Real-Time Imaging **2**, 187–199 (1996)

UWICL: A Multi-Layered Parallel Image Computing Library for Single-Chip Multiprocessor-based Time-Critical Systems

Many software libraries have been created to support the commonly used primitive operations needed in image processing, image analysis and image understanding. Generally, these libraries are based on the *single-layered* Application Program Interface (API). While a single-layered API provides the useful abstraction level to interact with the library and hides unnecessary implementation details from the user, it does not produce an efficient program when a new algorithm is implemented by assembling the selected existing library routines. The composed program suffers from the inefficient data movement and additional loop control overhead. Furthermore, when a system employs a highly integrated processor such as a single-chip multiprocessor, the single-layered API prevents the user from fully utilizing the resources available in the system.

In this article, we describe the University of Washington Image Computing Library (UWICL), the *multi-layered* high-performance parallel image computing library for Texas Instruments TMS320C80 Multimedia Video Processor (MVP)-based time-critical systems. Our goal in designing the UWICL is to provide the TMS320C80 user community with efficient and flexible image computing library routines. The UWICL provides three levels of APIs to the programmers under the multi-layered organization, the MVP-level API, the DSP-level API, and APIs for data flow and processing cores. By optimizing the processing core functions, we have achieved high performance in the individual function level, and by allowing the sub-primitive library routine composition, we can achieve efficient image processing application development, avoiding most problems encountered in using the single-layered library routines. The performance of the multi-layered organization *vs*. the single-layered one is analysed and compared using the Canny's edge detection algorithm as an example. The balanced composition based on the multi-layered organization outperforms the single-layered composition by 14 to 41% depending on the system's memory bandwidth available.

As an adjunct to the UWICL, we have also developed an integrated MVP performance monitor (MPM). The MPM can identify the performance bottleneck of the TMS320C80 applications and can be used in optimization by enabling the user to select the most efficient library composition level in building the application with the UWICL. In order to provide the overall performance evaluation model of the MVP, the simple MVP functional model has also been defined in the MPM. For the image thresholding operation, the difference between the measured execution time and the analysis

1077-2014/96/030187 + 13 $18.00

prediction is less than 2%. The design and implementation of the MPM, and the applicability and usefulness of the MPM and MVP performance model are described in this article.

Jihong Kim* and Yongmin Kim†‡

**Department of Computer Science and Engineering, Box 352350*
University of Washington, Seattle, WA 98195
E-mail: jihong@cs.washington.edu
†Department of Electrical Engineering, Box 352500
University of Washington, Seattle, WA 98195, USA
E-mail: kim@ee.washington.edu

Introduction

Many tasks in image processing and computer vision applications require a large number of computing cycles. For example, in order to convolve a 512×512 image with a 5×5 kernel, over 13 million multiplications and additions are necessary. In addition, as applications require more time-critical tasks (e.g., real-time road traffic monitoring [1] and interactive telemedicine for remote consultation/diagnosis), an even higher level of computing power is necessary. To meet the heavy computing requirements of various imaging applications in a cost-effective way, many specialized programmable digital signal processors (DSPs) such as Texas Instruments TMS320 series (e.g., TMS320C40 [2]) have been developed and used to support these applications.

Traditionally, high-performance DSPs have several unique architectural features optimized for signal processing such as single cycle multiplier, multiple operations per cycle and zero overhead looping. These performance-enhancing features, however, often make it more difficult for a compiler to produce the efficient code. So, DSPs are typically programmed in assembly language for time-critical applications. Because of the high development cost in writing the efficient DSP programs, there exist many software libraries which are optimized for specific DSPs. For example, there are numerous libraries available for Texas Instruments TMS320-series DSPs [3].

The library routines work as building blocks from which more complex applications can be assembled quickly. Typically, libraries are organized to provide the *single-layered* Application Program Interface (API) to the programmers. The single-layered API is defined in the primitive operation level and provides the useful abstraction level to interact with the library, hiding the implementation details of the library routines. However, when an application is developed by assembling the selected library routines, its performance tends to suffer from the inefficient data movement and additional loop overhead. This is because the single-layered API library routine is usually optimized at the individual function level. When several library routines are combined, some portions of the library routines become redundant. However, they are included in the application because it is not possible for the user to exclude redundant segments with the single-layered API.

For example, consider the simple image algebra, **A** + **B** * **C**, where **A**, **B** and **C** are images of the same resolution, and the + and * operators indicate pixel-by-pixel addition and multiplication, respectively. If this is implemented by two library routines, say, `img_add()` for an image addition operation and `img_multiply()` for an image multiplication operation, the resulting program shows generally poor performance compared to the one programmed manually without using library routines. The library routines assembled may take extra steps for loop control and data movement, and use redundant temporary space. If the **A** + **B** * **C** computation were implemented without using library routines, we could easily eliminate the additional loop control and data movement, and dispense with additional temporary space needed. Even though individual routines are well optimized, the composition overhead may limit the usefulness of library routines significantly, especially when the library routines are designed to support time-critical applications for specific DSPs.

Furthermore, the single-layered API is not appropriate to utilize the available resources effectively in highly integrated processors such as a single-chip multiprocessor with large on-chip memory [4–7]. In such processors, because of the large penalty associated with accessing off-chip memory, it may be more efficient to apply a sequence of primitive operations to one section of input

‡Correspondence should be addressed to: Y. Kim.

data (while residing in on-chip memory) at a time in a tiled fashion, instead of moving data back and forth to the off-chip memory whenever each primitive operation in the sequence is performed. However, since the single-layered API is defined in the primitive operation level, it is not possible to combine multiple library routines to process only one section of input data at a time. For example, in our example of **A** + **B** * **C**, it will be more efficient to perform both addition and multiplication on the single blocks from three images (which are brought into the on-chip memory) at a time, avoiding unnecessary data movements. Unfortunately, the single-layered API does not allow this sub-primitive level library routine composition because it was designed to process whole input images by one primitive at a time.

In order to solve the inefficiency problem of the library routine composition, the chaining mechanism was proposed [8], which allows the delayed execution of library routine invocation. By deferring execution, library routines can be chained together in a sequence in the order in which they are invoked, allowing some opportunities for further optimization. For example, if chaining is requested for **A** + **B** * **C**, more efficient codes can be dynamically generated and linked into the executing program, or the library can use one of the auxiliary routines for the common combinations of operations. While the chaining mechanism shows some promise of enhancing the performance of the library routine composition, its general applicability is somewhat limited. In order for the chaining mechanism to be efficient and general, the efficient dynamic code generation is necessary. But for many advanced DSPs available today, the efficient code generation from the high-level languages is not always possible. If the dynamic code generation and linkage are not supported in the library, the chaining would benefit only in a finite number of combinations that have been predetermined by the library designers, not by the library users. Furthermore, even with the chaining mechanism, the library interface is still single-layered and could not support the sub-primitive level library routine composition necessary for the efficient programming of the highly integrated processors.

