# Performance Analysis and Tuning for a Single-Chip Multiprocessor DSP

Jihong Kim
*Texas Instruments*
Yongmin Kim
*University of Washington*

*To achieve maximum performance, single-chip multiprocessor DSPs require a sophisticated performance-monitoring tool such as the MVP Performance Monitor. Using the MPM, programmers can easily and efficiently analyze and optimize DSP applications.*

Support for performance monitoring and tuning of complex parallel digital systems has long been an active research area. In recent years, the emphasis has been on evaluating the performance of large-scale parallel programs. For example, many performance tools have been developed for tuning parallel programs in shared-memory multiprocessors or distributed systems.[1–3] In this article, we discuss performance monitoring and tuning for much smaller parallel architectures: single-chip multiprocessor digital signal processors.

To meet the heavy computing requirements of emerging multimedia applications dealing with real-world data types such as video and voice, a new generation of high-performance programmable DSPs have been developed. These DSPs have a highly integrated parallel architecture, incorporating special-purpose hardware features, large on-chip memory, and multiple processors into a single chip. Figure 1 shows a generic architectural model of these DSPs. The on-chip memory, I/O controller, and processing elements connect through an interconnection network (for example, shared buses or a crossbar switch network). Because of the large data requirement in multimedia applications, data transfer by the I/O controller occurs in parallel with data processing by the PEs, thus improving the overall performance. Most high-performance DSPs, such as Analog Devices' ADSP-21060 Super Harvard Architecture (Sharc) DSP,[4] the Motorola DSP96002 Processor,[5] and the Texas Instruments TMS320C80 Multimedia Video Processor (MVP),[6] belong to this architectural family.

Developing an efficient DSP program for a single-chip multiprocessor


1063-6552/97/$10.00 © 1997 IEEE

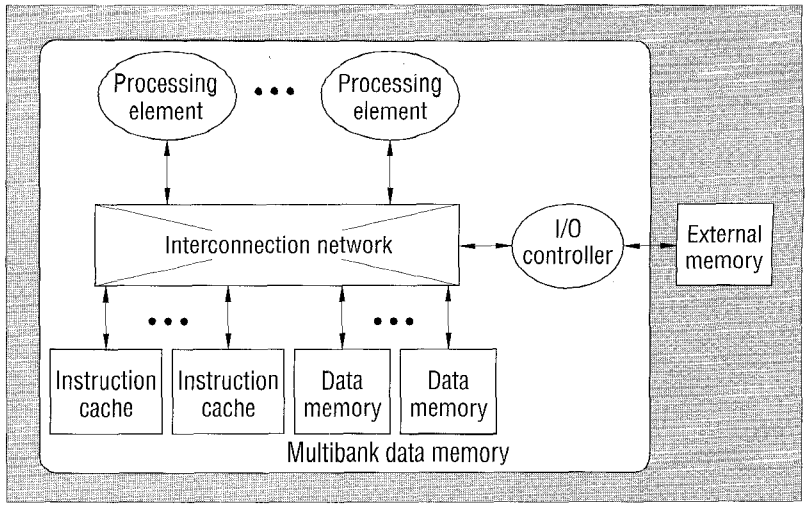

Figure 1. Single-chip multiprocessor DSP architecture.

DSP requires a good understanding of not only the algorithms and the DSP's intricacies but also the system-wide program behavior. Without system-wide performance understanding, a DSP program can suffer from various performance bottlenecks such as resource conflicts and unbalanced synchronization. This extra overhead could significantly degrade the program's overall performance. Unfortunately, this bottleneck is often difficult to predict and identify, even for experienced DSP programmers. Because DSPs are mainly used to achieve high performance, single-chip multiprocessor DSPs require performance-monitoring tools.

A performance-monitoring tool for DSPs must satisfy the DSP-specific requirements and support the performance parameters posed by DSPs. The *MVP Performance Monitor*, which works with TI's MVP, is such a tool. To demonstrate the MPM's effectiveness, we used it to analyze and optimize a 2D *discrete cosine transform* (DCT) implementation.

