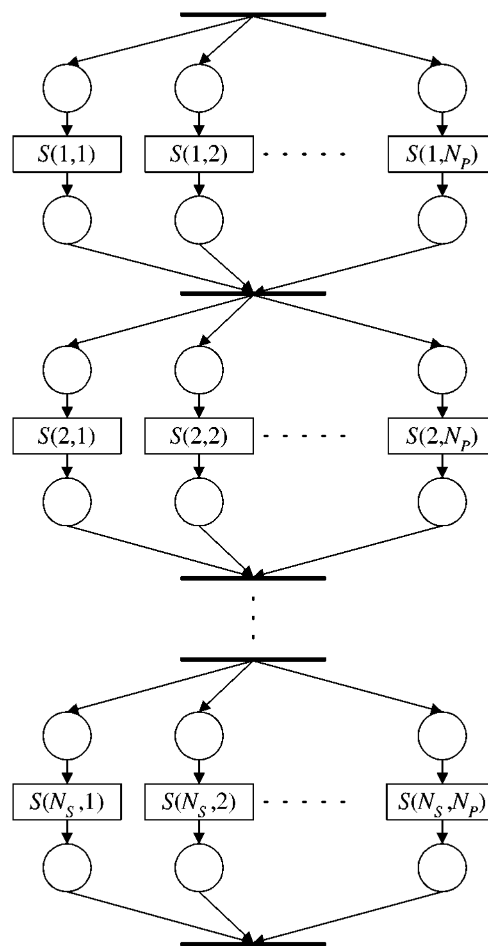


Fig. 4. Petri-net for parallel coding of a frame.

and vice versa. The delays for swapping buffer pointers (d_2, d_3, d_4) are considered insignificant compared with the coding time. Note that in this scheme, the coding of a frame is divided into two parallel transitions, t_5 and t_6, with delay d_5 and d_6, respectively. It is because the Header-VLC task is inherently sequential across the MBs while all the other functions from ME to IMC can be decomposed into parallel tasks.

This overlapping of video capturing and coding repeats and there is a delay of $2N_F$ frames in the coded bitstream. It can be shown that a new frame in the CB is skip only if the coding time is longer than the capture time. Therefore, the resulting average frame rate is determined either by d_5 or d_6, depending on how well t_6 is parallelized.

The transition t_6 is further expanded in Fig. 4. This Petri-net consists of rows of parallel transitions separated by synchronization points. The number of parallel transitions in each row equals N_P. The number of rows equals N_S, which is the number of synchronization points. In the net, each $S(i, j)$ denotes the sequence of tasks executed by processor j before the ith synchronization point. There is no other synchronization when $S(i, j)$ is executed.

The details of $S(i, j)$ is expanded in Fig. 5. In this Petri-net, $N_T(i, j)$ is the number of tasks in the sequence $S(i, j)$. These tasks are executed in one processor, which is assumed to have limited data cache size. Before execution of a task, the task data

is loaded into the cache from a global data server. After execution, the result is saved back to the server.

F. Theoretical Speedup Estimation

From Fig. 3, ignoring the time delays of d_2, d_3, d_4 and d_7 as they involve only swapping of pointers and skipping of frames, the significant delays are d_1 (frame capture), d_5 (Header-VLC) and d_6 (parallel task execution). As they are executed in parallel, the resulting frame coding time is

$$T_f = \max\{d_1, d_5, d_6\}. \tag{3}$$

From the expansion of t_6, and denoting the delay of $S(i, j)$ by $d(i, j)$, we have

$$d_6 = \sum_{i=1}^{N_S} \max_{j \in [1 \cdots N_P]} \{d(i, j)\}. \tag{4}$$

To determine $d(i, j)$, let us denote the computation delay, task loading delay, and saving delay by $T_P(i, j, k)$, $T_L(i, j, k)$ and $T_S(i, j, k)$, respectively, for the kth task in $S(i, j)$. From the expansion of $S(i, j)$, $d(i, j)$ equals to the sum execution delays of $N_T(i, j)$ tasks in $S(i, j)$ plus the leading task loading and trailing result saving delays, which is given by

$$\begin{aligned} d(i, j) = &T_L(i, j, 1) + T_S(i, j, N_T(i, j)) + T_P(i, j, 1) \\ &+ T_P(i, j, N_T(i, j)) + \sum_{k=2}^{N_T(i,j)-1} \\ &\cdot \max[T_P(i, j, k), T_S(i, j, k-1) \\ &+ T_L(i, j, k+1)]. \end{aligned} \tag{5}$$

Therefore, the resulting frame time is expressed as

$$T_f = \max \cdot \left\{ \begin{array}{l} d_1, d_5, \\ \displaystyle\sum_{i=1}^{N_S} \max_{j \in [1 \cdots N_P]} \left\{ \begin{array}{l} T_L(i, j, 1) + T_S(i, j, N_T(i, j)) \\ + T_P(i, j, 1) + T_P(i, j, N_T(i, j)) \\ \displaystyle + \sum_{k=2}^{N_T(i,j)-1} \max \left[\begin{array}{l} T_P(i, j, k), \\ T_S(i, j, k-1) \\ + T_L(i, j, k+1) \end{array} \right] \end{array} \right\} \end{array} \right\}. \tag{6}$$

Given the sequential frame time in (7), the speedup is the ratio between T_1 and T_f

$$T_1 = T_f|_{N_P=1}. \tag{7}$$

From the above equations, there are two conditions under which the scheme exhibits better performance. First, from (5), there is a constant delay due to the initial task loading and the trailing task saving, and there is further overhead if the task communication delay is larger than the task computation delay. The former delays cannot be hidden as such, but the latter communication delay can be hidden if $T_P(i, j, k) \geq T_L(i, j, k-1) + T_S(i, j, k+1)$, which depends on the communication channel bandwidth and the data size.