In this article, we describe the design and implementation of the University of Washington Image Computing Library (UWICL), the *multi-1ayered* high-performance parallel image computing library for Texas Instruments TMS320C80 Multimedia Video Processor (MVP)-based *time-critical* systems. The TMS320C80 is a *single-chip multiprocessor* with many powerful and unique features optimized for multimedia, video compression, image/signal processing and computer graphics [9]. While the TMS320C80 offers high performance over a wide range of imaging applications, efficiently programming the TMS320C80 to maximize its power is a major challenge. Therefore, our main goal in developing the UWICL was to provide the TMS320C80 user community with *efficient* and *flexible* image computing library routines which support most of the fundamental image computing algorithms. In order to support the efficient library routine composition as well as the highly optimized individual routines, and allow more flexible library routine composition taking full advantage of the highly integrated architecture of the TMS320C80, the UWICL provides three levels of API to the programmers, one of which is in the *sub-primitive* operation level. We call this organization *multi-layered* API approach. If applications are implemented by *carefully* combining sub-primitive level library routines, their performance could be significantly better than that of the single-layered API organization. We demonstrate here the advantage of the multi-layered API approach using the Canny's edge detection algorithm [10] as an example.

On the other hand, if sub-primitive level library routines are *carelessly* composed, the resulting performance could be worse than that of the single-layered API organization. In order to help the UWICL library users select the most efficient library composition level for their application, some form of performance monitoring of the composed application was necessary. We have developed an integrated performance modeling and monitoring tool for the TMS320C80, the MVP Performance Monitor (MPM), to support performance monitoring of the composed applications as well as individual library routines. The MPM identifies various performance bottlenecks for the given TMS320C80 program such as cache misses, on-chip shared memory contentions, and synchronization overhead. In this article, we describe both the UWICL and MPM in detail.

The TMS320C80 Processor and MS5000 System

In this section, we briefly describe the TMS320C80 (MVP) processor and a TMS320C80-based multimedia system, the MediaStation 5000 (MS5000). The MS5000 system has been used as a test system for the UWICL. (For the detailed description, see references [9,12,13] for the TMS320C80 and reference [11] for the MS5000 system.)

Overview of TMS320C80 Processor

The TMS320C80 can be described as a single-chip, heterogeneous, MIMD multiprocessor connected via a crossbar to

multiple on-chip shared memory modules. It combines a RISC processor and four advanced DSPs as well as an intelligent direct memory access (DMA) controller and two video controllers into a single-chip device. It is capable of processing more than 2 billion operations per second (BOPS) with the 2.4 Gbytes/sec on-chip data transfer rate. In order to reduce the data transfer overhead with the external memory/devices, the large on-chip memory (25 2-kbyte modules) is provided as well.

Figure 1 shows a high-level block diagram of the major functional blocks of the TMS320C80. The Master Processor (MP) is a general-purpose RISC processor with an integral IEEE 754 compatible floating-point unit. In a typical operation mode, the MP serves as the main supervisor and distributor of tasks within the TMS320C80. Also, the MP is the preferred processor for performing floating-point operations. The four Parallel Processors (PPs) or advanced DSPs (DSP 0-3) have a highly parallel architecture optimized for multimedia, video/image compression, image/signal processing and computer graphics. Each DSP is capable of performing up to 15 RISC-equivalent operations in a single clock cycle via a long instruction word (64 bits) mechanism and has many powerful features not found in conventional DSPs. They include:

(i) Single-cycle accesses to large on-chip memory, allowing two 32-bit data transfers per processor in every cycle concurrently with data operations.
(ii) Three-operand 32-bit arithmetic and logical unit (ALU) which can be optionally split into two 16-bit or four 8-bit units.
(iii) Multiple flags (`mf`) register which captures the multiple status results (flags) from split-ALU operations, and expander which takes 1, 2, or 4 bits in the `mf` register and replicates them 32, 16, or 8 times.
(iv) Barrel rotator which can pre-rotate an input to the ALU by 0 to 31 bits.
(v) Three zero-overhead hardware loop controllers which allow three levels of nested loops to be controlled with no associated overhead.
(vi) Dedicated adders for address generation which can also be used for arithmetic operations.
(vii) Conditional ALU and data transfer operations which substitute many compare-and-branch operations.

The Video Controllers (VC) provide supports for programmable video timing to control both capture and display. The processors and on-chip shared-memory modules are fully interconnected through the high-performance crossbar switch network.

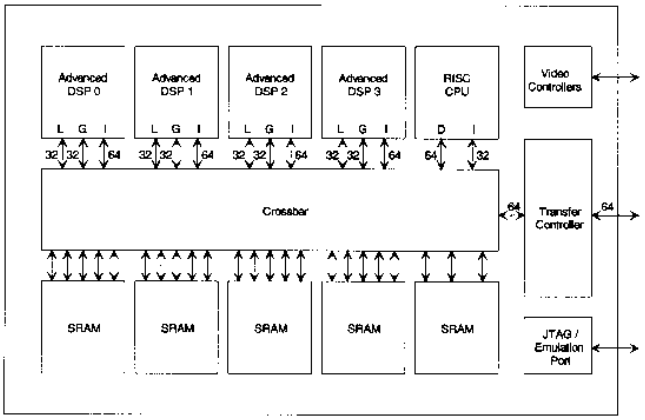

Figure 1. High-level block diagram of the TMS320C80.

While five processors (the MP and four DSPs) provide the computing power for the TMS320C80, the Transfer Controller (TC), a dedicated memory controller with sophisticated data transfer logic, manages all the data transfer requests and cache misses from these processors. The TC prioritizes different types of data transfer requests and transfers data within and between the on-chip and external memories. Because of the high data bandwidth required for image computing algorithms and the overhead of accessing off-chip memory directly, five processors typically work with data brought into the on-chip shared memory by the TC. Since the processors and the TC can operate in parallel, most data movement by the TC is hidden from the processors in the optimized implementation; while a processor is working on the current block of data residing in the shared memory, the TC is servicing a request for the next block in parallel. The TC which works as a dedicated memory controller for the whole MVP chip supports highly sophisticated data transfer logic. The TC's main data transfer mechanism is a packet transfer (PT), a transfer of data blocks between two areas of the MVP memory. Packet transfers are initiated by the MP, DSPs, VC or external devices as requested to the TC under the software or hardware control. Once a processor has submitted a transfer request, it can continue program execution without waiting for the completion of the transfer. Many different modes of packet transfers are available such as multi-dimensional transfers, table-guided transfers, fill-with-value, and serial register transfers (SRT) [13].