## Performance-monitoring tools for digital signal processors

In general, there are four performance-instrumentation levels: *hardware*, *system software*, *runtime system software*, and *application code*.[7] However, target application codes typically execute directly on top of the hardware in DSP-based systems, without much system software or runtime system support to minimize the overhead. So, more information from the hardware and application levels is necessary to further improve the performance of DSP applications.

Depending on the mechanism used to gather performance data, a performance-monitoring tool can be classified as a *hardware-based monitor* or *software-based monitor*.

A hardware-based monitor has separate performance-monitoring circuitry consisting of several hardware counters attached to an event-detection and activation logic module. Selection of the monitored events can be controlled by software. A performance counter is incremented whenever an event associated with a counter occurs. The hardware-based monitor is best suited for collecting hardware-performance data (for example, instruction-execution rate and cache-hit rate) in real time. Detailed hardware-performance information is useful in understanding the low-level behavior and interactions of programs running on a target system. Because separate hardware logic is dedicated to performance monitoring, the hardware-based monitor is unobtrusive (that is, the performance monitoring introduces no perturbation in the behavior of the monitored applications) and collects accurate performance data. However, gathering higher-level information is generally difficult, unless part of the monitor has been specifically designed to collect such information.

Performance monitoring is supported more often by software instrumentation. This approach modifies application programs to incorporate the instrumentation code, to collect performance data during program execution. The instrumentation code is manually inserted by modifying programs, or is automatically added by code-generation tools using their profiling features. Typically, software-based monitors measure primitive-level activities, such as process creation and destruction, message sending and receiving, procedure entry and exit, and blocking and unblocking by the scheduler.

Software instrumentation allows a more flexible and richer environment for performance monitoring than does the hardware-based approach. However, because the monitored application includes the instrumentation code, the performance-monitoring mechanism can perturb the application's behavior. This invasiveness of the data-collection mechanism reduces the accuracy of the collected performance data.

For DSP-based performance optimization, none of the hardware- or software-based approaches are adequate. In DSP applications, small code segments (for example, part of a single procedure) often dominate the overall performance. For these segments, software-instrumentation points are not easily defined, because not enough meaningful primitive-level activities exist in the segments. Furthermore, these segments are often written in assembly language, making it more difficult to pinpoint primitive-level activities for performance

monitoring. Because the performance of small code segments is being monitored in DSP applications, the software-instrumentation code added to these applications can significantly distort their runtime behavior, resulting in inaccurate performance data. On the other hand, a hardware-based monitor does not provide enough higher-level information, because of its limited monitoring scope.

To collect accurate hardware performance data, as well as higher-level information, without perturbing the system's behavior, a software-monitor approach based on a hardware-simulator model is more appropriate. If an accurate hardware-simulator model is available, this approach supports both detailed hardware data collection and higher-level analysis without introducing any significant artifact into measurements. Developing an accurate hardware simulator usually requires significant effort. In DSP-based systems, however, this is not an extra burden, because an accurate DSP simulator model is typically available from the DSP manufacturer. For example, most low-level DSP programs are developed using a simulator for a specific DSP. Thus, a performance-monitoring tool for this DSP can be developed by extending the existing DSP simulator.

## Requirements

A performance-monitoring tool for developing DSP applications should have at least four features.

First, the tool should be seamlessly integrated with debugging tools.[8] Debugging and performance monitoring should be unified: a continuum exists between debugging for correct functionality and debugging to achieve the desired performance objectives. Integration is especially important in DSP-based systems where the debugging tools play a central role in developing application programs. Mode switching between function debugging and performance monitoring should be simple and easy, so that these integrated tools are usable.

Second, the tool should present a uniform and familiar user interface throughout all its components, to relieve the users from learning many different interfaces. For example, the user should be able to use the same commands to control the program's flow in both the function-debugging mode and the performance-debugging mode.