Secondly, the overall performance also depends on whether the loading is evenly distributed across the processors. From (4), it is the sum of a series of maximum delays, each represents

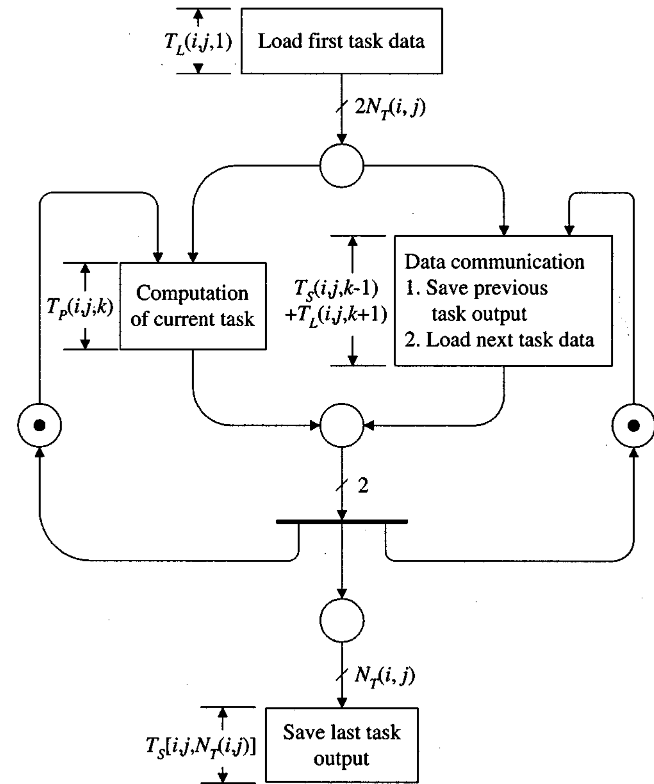


Fig. 5. Petri-net of task sequence $S(i, j)$.

a critical path between two synchronization points, that determines d_6. Therefore, d_6 is the smallest if workload is evenly distributed across the processors between each pair of synchronization points.

To further simplify the estimation, we assume the computation delay ($T_P(i, j, k)$) to be a random variable with Gaussian distribution. The mean and variance of the distribution are estimated from time measurement of a sequential execution. The mean and variance of $T_P(i, j, k)$ are used to determine T_1 and T_f as given by (6) and (7) when the communication delays are known.

G. Estimation of Communication Delay with Contention

For the communication delay, we assume a constant channel bandwidth with a constant initial setup delay. The parameters of the channel are estimated by an actual time measurement over the channel with different message size. We find that a linear model is applicable for processor-to-processor communication in systems such as the IBM SP2 [25] and the TMS320C80 [26]. To send a message of size M_C, let T'_C be the communication delay without contention, T_0 be the initial setup delay and W be the channel bandwidth, we have

$$T'_C = \frac{M_C}{W} + T_0. \tag{8}$$

As the frame data is served centrally, it is reasonable to expect a queue of requests pending on the server, which is served one by one. Assume a statistical queueing model [27], let n be the number of requests in the queue, $\lambda(n)$ be the request arrival rate, and μ be the request service rate, where both $\lambda(n)$ and μ are random variables with exponential distribution. Further

assume that all the processors generate requests at the same rate λ_0, and that a processor with a pending request in the queue does not generate requests until its pending request is served. Then the arrival rate at the queue is proportional to the number of processors that do not have pending request in the queue, or

$$\lambda(n) = (N_P - n) \cdot \lambda_0. \tag{9}$$

Let p_n be the probability of having n requests in the queue. At equilibrium, there is a set of local balance equations by equating the sum of flow of probability flux between adjacent states to zero, which are given as

$$\mu \cdot p_n = \lambda(n-1) \cdot p_{n-1}. \tag{10}$$

Solving this recursive equation for p_n gives

$$p_n = \left[\prod_{i=1}^{n} \frac{\lambda(i-1)}{\mu}\right] p_0 = \left[\left(\frac{\lambda_0}{\mu}\right) \frac{N_P!}{(N_P-n)!}\right] p_0. \tag{11}$$

To calculate p_0, we equate the sum of all probabilities to 1, which gives

$$p_0 = \frac{1}{1 + \sum_{n=1}^{\infty}\prod_{i=1}^{n} \frac{\lambda(i-1)}{\mu}} = \frac{1}{\left[\sum_{n=0}^{N_P} \left(\frac{\lambda_0}{\mu}\right)^n \frac{N_P!}{(N_P-n)!}\right]}. \tag{12}$$

By Little's Law [31], the mean delay of a request in the queue is given by

$$T_C = \frac{\overline{n}}{\overline{\mu}} = \frac{\sum_{n=1}^{\infty} n \cdot p_n}{\sum_{n=1}^{\infty} \mu \cdot p_n}. \tag{13}$$

From (13), T_C is the message transfer delay under contention. In general, T_C is greater than T_C', especially for large N_P. Fig. 6 depicts the ratio of T_C to T_C' versus N_P at different μ/λ_0. For small μ/λ_0, the ratio T_C/T_C' increases almost linearly and is large. It is because the request rate is larger than the service rate, resulting in long queue length. For large μ/λ_0, the request rate is smaller than the service rate, hence the server is able to keep the queue length short, and T_C/T_C' small.