Overview of MS5000

The MS5000 system based on the TMS320C80 is a highly integrated desktop multimedia system implemented on a single personal computer (PC) plug-in board. It transforms

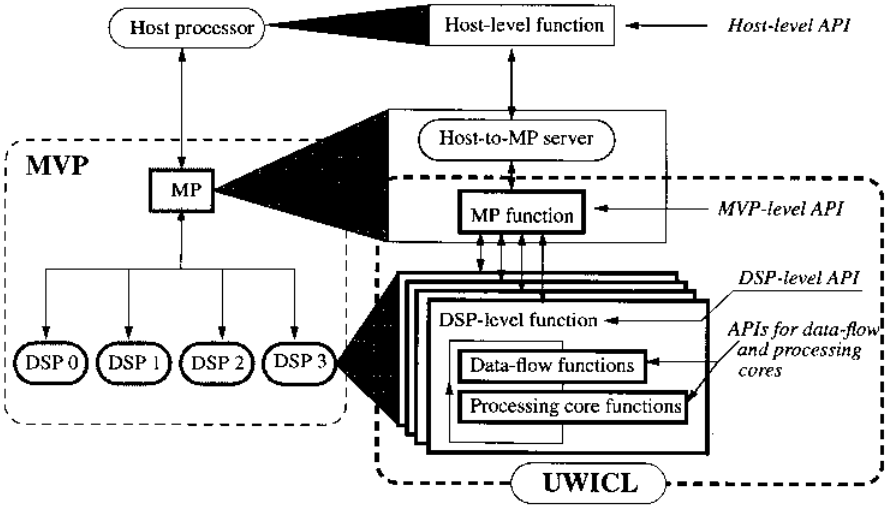

Figure 2. Overall structure of the UWICL program and its mapping to the MVP.

a PC into a programmable high-performance multimedia and imaging workstation, off-loading the host computer from computer-intensive tasks using a TMS320C80, custom programmable logic devices and other supporting chips. It supports real-time MPEG compression and decompression [14] and still-picture processing [15] as well as image processing and 2D/3D computer graphics. The MS5000 has several built-in special I/O features such as a video digitizer and a stereo audio coder/decoder for multimedia applications as well.

Overview of UWICL

In programming the MVP, three main components are typically required. The first component is the communications between the MP and the host and between the MP and the DSPs. The host processor issues a service request to the MP which in turn may off-load the image computing task to the DSPs. Such a service request typically involves the passing of parameters between the host, the MP and the DSPs. The second component of an MVP program is the transfer of data between the external memory and on-chip shared-memory associated with each DSP. These transfers usually take place by issuing packet transfer (PT) requests to the TC. The third component is the processing of the data inside the DSPs. The four DSPs can operate in parallel, pipelined or many different modes.

Corresponding to these three components, we have divided an MVP program into three *hierarchical* segments in the UWICL. The overall structure of the UWICL program and its mapping to the MVP are shown in Figure 2. The first segment is the MP module whose main function is to interact with the host and the DSPs. The MP module includes MP functions, and a simple server program (host-to-MP server in Figure 2) which interacts with the host and invokes the appropriate MP function for servicing a host request. The MP function receives the commands and parameters such as the addresses of the input and the output image locations from the host and is responsible for dividing the task among the DSPs, passing the parameters and issuing commands to the DSPs. These MP functions form the highest level of three-layered APIs, the MVP-level API, in the UWICL.

The second segment is the DSP-level functions. The entry point for the DSP-level function is passed to the DSP from the MP when the MP function configures a task on each DSP. The DSP-level function also receives the parameters from the MP and sets up all the required PTs to move the data between the external memory and DSP's on-chip memory using the data-flow functions. Once the data are available in the on-chip memory, the DSP-level function calls the processing core functions for the actual computing with on-chip data. The DSP-level functions form the mid-level API of the UWICL while the processing core and data-flow functions provide the lowest-level API. The division of the DSP-level task into the processing and data-flow parts is based on the typical operation mode of many image computing algorithms, i.e., *the repetition of structured operations to the sequence of localized regions.* For example, in the convolution operation with a $p \times p$ kernel, p^2 multiplications and p^2-1 additions are repeatedly applied between the sequence of $p \times p$ sub-images and the $p \times p$ kernel.

In programming these algorithms at the lowest level, there are two main issues: (i) data-flow programming which specifies how to divide an image into localized regions, and when and where to move localized regions and their processed output, and (ii) processing programming which specifies how to produce the desired output utilizing data from localized regions. Since most of data processing take place in the processing core functions, the processing core functions are implemented in the DSP assembly language for the maximum efficiency.

A hierarchical organization of the UWICL for a simple image invert operation is illustrated in Figure 3. The host-level API, **host**_img_invert(), is linked to the MVP-level API, **mp**_img_invert(), by the host interface and related driver routines. The **mp**_img_invert() routine divides the input image into four subimages, passes the addresses of one input subimage and its corresponding output subimage locations to each DSP and starts the DSP-level routine, **dsp**_img_invert(), on each DSP. The **dsp**_img_invert() routine then calls the data-flow library routine, **dataflow**_set_up_row_by_row_transfer(), which sets up the packet transfer tables for bringing one row of the input image into on-chip memory and writing back one row of the inverted image to the external output image location. The actual data transfer takes place when the **dataflow**_get_next_row(), **dataflow**_request_next_row()* or **dataflow**_write_back_output_row() routines are executed. These routines submit the prepared packet transfers to the TC. Once data are read into on-chip memory, the processing core library routine, **proc_core**_img_invert(), inverts image data and stores the results temporarily in on-chip memory before they are written back to their output image locations.

The multi-layered and modular design approach used in the UWICL provides the user with a flexible architecture. The user may choose any of three hierarchical APIs for building an application depending on the application's performance requirement and the user's familiarity with the MVP programming. A novice user may choose the packaged MVP-level API which uses all the MVP resources in a predetermined way. A more experienced user who would like to customize the use of some of the MVP resources may choose the DSP-level API. Still, a more advanced MVP user who understands the intricacies of the MVP's data transfer protocol and DSP programming in assembly may choose only the data-flow and processing

* Both routines read one row of the input image into on-chip memory. However, in **dataflow**_get_next_row(), the DSP is busy-waiting until the data transfer is completed while, in **dataflow**_request_next_row(), the DSP continues its processing after submitting the packet transfer.