Third, the tool should produce useful results at a reasonable cost. In DSP applications, a multiple number of small code segments such as tight loops are often the candidates for in-depth performance monitoring and analysis. If performance monitoring takes an excessive amount of time to produce measurements and analysis results, its usefulness is significantly reduced.

Fourth, the user should be able to extend the tool. For example, for a single-chip multiprocessor, predicting all the possible combinations of events is impossible. The user should be able to add new types of events to be monitored.

## Performance-monitoring parameters

The target DSPs we've described have three main performance factors.

The first is the balance between I/O time (by the I/O controller) and compute time (by the PEs). Because our target DSPs can perform data processing and data movement concurrently, analyzing data processing and data movement requirements for a given function is very important. To achieve an optimal program implementation, we must know whether a specific implementation of an individual subtask (of a program) is I/O-bound or compute-bound. We must also know each subtask's degree of I/O-boundness or compute-boundness. For example, if we find that one subtask is I/O-bound, its degree of I/O-boundness can guide us in implementing other subtasks, as we try to balance I/O time and compute time for the overall program.

The second factor is on-chip instruction-cache behavior. Because one I/O controller serves both instruction-cache misses and data-movement requests from PEs in target DSPs, instruction-cache misses affect not only a specific function's compute time but also its I/O time. Cache misses increase the overall I/O time of an I/O-bound implementation because they interrupt the I/O controller's data-transfer services. Thus, cache misses directly affect the overall program's execution time. Because small code segments typically dominate DSP programs and the cache-miss service time varies depending on the I/O controller's workload, a simple count of the total cache misses would not provide enough information to understand and improve the program's cache behavior. More detailed information such as source address, frequency, and service time for each cache miss is necessary.

The third factor is interconnection-network contentions among PEs and the I/O controller. Contentions among PEs would increase the total compute time, while contentions between PEs and the I/O controller would increase either the compute time or the I/O time, depending on the interconnection-network priorities of the PEs' and the I/O controller's accesses. For interconnection-network contentions, the total number of

contentions for each PE and for the I/O controller provides enough information to improve DSP programs. This information is sufficient because the interconnection-network priorities for PEs and the I/O controller tend to remain relatively constant and can be modified based on the total number of contentions. If these priorities must be changed more dynamically, detailed information for each contention, such as contending PEs, would be necessary.

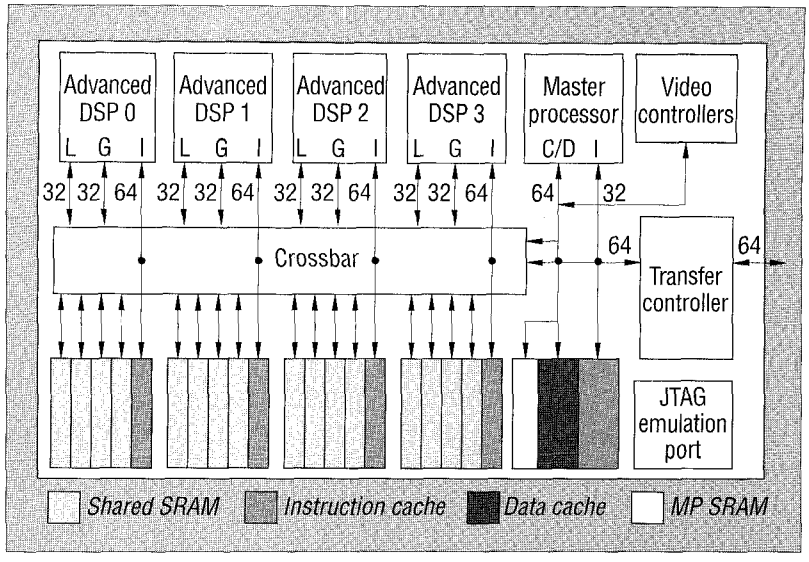


Figure 2. High-level block diagram of the TMS320C80 Multimedia Video Processor.