To apply the above to the communication delay estimation, we first use the message size and the raw channel bandwidth without contention to estimate $T_S'(i, j, k)$ [or $T_L'(i, j, k)$] and then apply the queueing theory to obtain $T_S(i, j, k)$ [or $T_L(i, j, k)$]. This requires an estimate of the average request generation rate λ_0 and the service rate μ. As request is generated at the start of each task, the request generation rate is therefore the reciprocal of the mean task execution delay, and the service rate is equal to the reciprocal of the service delay $T_S'(i, j, k)$ [or $T_L'(i, j, k)$].

After each synchronization point, all the processors issue requests to the DS almost simultaneously. This transient period causes the longest loading delay when the DS serves the processors sequentially. In this estimation, the initial loading delay $T_L(i, j, 1)$ and the final saving delay $T_S(i, j, N_T(i, j))$ are multiplied by N_P to account for this transient period.

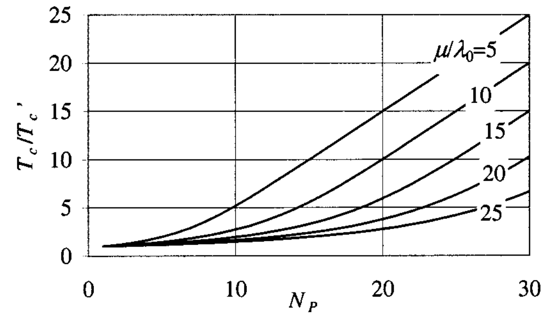


Fig. 6. Communication delay increases with contention.

H. Limitations and Applicability

This method assumes independent processing between MBs in a frame, which is not entirely true in some cases. For ME in H.263, if the long-vector feature is used under the *unrestricted motion vector* option, the search area of the ME is relative to the motion vector predictor. Since the predictor is obtained from the motion vectors of three nearby MBs, the ME of these MBs must be performed in a particular spatial order. This limitation only occurs if long vector of range $[-32, +31]$ is used.

When bit-rate control is concerned, none of the coding standards specifies how it is done. A commonly adopted method is to adjust the quantizer(s) base on factors such as discrepancy between the target and actual number of bits generated, and the estimated content of each MB in terms of variance. For parallel coding, the current number of bits can be placed in the DS so that all the processors can reference and update it. The granularity of quantizer adjustment can be at the MB, MB row or frame level. However, in H.263, there is a limit of plus or minus 2 on the quantizer relative to the left neighboring MB such that the validity of the quantizer is not known, until the neighboring MB quantizer has been determined. This restricts the coding of MBs in certain order.

Apart from the above limitations, this method does not impose any restrictions on the list of tasks performed when coding. Different coding standards in general may be represented by the task allocation scheme shown in Fig. 2, where precedence relationship could be accommodated using appropriate synchronization points. Similarly, the Petri-net representation as depicted in Fig. 3 can be applied to all four standards. The only difference between them would be t_6. Moreover, Figs. 4 and 5 are equally applicable to all cases disregarding the actual list of tasks performed in individual coding standards. Table I lists the possible tasks for each of the four coding standards. Out of the list, there may also be tasks such as coding mode determination, rate control and scalability options, which can be incorporated into the model without any restrictions.

III. Implementation on the TMS320C80

A. Development Board and Internal Architecture

To verify the methodology, the implementations were carried out on a TMS320C80 Software Development Board (SDB) [28]. It consists of 8 Mbytes on-board memory, called EXTMEM, that is used to store program code and data, hardware for video frame grabbing, video display, audio, interrupt control and PCI

TABLE I
POSSIBLE TASKS FOR THE 4 VIDEO CODING STANDARDS

H.261	H.263	MPEG-1	MPEG-2
ME PREDICTION (MC) ENC (DCT, QUANT, ZIGZAG) TC_VLC RECONSTRUCTION (IQUANT, IDCT, IMC) HEADER-VLC	P-MB MV PREDICTION P-MB ME B-MB MVD SEARCH P-MB PREDICTION B-MB PREDICTION ENC P-MB RECONSTRUCTION TC_VLC HEADER-VLC	P-MB ME B-MB ME PREDICTION ENC TC_VLC RECONSTRUCTION HEADER-VLC	FRAME/FIELD ME DUAL-PRIME FRAME/FIELD ME PREDICTION DCT TYPE ESTIMATION ENC TC_VLC RECONSTRUCTION

interface to the host PC. Both the hardware video frame grabbing and display can be done in real-time (30 fps) on different frame sizes.

Inside the TMS320C80 chip, there are four parallel processors (PPs) for number crunching and one master processor (MP) for program control, system management, and I/O. All of them access EXTMEM and the other hardware devices on the SDB through an on-chip communication processor: the transfer controller (TC). Requests to the TC from the processors and the video controller (VC) are queued in the form of packet transfer, where they are prioritized and serviced one at a time. The role of the VC is to handle all the frame grabbing and display facilities. Having a separate TC fits well with the communication hiding concept employed by our methodology, and the presence of the MP and VC allows the PPs to be dedicated to performing the coding tasks. There is also a total of 50-kB cache memory, which is divided into 25 2-kB cache blocks. These cache blocks are classified into parameter RAM (PRAM), instruction cache (IC), and data RAM. The PRAM is mainly used to store system parameters, such as the TC packet transfer table, system stack and interrupt vectors. Part of the PRAM is available for user applications too. The PPs and MP have a PRAM each. The purpose of the IC is to store recently accessed instruction codes. Each PP has one IC, whereas the MP has two. The data RAM is the working space of user programs. The two data RAM blocks in the MP work as data cache with automatic cache replacement mechanism. For the PPs, each has three data RAM blocks without cache replacement facility. They rely on the application program to handle all the data caching. This is, in fact, advantageous for parallel program analysis as the programmer would have more control over data movement. Finally, the communication of the MP, PP, TC, and cache blocks are through a dedicated crossbar switch. This crossbar switch enables random access of the cache blocks by the MP or PPs within a couple of cycles. This mechanism enables the triple buffering idea to be implemented easily.