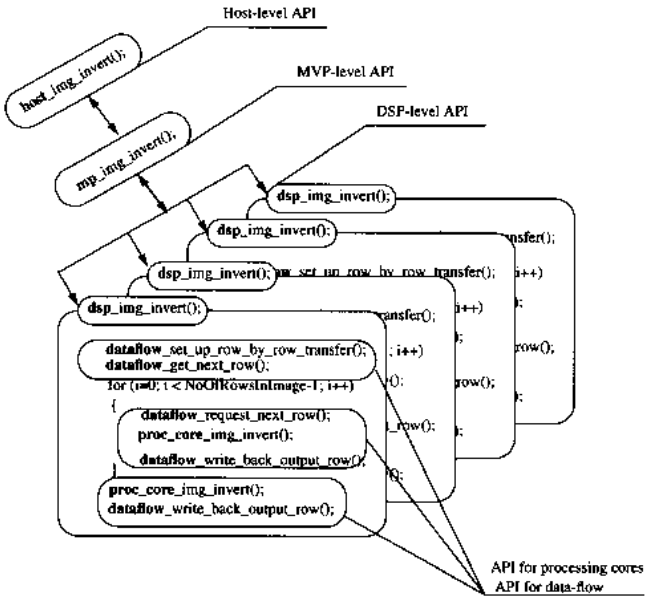

Figure 3. Multi-layered API organization for an image invert operation.

core API. The MVP-level API allows the users to build their applications quickly with minimal coding and understanding. But the composed application program is large in size, and the large external memory is necessary for storing the temporary outputs. On the other hand, if the processing core API is used and the composed application does not introduce any new performance overhead, the better performance can be obtained with the smaller code size and no extra memory required. However, more custom coding (both in the MP and DSPs) would be necessary.

UWICL Performance

In this section, we discuss the performance of the UWICL library routines. Then, the UWICL implementation of the Canny's edge detection algorithm is described as an example to demonstrate the efficient composition of a higher-level routine under the UWICL's multi-layered organization.

Performance of Individual Library Functions

The current release of the UWICL (Version 1.1) supports 94 image computing functions including most low-level image processing functions such as filtering operators (e.g., convolution and median filtering), morphological operators (e.g., erosion and dilation), histogram operators (e.g., histogram equalization), unitary transform operators (e.g., Fourier transform and discrete cosine transform) and

geometric operators (e.g., rotate, zoom and warping). It includes arithmetic and logical operators (e.g., addition and exclusive-OR) and segmentation operators (e.g., threshold) as well. Table 1 shows the execution time of the selected UWICL MVP-level library routines implemented on the MS5000 with the TMS320C80 running at 50 MHz. 512×512 8-bit test images have been used. As listed in Table 1, most functions run in real time (less than 33 ms). All these functions utilize four DSPs operating in parallel. In most functions, each DSP is assigned with a quarter of the image, and runs to its completion independent of the others.

High performance of the UWICL routines shown in Table 1 also indicates that no significant degradation in performance is introduced in the individual function level because of the UWICL's multi-layered organization. In order to measure the overhead of using the multi-layered organization over the single-layered full assembly implementation in the DSP level, we have implemented several functions in both organizations and compared their performance. Because of the added flexibility in the multi-layered organization, the performance of the UWICL routines cannot exceed that of the full assembly implementation. However, the performance difference was negligible because of the efficient hierarchical decomposition in the UWICL as well as the highly optimized implementation of processing core library routines. For most functions, the performance difference between the multi-layered and single-layered organizations was less than 5%.

In implementing the processing core library routines, several key features of the TMS320C80 described earlier are heavily utilized to speed up execution. For example, consider the thresholding operation which produces an 8-bit binary output image from an 8-bit input image. If the input pixel is greater than or equal to the threshold value, the corresponding output pixel is set to the user-selected non-zero output pixel value. Otherwise, they are set to zero. In the first instruction, the 32-bit ALU is split into four 8-bit units, and the differences of four input pixels and a threshold value are calculated simultaneously in each DSP. The carry bits from this operation are saved in the `mf` register. The carry bit is set if the input pixel is larger than or equal to the threshold value. In the next instruction, the saved four least significant bits of the `mf` register are replicated 8 times using the expander, and four bitwise logical AND operations with the user-specified output pixel value are performed. If the carry bit was 1, the bitwise logical AND operation is performed between the user-specified pixel value and 0xFF. If the carry bit was 0, the bitwise logical AND operation is performed between the user-specified pixel value and 0x00. Therefore, thresholding can be performed in 0.5 cycles per pixel on each DSP. With four DSPs running concurrently, 8 pixels can be thresholded in every cycle. This demonstrates the MVP's processing power when programmed in an optimal fashion. A similar technique is used as well to optimize the implementation of 3×3 median filtering.

Table 1. Execution time of various UWICL MVP-level library routines on the MS5000 system with a 512x512 8-bit image

Operation	Execution time (ms)
Thresholding	3.6
3×3 convolution	19.4
3×3 median filter	10.7
8×8 block-based discrete cosine transform	9.5
16-bit to 8-bit mapping	8.5
Binary erosion or dilation with a 5×5 structuring element	12.7
Wavelet transform of up to 6 levels with Daubechies' D4 wavelet	27.7
Histogram equalization	8.3

Example Application: Canny's Edge Detection Algorithm

The Canny's edge detection algorithm is a popular method of computing higher-precision edge images [10]. It is based on a computational approach to edge detection. Its various implementations differ in several details such as how to establish the gradient direction and how to suppress non-maximal edge points. Our implementation is similar to the one described by Canny [10] and involves the following tasks:

- **task 1:** convolve the input image I by a 3×3 horizontal Canny kernel to get the horizontal component of the gradient image, G_x.
- **task 2:** convolve the input image I by a 3×3 vertical Canny kernel to get the vertical component of the gradient image, G_y.
- **task 3:** compute the absolute value images, $|G_x|$ and $|G_y|$.
- **task 4:** compute the approximate gradient magnitude image M by adding $|G_x|$ and $|G_y|$.
- **task 5:** compute the approximate gradient direction image D by encoding one of four directions (N-S, E-W, NE-SW, and NW-SE) for each pixel.
- **task 6:** refine the rough boundary by non-maximum suppression. For each pixel $I(i, j)$, the gradient magnitude $M(i, j)$ is compared with two neighboring pixels' (along the gradient direction $D(i, j)$) gradient magnitude values. If the gradient magnitude $M(i, j)$ of the pixel $I(i, j)$ is the maximum among three pixels, it is unchanged. Otherwise, it is set to zero. The final gradient magnitude image contains the edge image.

(a) (b)

Figure 4. Canny's edge detection example: (a) original image and (b) edge image.