## *The MVP Performance Monitor*

The MPM, based on a cycle-accurate DSP simulator, satisfies the requirements of DSP-based monitoring tools and supports the three performance parameters we've just considered. However, before we discuss the MPM, let's briefly look at the MVP, the DSP for which we designed it.

### The TMS320C80 MVP

The MVP is a single-chip, heterogeneous, MIMD multiprocessor connected by a crossbar to multiple on-chip shared-memory modules.[6,9,10] It combines a RISC processor, four parallel processors, an intelligent direct-memory-access controller, and two video controllers. It can process more than 2 billion operations per second, with on-chip data transfer of 2.4 Gbytes per second. To reduce the data-transfer overhead from external memory and devices, the MVP has a large on-chip memory (25 2-Kbyte modules).

Figure 2 shows a high-level diagram of the MVP's major functional blocks. The *master processor* 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 MVP. Also, the MP is the preferred processor for performing floating-point operations. The four parallel processors—*advanced DSPs*—have a highly parallel architecture optimized for multimedia, video and image compression, image and signal processing, and computer graphics. Each ADSP can perform up to 15 RISC-equivalent operations in a single clock cycle via a long-instruction-word (64 bits) mechanism. It also has many powerful features not available in conventional DSPs. For example, each ADSP has a three-operand, 32-bit arithmetic and logical unit (ALU), which can be optionally split into two 16-bit or four 8-bit units. The *video controllers* (VCs) support programmable video timing to control both capture and display. The high-performance *crossbar switch network* fully interconnects the processors and on-chip shared-memory modules.

While five processors (the MP and four ADSPs) provide the computing power, the *transfer controller*, 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 multimedia applications 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. Because the processors and the TC can operate in parallel, most data transfers performed by the TC are hidden to the processors in the optimized implementation. While a processor works on the current block of data in the shared memory, the TC services a request for the next block in parallel.

The TC's main data-transfer mechanism is a *packet transfer*, a transfer of data blocks between two areas of the MVP memory. The MP, ADSPs, VC, or external devices initiate packet transfers by sending requests to the TC, using software or hardware. Once a processor has submitted a transfer request, it can continue program execution without waiting for the transfer's completion. Many different modes of packet transfers are available, such as multidimensional transfers, table-guided transfers, fill-with-value, and serial register transfers.[10]

### MPM architectural overview

Figure 3 shows the MPM's architecture. The MPM is tightly integrated with the Texas Instruments MVP

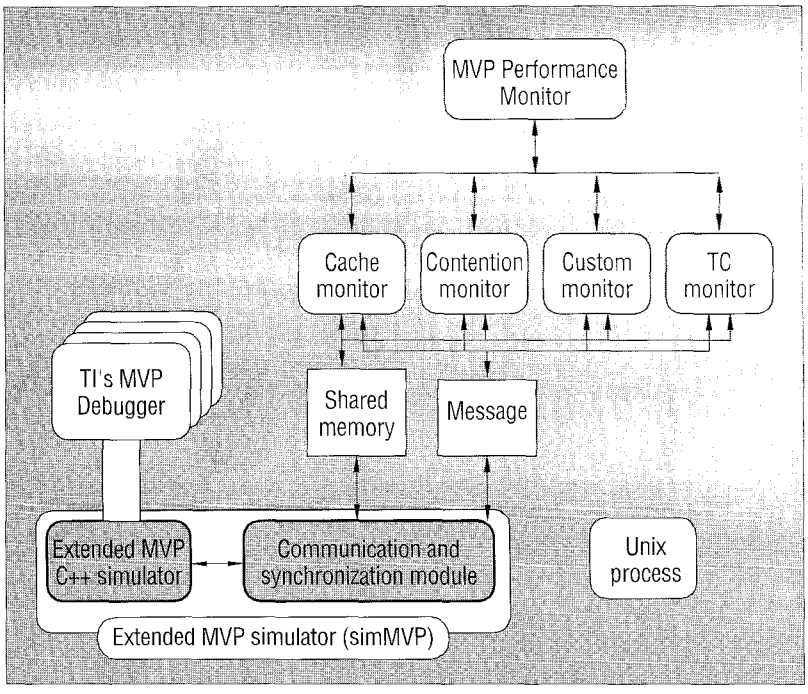

Figure 3. The MVP Performance Monitor's software architecture.