In general, inter-processor communication between the PPs and MP is performed through different levels of the memory hierarchy. The first level is the on-board memory that can hold a large amount of data from which all the processors can access through the TC. The penalty for doing so is the slow access rate due to the initialization of the TC. The second level is through the on-chip data RAM, which can be accessed in two cycles by any one of the processors. The third level is through the use of the communication register (COMM). It is a very fast and effective way to provide an inter-processor signaling, but not data transfer. Since only a local register is accessed, there is no burden on the crossbar switch or the TC.

For the internal architecture of the PP [29], there is a certain degree of parallelism per instruction cycle. The main components of a PP are one data unit and two address units. The data unit composes of a splitable 32-bit ALU and a splitable 16×16-bit integer multiplier. Using specific assembly language instruction, the ALU can be configured as one 32-bit unit, two 16-bit units, or four 8-bit units. The multiplier can be configured to work similarly. In the extreme, the data unit can be configured to deliver up to six 8-bit integer operations per cycle. For the two address units, the global address unit covers a greater range of the memory address space than the local address unit. They can work simultaneously to allow two parallel data transfer of 64-bit word between the local cache memory and registers in one cycle. For accesses to other memory, such as the EXTMEM, the transfer must be done through the TC.

B. Implementation Issues

When implementing the H.261, N_S is set to 2 such that there are 2 synchronization points for each frame. The number of MBs is evenly distributed over the PPs, and the video capture, Header-VLC calculation and other system functions are done in the MP. The TC and EXTMEM together act as a data server for handling input frames, decoded frames and output bitstream. The $S(1, j)$ task consists of just ME, whereas the other eight tasks from MC, DCT to IMC for each MB are grouped together to form the $S(2, j)$ task termed ENC. The reason for choosing $S(1, j)$ and $S(2, j)$ in this way is mainly due to memory access locality.

For ME, out of the nine referenced MBs, six of them are also referenced by a neighboring MB. Therefore, maintaining MB data locality can save considerable communication delay. Overlapping of computation and data transfer is also allowed with a cache size of 12 referenced MBs plus two MBs for triple buffering of the MB under coding. The ME tasks are ordered by raster scan order and then partitioned into even sequences for allocation to each PP. Each PP has to follow the same ordering in processing the sequence of ME tasks allocated to it. With this arrangement, each task data communication involves loading three reference MBs, one input MB and saving a motion vector and the MB attributes, i.e., approximately $16 \times 16 \times 3 + 16 \times 16$ or 1024 bytes. For the ENC task, there is no overlap in memory access between different MBs. As such, the order

Fig. 7. Implementation scheme on the TMS320C80. (a) Task allocation to MP and VC. (b) Task allocation to PPs and TC. (c) Detailed task allocation to PPs and TC.

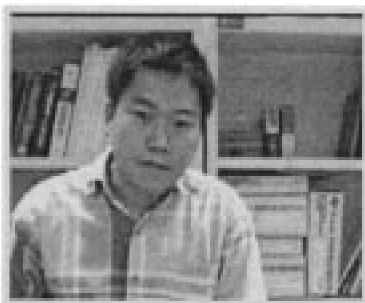


Fig. 8. A tested QCIF sample.

of ENC task processing has no influence on the communication cost. The task data communication involves saving a decoded MB, loading an input MB and a referenced MB, i.e., a total of $(16 \times 16 + 2 \times 8 \times 8) \times 3$, or 1152 bytes. Fig. 7 depicts the mapping of the model onto the TMS320C80 system.

IV. Results and Discussions

A. Measurement Criteria and Conditions

The results presented in this section have been obtained from timestamps taken from the actual implementation on the PPs.

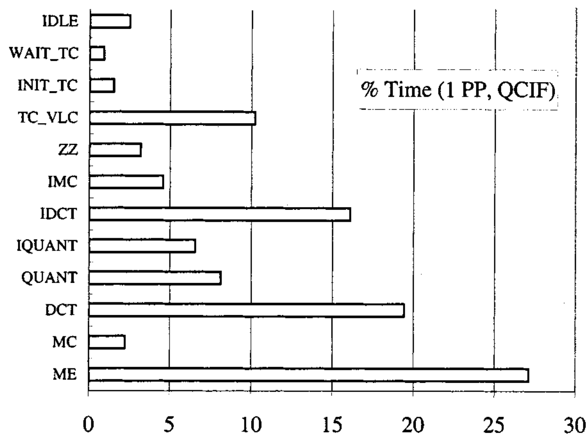


Fig. 9. Serial coding time break down.