Figure 4 shows the input image and its corresponding edge image produced by the above algorithm implemented on the MS5000 system.

Because of the UWICL's multi-layered architecture, there are many variations in mapping these tasks into the UWICL library routines. For example, all the operations can be mapped to the MVP-level functions or to the processing core functions. For the best performance, we have implemented the above six tasks into two MVP-level functions, one for tasks 1–5 and the other for task 6 as shown in Figure 5. This division of tasks resulted in a balanced implementation between the processing and data transfer times and did not introduce any new overhead from combining the five processing core functions in constructing the first MVP-level function. If we had combined all six tasks in the processing core function level, we would have introduced extra cache misses in each iteration because of the large program size, and the resulting performance would have been much worse than that of using two MVP-level functions.

In order to demonstrate the composition efficiency of the UWICL's multi-layered approach over the single-layered one, we have also implemented the Canny's edge detection by chaining six separate MVP-level functions. If we had used the single-layered API approach for the UWICL, the Canny's edge detection algorithm would have been implemented this way. Table 2 summarizes the execution time of two implementations measured on the MS5000. A 512×512 input image was used, and the 3×3 window was used for convolution and non-maximal suppression. The multi-layered columns correspond to the balanced implementation with two MVP-level functions while the single-layered columns correspond to the implementation with six MVP-level functions. The balanced composition outperforms the full MVP-level composition (thus, the single-layered API approach) by 14 to 41% depending on the data bandwidth available. The balanced composition avoids any extra I/O time by sharing the common data flow for the five tasks, thus improving performance significantly over the single-layered composition, especially when slower and narrower memories are used as the source and destination. The performance advantage of the multi-layered organization is amplified when the off-chip data bandwidth is limited because the MVP-level functions need to spend more time in data I/O.

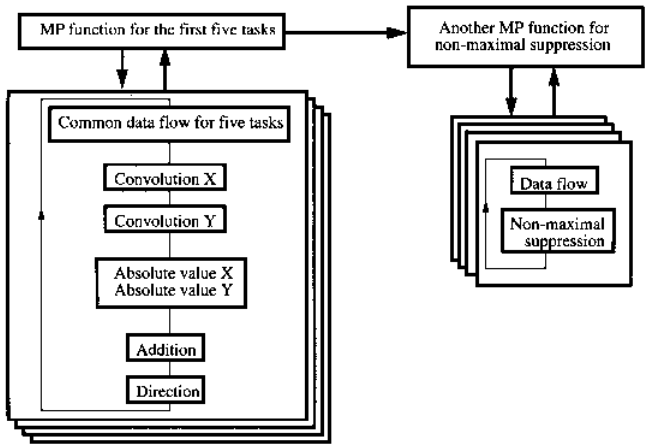

Figure 5. Balanced UWICL implementation of the Canny's edge detection algorithm.

Table 2. Execution time of two implementations of the Canny's edge detection algorithm on the MS5000 using the UWICL library routines

	Faster memory[a]		Slower memory[b]	
	Multi-layered	Single-layered	Multi-layered	Single-layered
Horizontal convolution	19.8 ms	19.8 ms	19.8 ms	19.8 ms
Vertical convolution	19.8 ms	19.8 ms	19.8 ms	19.8 ms
Two absolute values	1.6 ms	7.2 ms	1.6 ms	21.4 ms
Two additions	0.8 ms	5.4 ms	0.9 ms	16.0 ms
Gradient direction	5.4 ms	5.6 ms	5.4 ms	16.4 ms
Non-maximal suppression	17.6 ms	17.6 ms	17.6 ms	17.6 ms
Total	65.0 ms	75.4 ms	65.1 ms	111.0 ms
Difference	10.4 ms		45.9 ms	

[a]64-bit bus width and 2 cycles per access
[b]32-bit bus width and 3 cycles per access

MVP Performance Monitor (MPM)

The MVP is a multiprocessor *system* even though it is a single-chip DSP. In order to develop an efficient MVP program, a programmer should have a good understanding of not only the algorithm and DSP's intricacies, but also the overall system's performance. Without the system-wide performance understanding, the MVP program can suffer from the overheads such as the resource conflicts (e.g., on-chip shared-memory access contentions among the DSPs) and the unbalanced synchronization (e.g., one DSP's excessive waiting for the other DSPs). These extra overheads could significantly degrade the overall performance of the MVP program. Unfortunately, these overheads are often difficult to predict and identify even for the experienced programmers. For example, in the MVP, each DSP can access all the on-chip shared-memory modules except others' instruction caches through the crossbar switch network. No (or minimal) crossbar switch contentions among the DSPs would be desirable for efficient algorithm implementation. However, it is difficult without any systematic tool support to flnd out the number of contentions in the MVP program. Even if the number of contentions were found, it would not be easy to determine if the crossbar contention is really the performance bottleneck.

The MVP Performance Monitor (MPM) has been developed to help the MVP programmers to understand the potential problem areas easily and identify performance bottlenecks among potential candidates. It supports three types of performance monitoring (cache monitoring, contention monitoring and custom monitoring) and works with the simple MVP performance model. The MPM could improve not only the performance of custom MVP programs, but also that of an application implemented by composing the UWICL library routines. As discussed earlier, the application implemented from the processing core API can be more efficient than one built at the higher-level APIs such as MVP-level API. However, if the composition of the processing core library routines introduces any new performance penalty such as extra cache misses due to the increased code size of the developed application, the performance might be worse than that of the application implemented from the higher-level library routines. For example, in the Canny's edge detection algorithm, if the combined processing cores had not fitted nicely into the DSP's instruction cache, the extra cache misses that occur in each iteration would have degraded the overall performance significantly. The MPM can be used to detect such cases and help the application developers to select the most efficient API level for their applications.

Architectural Overview of MPM

The overall architecture of the MPM is shown in Figure 6. The MPM was designed to work together with the Texas Instruments MVP Debugger tool which is widely used in developing the MVP program. The core of the MPM is the extended MVP simulator. The extended MVP simulator consists of the MVP C++ simulator which was developed by Texas Instruments, the customized MPM extensions to the MVP C++ simulator which include three types of monitoring support and TC debugging capability, and the communication and synchronization module (CSM) which is responsible for communicating with the MPM user interface. The MPM user interface (which was implemented using the Tcl/Tk languages) spawns a child process which monitors the user-specified monitoring event [16]. The communication between the MPM user interface and the monitoring process is supported using the **send** command from the Tk language while the communication between the monitoring process and the

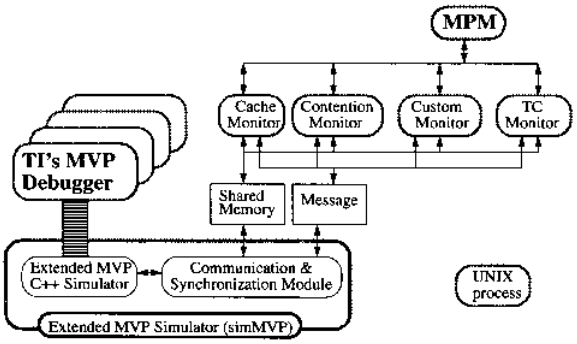

Figure 6. Overall software architecture of the MPM.