Debugger tool, which is widely used in developing MVP programs. The user can interactively switch between the performance-monitoring mode, using the MPM, and the function-debugging mode, using the MVP Debugger. The performance-monitoring mode is approximately three times slower than the function-debugging mode (running at a speed of 700 to 1,000 instructions per second on a Sun Sparcstation 20). This speed is adequate for monitoring the performance of multiple code segments switching interactively between the two modes.

The MPM's core is the extended MVP simulator, which consists of

- *the MVP C++ simulator*, which accurately models the MVP up to a half-cycle resolution (when the MVP runs at 50 MHz, a full clock cycle is 20 ns, so a half cycle is 10 ns);
- *customized MPM extensions* to the MVP C++ simulator, which include three types of monitoring support and TC debugging capability; and
- *a communication and synchronization module*, which communicates with the MPM user interface.

The MPM user interface spawns a child process that monitors the user-specified event.

### Performance monitoring

The MPM supports the performance parameters with *cache monitoring*, *contention monitoring*, and *custom monitoring*. Figure 4 shows a snapshot of the MPM environment. The whole simulation starts from the MPM user interface (the upper-left window). Once the MPM starts, the MVP Debugger starts simulating the MVP program (the lower-right window). The user typically sets a breakpoint at the start address of a code segment $S$ that will be monitored by the MPM. If the simulation stops 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 (the upper-right window). In Figure 4, cache monitoring is selected. The user then sets another breakpoint at the end address of $S$ and continues the simulation. When the simulation stops, the user can examine the monitoring result on a separate (the lower-left) window. For a typical DSP tight loop (consisting of approximately 10 to 30 instructions), a data-collection session takes from a few dozen seconds to a few minutes.

For cache monitoring, the MPM displays (in the lower-left window in Figure 4) the collected information on cache misses for the code segment $S$, including the total number of cache misses, the total cache-miss service time, the total number of noncompulsory cache misses, and the total cache-miss service time for noncompulsory cache misses. The MPM also displays the summary of all the cache misses in a table that lists the source and destination addresses, average service time, and frequency of each cache miss. Based on this information, the user can restructure the MVP program or adjust the program size to reduce the number of noncompulsory cache misses. Contention monitoring works similarly and displays the total number of crossbar switch contentions for each processor.

Custom monitoring is used to observe a user-defined event—unlike cache and contention monitoring, where the MPM predetermines monitoring events. The MPM specifies a user-defined event by ADSP checkpoints that are the addresses of selected ADSP instructions. Custom monitoring records the execution of ADSP checkpoints. The MPM displays the result graphically (see Figure 5). The $x$-axis of this graph indicates the MVP clock cycle numbers. The lines in the upper row display the status of data movements (that is, packet transfers) requested by a specific ADSP. The thick line indicates when the packet transfer service is delayed because the TC is busy servicing higher-priority requests. The thin line indicates when the TC services the requested packet transfer. The line in the lower row shows when the spec-