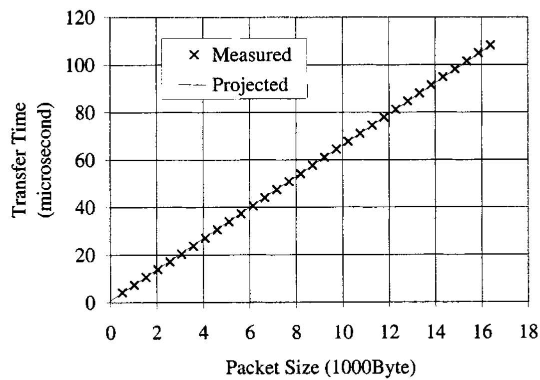


Fig. 10. TC data transfer time

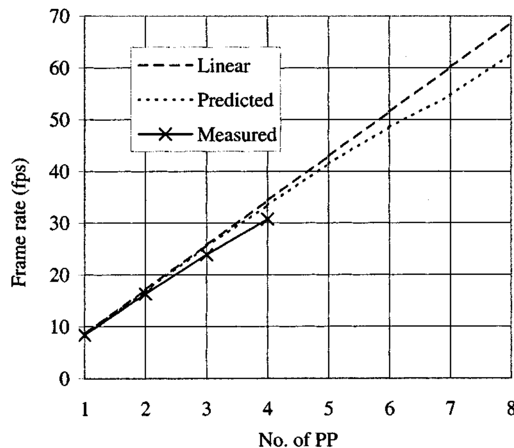


Fig. 11. Frame rate of QCIF.

All the PPs use a common clock tick generated once every 10 μs by the MP TIMER to make timestamps at various instances of program execution. At each clock tick, an interrupt is generated to the MP, which writes the updated clock tick value to a specified location in the PRAM of each PP. As it is, this incurs overhead to the MP, and the PPs may experience contention in accessing their PRAM if the MP is writing the clock tick value simultaneously. To reduce this contention to a negligible level, consecutive clock ticks are separated by 10 μs, which is equivalent to 400 instruction cycles on the PP. To achieve more accurate timestamps, finer clock tick may be used with the penalty of more frequent MP interrupts and PP PRAM contentions.

B. Serial Performance

As a baseline reference, a serial encoder was first executed on one PP and its average frame time was measured over 50 frames. A typical frame of the video captured is depicted in Fig. 8, which consists of a typical head-and-shoulder view of a person. The average frame time measured at frame sizes of 352×240 and QCIF (176×144) were 406.5 and 119.5 ms, respectively, i.e., 2.46 and 8.37 fps, respectively. The average percentage time breakdown for QCIF is shown in Fig. 9. The most time-consuming task is the ME as expected (27.1%), followed by the DCT (19.4%), IDCT (16%), and TC_VLC (10.2%). It should be noted that these figures are obtained after the assembly codes have been manually optimized. The rest ranges from 8.1% (QUANT), 6.5% (IQUANT), to just below 1%. On the chart, INIT_TC (1.5%) is the time spent on initialization of packet transfer table, and WAIT_TC (0.9%) is the time spent on waiting for data transfer to complete. Furthermore, there is a component called IDLE (2.5%) which is the time spent on doing neither communication nor computation. This is caused by the imbalance of workload among the PPs in the multiple PP case, plus the waiting time for the MP initialization. Since there is only one PP in this case, 2.5% represents mainly the waiting time for the MP initialization.

The bandwidth of the TC without contention was estimated by measuring the time for transferring messages from EXTMEM to the on-chip RAM and vice versa. For different message sizes, a number of measurements were conducted and the average time was taken. As depicted in Fig. 10, W and T_0 are estimated to be 153 Mbytes/s and 0.8 μs, respectively.

C. Parallel Performance

For the parallel performance, the theoretical performance was first predicted based on Section II-F and II-G, while the actual performance of the implementation was measured under the criteria mentioned in Section IV-A and compared with the predicted and ideal linear performance. Figs. 11–13 depict the frame rate, speedup and efficiency for coding the QCIF video. In Fig. 11, the predicted and measured frame rate rises almost linearly up to four PPs. The measured frame rate achieved is 30.7 fps using four PPs, while the speedup is 3.67, as shown in Fig. 12. The almost linear speedup indicates that the system has successfully hidden most of the communications and has negligible processor idling delay. This may be explained as the mean computation delay being around 300 and 800 μs for the ME and ENC tasks, respectively. The communication delay without TC contention is about 9.5 and 13.4 μs for the two tasks, respectively, (with contention, they are 10.3 and 14.0 μs, respectively, for four PPs). In both cases, the communication delay is substantially smaller than its computation delay, which can be effectively hidden for each PP. It should also be noted that the measured performance is slightly less than the predicted performance. This is due to the fact that parallelization overhead has not been taken into account in the theoretical model.

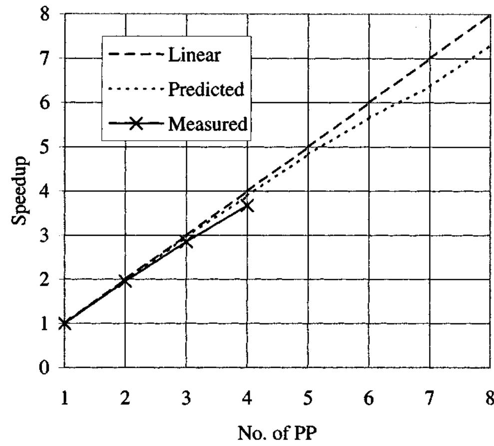


Fig. 12. Speedup of QCIF.

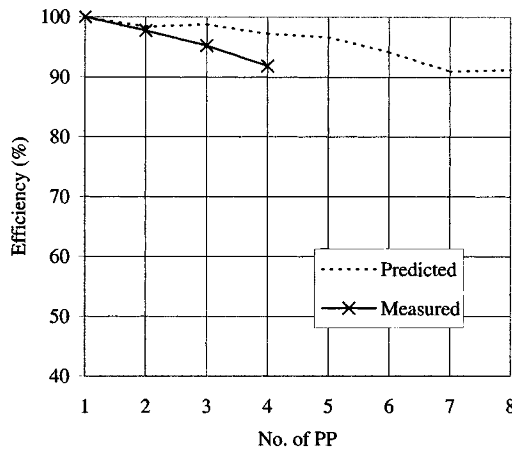


Fig. 13. Efficiency of QCIF.