CSM of the extended MVP simulator is supported through the shared memory and message type UNIX interprocess communication utilities.

Figure 7 shows the snapshot of the MPM environment with a cache monitoring example. The whole simulation starts from the MPM user interface (Area 1). Once the MPM is started, the TI MVP Debugger is started (Area 2) for simulation of the MVP program. The user typically sets a breakpoint at the start address of a code segment *S* which will be monitored by the MPM. If the simulation is stopped at the breakpoint, the user selects an appropriate type of monitoring from the MPM user interface, and the user interface for the selected monitoring event appears (Area 3). In Figure 7, cache monitoring was selected. The user then sets another breakpoint at the end address of the code segment *S* and continues simulation. When the simulation stops, the user can examine the monitoring result on the separate window (Area 4). For cache monitoring, information on the cache misses for the code segment *S* is displayed, including the total number of cache misses, the total cache-miss service time, the total number of non-compulsory cache misses, and the total cache-miss service time for non-compulsory cache misses. It also displays the summary of all the cache misses in the table where the source and destination addresses for the cache miss, the average service time for each cache miss and the frequency of the cache miss are displayed. Based on this information, the user can restructure the MVP program or reduce the program size to improve the performance. Contention monitoring works in a similar fashion and displays the total number of crossbar switch contentions for each processor.

Custom monitoring is different from cache and contention monitoring in that the event in custom monitoring is

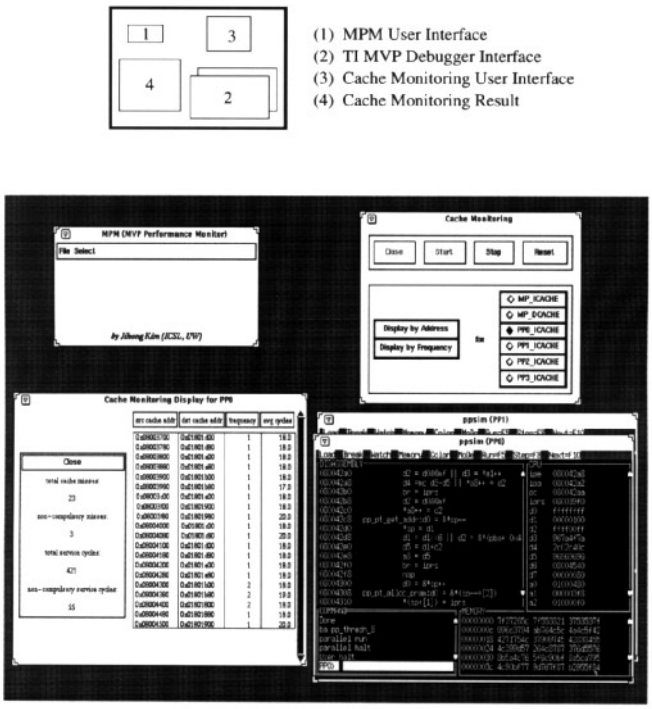

Figure 7. Snapshot of the MPM environment.

defined by the user, not predetermined by the MPM. In the current version of the MPM, a user-defined event is specified by DSP checkpoints which are the addresses of interesting DSP instructions. For example, when the user is interested in knowing how long the DSP waits for the packet transfer completion, the address of the DSP polling instruction which checks for the packet transfer completion could be set as a DSP checkpoint. The result from custom monitoring is displayed in a graph, e.g., Figure 8. The *x*-axis of this graph indicates the MVP cycle numbers. In this example, the DSP polling instruction for the packet transfer completion was set as the DSP checkpoint. Figure 8 shows that this polling instruction is executed for a large number of cycles (a solid line in the lower row), meaning that the DSP3 is wasting a number of cycles waiting for the completion of the requested packet transfer. The lines in the upper row of Figure 8 indicate the status of the packet transfers submitted by the DSP3. The thick line indicates the time interval when the packet transfer service is delayed because the TC is servicing the higher priority requests while the thin line shows the time interval when the requested packet transfer is actually serviced by the TC. The user can measure each of these intervals by clicking the mouse button.

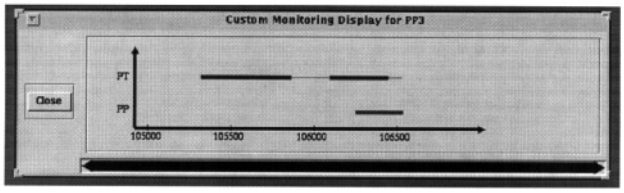

Figure 8. An example display for the custom monitoring result.

MVP Functional Model

In designing a functional model for the MVP, we have assumed a simple interaction model between the MP and the DSPs: they interact with each other only when the MP initiates the DSPs and the DSPs inform the MP of the completion of their subtasks. With this assumption, the execution time, $\mathbf{t}_{MVP,A}$, of a task *A* on the MVP with *p* DSPs is given by

$$\mathbf{t}_{MVP,A} = \mathbf{t}_{mp-initialization} + \mathbf{t}_{mp-wrapup} + \max_{k \in \{i_1,\ldots,i_p\}} \left(\mathbf{t}_{DSP(k),invoke-waiting} + \mathbf{t}_{DSP(k),subtask(A,k)} \right) \quad (1)$$

where $\mathbf{t}_{mp-initialization}$ is the MP's initialization time including the DSP subtask configuration and parameter construction, and $\mathbf{t}_{mp-wrapup}$ is the MP's time spent in cleaning up after all the subtasks are completed. The DSP's waiting time before getting started by the MP is $\mathbf{t}_{DSP(k),invoke-waiting}$, and the execution time of a subtask on the DSP is $\mathbf{t}_{DSP(k),subtask(A,k)}$. Figure 9 illustrates the MVP functional model with a task running on four DSPs. In this example, the DSP1 takes the most time to complete its subtask.