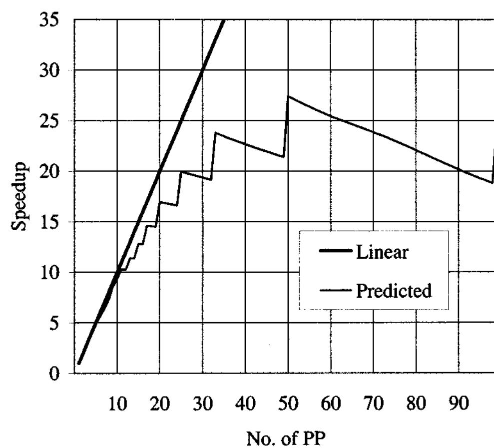


Fig. 14. Projected speedup for QCIF.

In Fig. 13, the efficiency is calculated as the percentage ratio between the measured or predicted speedup and the linear speedup. Practical parallel algorithms would have efficiency less than 100% due to parallelization and communication overheads. In this case, the predicted efficiency is over 90% for up to eight PPs, whereas the measured efficiency for up to four PPs is also over 90%. This is some 10% better than any parallel implementation reported so far for small or large N_P. However, if the measured efficiency is projected for larger

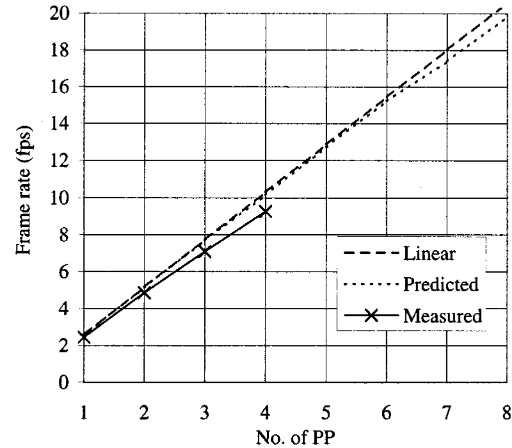


Fig. 15. Frame rate of 352×240.

N_P, the trend seems to be a more rapid decrease beyond four PPs. Even with this behavior, the method presented here still compares favorably with other published methods.

To illustrate how the model behaves beyond $N_P = 8$, Fig. 14 depicts the predicted speedup up to 100 PPs. As can be seen, the estimated speedup rises steadily and almost linearly up to 10 ($N_P = 10$), beyond which the shape of the curve becomes stepwise. This is due to the small number of MBs, M, integrally divided by the number of PPs. As a result, there tends to be uneven distribution of MBs on the PPs if M is not divisible by the number of PPs. For this reason, some PPs complete computation earlier than the others and have to wait. This reduces the speedup and becomes worse when the number of PPs having to wait is in majority. This stepwise characteristic may be smoothed and improved if the workload is balanced [30]. In fact, when the number of PP approaches the number of MBs, i.e., 99, the initial and final communication delays dominate and there will be no benefit in using more PPs. In theory, a finer granularity should give smoother speedup and extend farther with more processors. However, this is not true in practice since the communication overhead increases rapidly with the number of processors because, first, due to the contention of communication channel, the communication delay increases with the number of processors. Second, with finer granularity, data sharing between tasks will increase which results in duplicated communication. In our current implementation, we find that the TC can support finer granularity since the current computation delay is greater than the communication delay (between 30–60 times). For tasks with larger data size, due to the limitation of the cache size, finer granularity may be necessary.

For 352×240, the measured and predicted performance have similar trend to that of QCIF. The frame rate and speedup figures are depicted in Figs. 15 and 16, respectively. A frame rate of 9.25 fps has been achieved for $N_P = 4$. Extending our prediction, 30 fps is achievable at around $N_P = 14$, i.e., if parallelization overhead is included, four C80 will probably give 30 fps. The measured speedup for four PPs is 3.76. In general, the speedup for the 352×240 video is better than QCIF because of the larger amount of data involved in the former case. From the percentage efficiency depicted in Fig. 17, it is observed that the predicted efficiency remains well over 90% up to $N_P = 8$. The

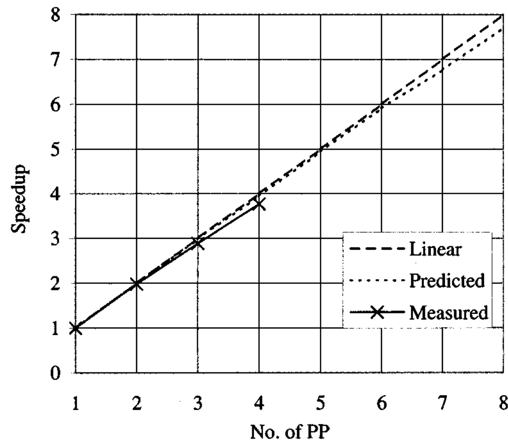


Fig. 16. Speedup of 352×240.

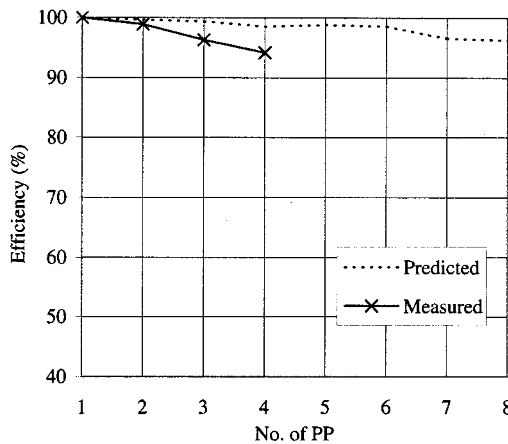


Fig. 17. Percentage efficiency of 352×240.

measured efficiency however, showed the same tendency as in QCIF where the parallelization overhead becomes more significant with large N_P. Although the measured efficiency is 94% at $N_P = 4$, if the curve is extended, the projected measured efficiency would be down to 85% for $N_P = 8$.

In Fig. 18, the predicted speedup is almost linear up to $N_P = 20$, beyond which the increase is stepwise and reaches a maximum of 48, as compared with 27.5 for QCIF. The reasons why this figure is larger than the QCIF case are, firstly, for 352×240 video, there are more tasks in the sequence. As a result, the initial task loading and final saving delays in each task sequence constitute a smaller proportion of delay to the overall sequence execution time, giving a higher speedup. Second, as there is a synchronization point between the PPs at the end of each task sequence, the expected overall execution time is the expected maximum of the N_P sequence execution time. A large standard deviation in sequence execution time implies a large maximum time among N_P sequences and a large proportion of processor idling time. Assume each task has a mean execution time m and standard deviation σ. When n such tasks are executed, the overall mean and standard deviation becomes nm and $\sqrt{n}\sigma$ respectively. Therefore, the standard deviation increases in proportion to $\sqrt{n}$ and is slower than the increase in the mean. As such, the variation relative to the mean decreases with increasing

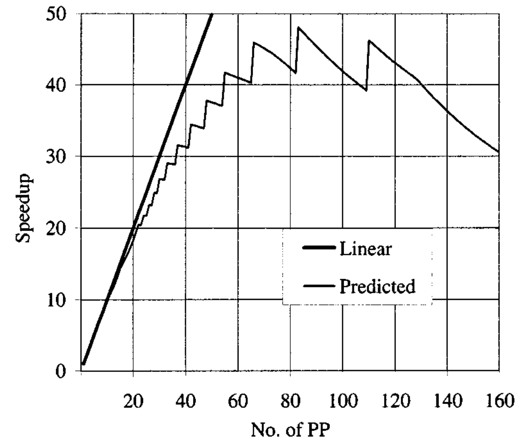


Fig. 18. Projected speedup for 352×240.

number of tasks in the sequence. Therefore, the expected percentage idling time should be smaller with sequences having larger number of tasks.

V. Conclusion

In conclusion, a new parallelization methodology for video coding using the concept of communication hiding has been successfully developed and implemented in this paper. It considers task/data size, processor cache capacity and communication contention in a practical manner, through the application of Petri-nets and task graphs. From that, the performance of the parallel method can be theoretically studied and analyzed. This approach is appropriate because, firstly, when task and/or data decomposition is oriented toward matching the capacity of the cache, substantial number of cache misses can be avoided. Secondly, communication contention often contributes significantly and realistically to the delay in accessing remote data. Being able to take this into consideration helps to come up with a closer prediction of the actual performance of the implementation, which further enables us to refine the implementation. As the parallel method is reasonably independent from the target video-coding standard, it is fair to assume that the method is equally applicable to the remaining three standards. In fact, full implementation has been tested on H.261, and the H.263 standard was also implemented based on this method with very similar performance characteristics.

From the measured results, it can be observed that first, the predicted performance and measured performance are very similar. Second, frame rates of 30.7 and 9.25 fps have been achieved for QCIF and 352×240 video, respectively, with only one TMS320C80. Comparing with other practical implementations (excluding simulations), these results are very respectable. Third, system efficiency of over 90% has also been achieved. Both the speedup and efficiency are due to the parallelization method as well as how optimized the serial encoder codes are. We noticed a marked performance increased after the assembly codes have been manually optimized. Fourth, this parallelization method is particularly suitable for small N_P. For QCIF, the measured speedup is almost linear for $N_P \leq 10$, whereas for 352×240 video, the measure speedup is almost linear for $N_P \leq 20$.

REFERENCES

[1] "ITU-T recommendation H.261: video codec for audiovisual services at p × 64 kbits," International Telecommunication Union, 1990.

[2] "MPEG-1: Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s," ISO/IEC 11172, 1993.

[3] "ITU-I recommendation H.263: video coding for low bitrate communication," International Telecommunication Union, 1995.

[4] "MPEG-2: Generic coding of moving pictures and associated audio," ISO/IEC 13 818, 1995.

[5] "Overview of the MPEG-4 standard," ISO/IEC JTC1/SC29/WG11 N1730, 1997.

[6] "MPEG-7: context and objectives (v.5 - Fribourg)," ISO/IEC JTC1/SC29/WG11 N1920, 1997.

[7] L. Torres and M. Kunt, *Video Coding: The Second Generation Approach*. Norwell, MA: Kluwer, 1996.

[8] C. Huang and J. L. Wu, "New generation of real-time software-based video codec: Popular video coder II (PVC-II)," *IEEE Trans. Consumer Electron.*, vol. 42, pp. 963–973, Nov. 1996.

[9] K. Li and H. Yuen, "A high performance image compression technique for multimedia applications," *IEEE Trans. Consumer Electron.*, vol. 42, pp. 239–243, May 1996.

[10] F. Sijstermans and J. Van der Meer, "CD-I full-motion video encoding on a parallel computer," *Commun. ACM*, vol. 34, no. 4, pp. 81–91, 1991.

[11] S. M. Akramullah, I. Ahmad, and M. L. Liou, "Performance of software-based MPEG-2 video encoder on parallel and distributed systems," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 7, pp. 687–695, Aug. 1997.