The subtask execution time $\mathbf{t}_{DSP(k),subtask(A,i)}$ on the DSP(*i*) is given by

$$\mathbf{t}_{DSP(i),subtask(A,i)} = \max\left(\mathbf{t}_{compute}, \mathbf{t}_{i/o}\right) \quad (2)$$

where $\mathbf{t}_{compute}$ is the processing time for *subtask*(*A*,*i*) and $\mathbf{t}_{i/o}$ is the data transfer time for *subtask*(*A*,*i*). If $\mathbf{t}_{i/o}$ is greater than $\mathbf{t}_{compute}$, the subtask is I/O-bound. Otherwise, it is compute-bound. $\mathbf{t}_{compute}$ and $\mathbf{t}_{i/o}$ can be defned by

$$\mathbf{t}_{compute} = \mathbf{t}_{pure-processing} + \mathbf{t}_{dsp-xbar-cont} + \mathbf{t}_{cache-miss-overhead} + \mathbf{t}_{busy-waiting} \quad (3)$$

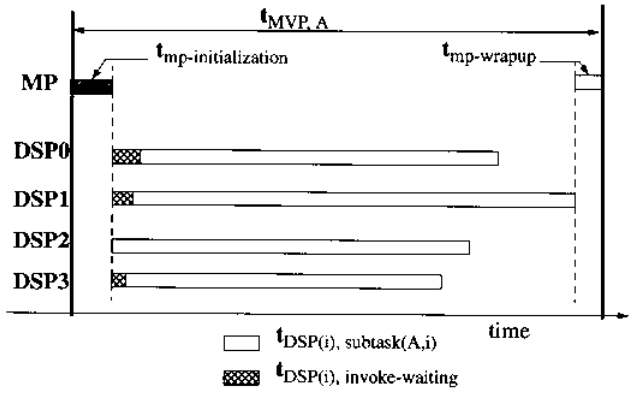

Figure 9. An example task analysis with the MVP functional model of equation (1).

$$\mathbf{t}_{i/o} = \mathbf{t}_{pure-i/o} + \mathbf{t}_{i/o-waiting} + \mathbf{t}_{pt-overhead} + \mathbf{t}_{mem-overhead} + \mathbf{t}_{tc-xbar-cont} + \mathbf{t}_{pt-waiting-overhead} \quad (4)$$

where $\mathbf{t}_{pure-processing}$ and $\mathbf{t}_{pure-i/o}$ are the pure processing and data transfer time without any delay, respectively. $\mathbf{t}_{dsp-xbar-cont}$ and $\mathbf{t}_{tc-xbar-cont}$ are time delayed because of the crossbar switch contentions with other processors. $\mathbf{t}_{cache-miss-overhead}$ is the total cache-miss service time, and $\mathbf{t}_{pt-waiting-overhead}$ is the total waiting time related to the packet transfer services. $\mathbf{t}_{pt-waiting-overhead}$ includes the initial waiting time (before the submitted packet transfer is actually serviced by the TC) and the suspended service time (because of the TC being preempted by other higher-priority TC requests). $\mathbf{t}_{busy-waiting}$ is the total waiting time related to any type of synchronization among the DSPs. $\mathbf{t}_{i/o-waiting}$ is the total waiting time before the first data transfer request is submitted to the TC from the DSP. $\mathbf{t}_{pt-overhead}$ is the total overhead related to managing the packet transfer information, and $\mathbf{t}_{mem-overhead}$ is the total overhead related to the external memory system organization and depends on the memory type, page size, refresh rate. Figure 10 illustrates the subtask functional model. The example subtask runs on the DSP1 and is compute-bound because $\mathbf{t}_{compute}$ is larger than $\mathbf{t}_{i/o}$.

Table 3 summarizes how to determine 13 parameters used in the MVP functional model. There are four categories. The parameters in the first group are ignored in performance evaluation since their contribution is negligible in the most image computing routines. For example, $\mathbf{t}_{DSP(i),invoke-waiting}$ is very small because all the DSPs start almost simultaneously. The parameters in the second group are manually calculated by analysing the DSP program and system organization. $\mathbf{t}_{pure-processing}$ and $\mathbf{t}_{pure-i/o}$, for

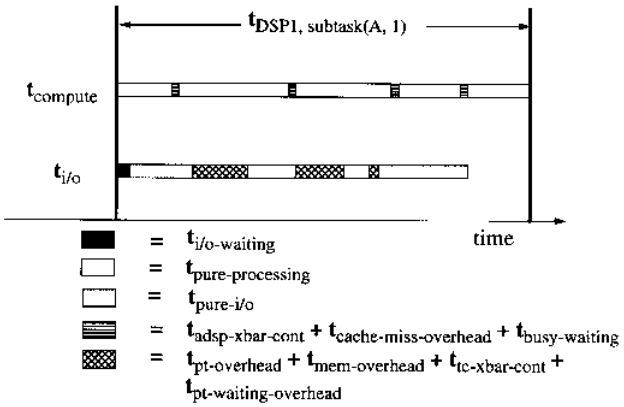

Figure 10. An example subtask analysis with the subtask functional model of equation (2).

example, are easily computed from the DSP program once the data-flow and processing core functions are finalized. The parameters in the third group are experimentally measured from the actual system. $t_{pt\ -\ overhead}$ is computed by multiplying the measured overhead per packet transfer (about 15 cycles) with the total number of packet transfer submissions in the DSP program. $t_{i/o\ -\ waiting}$ is determined by actually measuring the execution time of the DSP initialization part which is executed before any data transfer is started. The parameters in the last group are monitored by the MPM. They include the overheads from cache misses, crossbar switch contentions, synchronization among the processors, and idling packet transfer service cycles.

Example: Image Thresholding

In order to demonstrate the accuracy of the MVP functional model and the MPM's monitoring capability, the performance of the MVP-level image thresholding library routine on the MS5000 was analysed in detail. As described earlier, image thresholding was very efficiently implemented on the DSPs and the overall implementation becomes I/O-bound. Table 4 lists the calculated, measured and MPM-monitored times of various I/O parameters derived for a single DSP, DSP1, with four DSPs running in parallel. The total subtask execution time on the DSP1 with the functional model was 3.65 ms, 72% of which come from $t_{pt\ -\ waiting\ -\ overhead}$. This large $t_{pt\ -\ waiting\ -\ overhead}$ value is caused by the long waiting time before the TC can start servicing packet transfers submitted by the DSP1. Due to the I/O-bound nature of the image thresholding algorithm on the MVP, the TC is usually servicing other DSPs' packet transfers when the DSP1 submits its packet transfer requests. We have measured the actual execution time on the MVP of the MS5000 system (with four DSPs running in parallel) to compare with the analysis result. The measured execution time was 3.60 ms, a 1.4% deviation from the predicted execution time. The closeness of two numbers shows the accuracy of the model and the MPM.

Table 3. Summary on how the MVP functional model parameters are determined

Category	Parameter
Ignored	$t_{mp\ -\ initialization}$
	$t_{mp\ -\ wrapup}$
	$t_{DSP(i),invoke\ -\ waiting}$
Calculated	$t_{pure\ -\ processing}$
	$t_{pure\ -\ i/o}$
	$t_{mem\ -\ overhead}$
Measured	$t_{i/o\ -\ waiting}$
	$t_{pt\ -\ overhead}$
MPM Monitored	$t_{dsp\ -\ xbar\ -\ cont}$
	$t_{tc\ -\ xbar\ -\ cont}$
	$t_{cache\ -\ miss\ -\ overhead}$
	$t_{busy\ -\ waiting}$
	$t_{pt\ -\ waiting\ -\ overhead}$

Table 4. Performance analysis example with an image thresholding operation

Category	Parameter	Time (ms)
Calculated	$t_{pure\ -\ i/o}$	0.66
	$t_{mem\ -\ overhead}$	0.12
Measured	$t_{i/o\ -\ waiting}$	0.11
	$t_{pt\ -\ overhead}$	0.15
MPM Monitored	$t_{tc\ -\ xbar\ -\ cont}$	0.00
	$t_{pt\ -\ waiting\ -\ overhead}$	2.61
	total	3.65

Conclusion

We have described the UWICL, a multi-layered parallel image computing library for the TMS320C80-based time-critical systems. Our goal in designing the UWICL was to provide the TMS320C80 user community with efficient and flexible image computing library routines. The UWICL provides three levels of APIs to the programmers under the multi-layered organization, the MVP-level API, the DSP-level API and APIs for data flow and processing cores. By optimizing the processing core functions, we have achieved high performance in the individual function level. By allowing the sub-primitive library routine

composition, we can achieve efficient image processing algorithm developments, avoiding most problems encountered in using the single-layered library routines, such as additional data movement, extra memory requirement and extra loop control overhead. Using the Canny's edge detection algorithm, we have demonstrated the process and performance advantage of the multi-layered organization over the single-layered one in composing new applications using the UWICL routines. Two implementations of the Canny's edge detection algorithm showed that by sharing the common data flow in several processing cores, we can improve the performance of the composed application significantly compared to the performance with the single-layered library routines, especially when the I/O bandwidth is limited. Furthermore, two lower-level APIs allow the user to configure the TMS320C80 flexibly with many different combinations of the DSPs and subtasks.

As an adjunct to the UWICL, we have also developed the MPM, an integrated MVP performance monitor. The MPM can identify the performance bottleneck of the TMS320C80 applications and can be used in optimization by enabling the user to select the most efficient API level in building the application using the UWICL. In order to provide the overall performance evaluation model of the MVP, the simple MVP functional model was also defined in the MPM.

Currently, we are extending both the UWICL and the MPM. The first and second releases of the UMCL were distributed in 1995 through the industry consortium for the UWICL, and we are working for the next release in 1996. The next release will support many more image computing algorithms such as color image processing, color space conversion, image registration and generalized warping. Over the next few years, we are planning to support enough functions to implement the PIKS foundation compliance profile (of ISO/IEC image processing and interchange standard) which specifies a minimally compliant level of PIKS functionality [17,18]. The MPM is also going to be extended to support the custom monitoring more efficiently. For example, in the current version, only one type of the user-defined event can be specified for a single monitoring session. It can be extended to support the multiple number of the user-defined events simultaneously within one monitoring session.

References

1. Ali, A. T. & Dagless, E. L. (1992) A parallel processing model for real-time computer vision-aided road traffic monitoring, *Parallel Process Lett.* **2**(3): 257–264.
2. Texas Instruments (1991) *TMS320C4x User's Guide.*
3. Texas Instruments (1994) *TMS320 Software Cooperative: A Library of Standard DSP Building Blocks.*
4. Araki, T., Toyokura, M., Akiyama, T., Takeno, H., Wilson, B. & Aono K. (1994) Video DSP architecture for MPEG2 codec, in *Proceedings IEEE 1994 International Conference on Acoustics, Speech and Signal Processing*, pp. 417–420.
5. Veendrick, H., Popp, O., Postuma, G. & Lecoutere, M. (1994) A 1.5 GIPS video signal processor (VSP), in *Proceedings IEEE 1994 Custom Integrated Circuits Conference*, May 1994, pp. 95–98.
6. Goodenough, J., Meacham, R. J., Morris, J. D., Seed, N. L. & Ivey, P. A. (1994) A general purpose, single chip video signal processing (VSP) architecture for image processing, coding and computer vision, in *Proceedings IEE Colloquium on Parallel Architectures for Image Processing*, May 1994, pp. 1–4.
7. Young, I. & Tomisawa, O. (1993) Microporcessors in the year 2000, in *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pp. 202–203.
8. Cok, D. R. & Cok, R. S. (1992) A chaining and extension mechanism for image processing software, in *Proceedings SPIE Image Processing and Interchange: Implementation and Systems*, **1659**, pp. 192–203.
9. Guttag, K., Gove, R. J. & Van Aken, J. R. (1992) A single-chip multiprocessor for multimedia: The MVP, *IEEE Computer Graph. Appl.*, **12**(6): 53–64.
10. Canny, J. (1986) A computational approach to edge detection, *IEEE Transactions on Pattern Anal. Mach. Intell.*, **8**(6): 679–698.
11. Lee, W., Kim, Y., Gove, R. J. & Read, C. J. (1994) MediaStation 5000: integrating video and audio, *IEEE MultiMedia*, **1**(2): 50–61.
12. Gove, R. J. (1994) The MVP: a highly-integrated video compression chip, in *Proceedings 4th IEEE Data Compression Conference*, pp. 215–224.
13. Texas Instruments (1994) *TMS320C8x (MVP) Online Reference (Release 1.00).*
14. Le Gall, D. (1991) MPEG: a video compression standard for multimedia applications, *Comm. ACM*, **34**(4): 46–58.
15. Wallace, G. K. (1991) The JPEG still picture compression standard, *Comm. ACM*, **34**(4): 30–44.
16. Ousterhout, J. K. (1994) *Tcl and the Tk Toolkit*. Reading, MA: Addison-Wesley, 1994.
17. Image processing and interchange functional specification, part 2: programmer's imaging kernel system application program interface, ISO/IEC, JTC1 SC24 IS 12087-2, 1994.
18. Pratt, W. K. (1992) An overview of the ISO/IEC programmer's imaging kernel system application program interface, in *Proceedings SPIE Image Processing and Interchange: Implementation and Systems*, 1659, pp. 117–129.