[12] K. Shen, L. A. Rowe, and E. J. Delp, "Parallel implementation of an MPEG-1 encoder: faster than real time," *SPIE*, vol. 2419, pp. 407–418, Feb. 1995.

[13] K. Shen and E. J. Delp, "A spatial-temporal parallel approach for real-time MPEG video compression," in *Pro. 25th Int. Conf. Parallel Processing*, 1996, pp. II100–II107.

[14] N. H. C. Yung and K. K. Leung, "Parallelization of the H.261 video coding algorithm on the IBM SP2 multiprocessor system," in *Proc. IEEE Int. Conf. Algorithm, Architectures and Parallel Processing*, 1997, pp. 571–578.

[15] S. M. Akramullah, I. Ahmad, and M. L. Liou, "Software-based H.263 video encoder using a cluster of workstations," *SPIE*, vol. 3166, pp. 266–273, Jul. 1997.

[16] Y. Yu and D. Anastassiou, "Software implementation of MPEG-II video encoding using socket programming in LAN," *SPIE*, vol. 2187, pp. 229–240, Feb. 1994.

[17] I. Agi and R. Jagannathan, "A portable fault-tolerant parallel software MPEG-1 encoder," *Multimedia Tools and Applic.*, vol. 2, pp. 183–197, 1996.

[18] H. Mooshofer, A. Hutter, and W. Stechele, "Parallelization of a H.263 encoder for the TMS320C80 MVP," Texas Instruments, Paris, France, SPRA339, Sept. 1996.

[19] W. Lee, J. Golston, R. J. Gove, and Y. Kim, "Real-time MPEG video codec on a single-chip multiprocessor," *SPIE*, vol. 2187, pp. 32–43, Feb. 1994.

[20] T. Akiyama *et al.*, "MPEG-2 video codec using image compression DSP," *IEEE Trans. Consumer Electron.*, vol. 40, pp. 466–472, 1994.

[21] C. Bouville, P. Houlier, J. L. Dubois, I. Marchal, B. Thebault, and M. Klefstad, "DVFLEX: A flexible MPEG real time video codec," in *Proc. IEEE Int. Conf. Image Processing (ICIP'96)*, vol. II, 1996, pp. 829–832.

[22] K. K. Leung, "Parallelization methodology for video coding - an implementation on the TMS320C80," Dept. Elect. and Electron. Eng., Univ. Hong Kong, Hong Kong, Res. Rep., May 1998.

[23] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. New York: McGraw-Hill, 1993.

[24] David and Rene, *Petri Nets and Grafcet: Tools for Modeling Discrete Event Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[25] C. B. Stunkel *et al.*, "The SP2 high-performance switch," *IBM Syst. J.*, vol. 34, no. 2, pp. 185–204, 1995.

[26] (1995, Mar.) TMS320C80 (MVP) Transfer Controller User's Guide. Texas Instruments, USA, Doc. Code: SPRU105A. [Online] Available: http://www.ti.com

[27] T. G. Robertazzi, *Computer Networks and Systems—Queuing Theory and Performance Evaluation*. New York: Springer-Verlag, 1994.

[28] *TMS320C8x Software Development Board Technical Reference*: Texas Instruments, USA, Doc. Code: SPRU178, 1996.

[29] *TMS320C80 (MVP) Parallel Processor User's Guide*: Texas Instruments, USA, Doc. Code: SPRU110A, 1995.

[30] N. H. C. Yung and K. C. Chu, "Fast and parallel video encoding by workload balancing," in *Proc. IEEE SMC'98*, Oct. 1998, pp. 4642–4647.

[31] J. D. C. Little, "A proof of the queuing formula $L = W$," *Oper. Res.*, vol. 9, pp. 383–387, 1961.

Kwong-Keung Leung (S'97) received the B.Sc. and M.Sc. (with distinction) degrees in 1990 and 1997, respectively, from the Department of Electrical and Electronic Engineering, University of Hong Kong, where he is currently working toward the Ph.D. degree.

His research interests include multiprocessor scheduling, dynamic load balancing, heterogeneous computing, and parallelization of multimedia systems.

Nelson H. C. Yung (S'82–M'85–SM'96) received the B.Sc. and Ph.D. degrees from the University of Newcastle-Upon-Tyne, U.K., in 1982 and 1985, respectively.

He was Lecturer at the same university from 1985 to 1990, involved in the R&D of digital image processing and parallel processing. From 1990 to 1993, he was Senior Research Scientist with the Department of Defence, Australia, where he headed a team on the R&D of military-grade signal analysis systems. He joined the University of Hong Kong in late 1993 as an Associate Professor, where he leads a research group in Digital Image Processing and Intelligent Transportation Systems. He is the founding Director of the Laboratory for Intelligent Transportation Systems Research, and has published over 90 research papers.

Dr. Yung serves as Reviewer for the IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS PART B, IEEE TRANSACTIONS ON SIGNAL PROCESSING, IEE PROCEEDINGS PART G, *SPIE Optical Engineering, HKIE Proceedings*, and the *Microprocessors and Microsystems Journal*. He is a Chartered Electrical Engineer, and a Member of the HKIE and IEE. His biography is published in *Marquis' Who's Who in the World*.

Paul Y. S. Cheung (M'82–SM'92) was born in Hong Kong. He received the B.S. and Ph.D. degrees in electrical engineering from Imperial College, University of London, London, U.K., in 1973 and 1977, respectively.

After working for two years, he returned to Hong Kong in 1978 and taught at Hong Kong Polytechnic. In 1980, he joined the University of Hong Kong, where he is currently the Dean of the Faculty of Engineering. In addition, he instructs three courses and supervises undergraduate and postgraduate students.