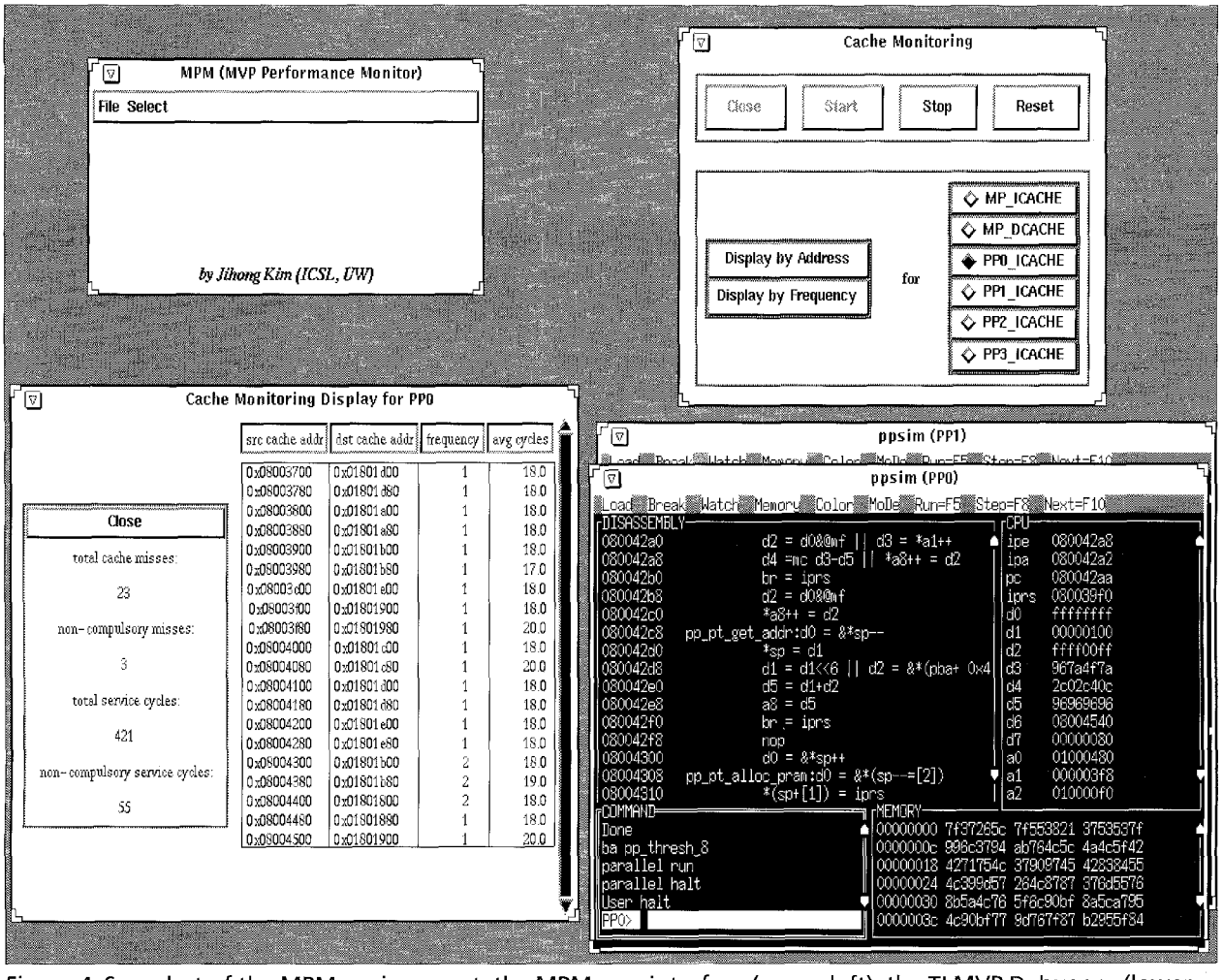


Figure 4. Snapshot of the MPM environment: the MPM user interface (upper left); the TI MVP Debugger (lower right); the cache-monitoring user interface (upper right); the cache-monitoring result (lower left).

ified ADSP checkpoint executes. (If an ADSP instruction at the checkpoint executes only once, the graph displays a point instead of a line.) The user can measure each interval by clicking the mouse button.

One main use of custom monitoring is to evaluate whether an implemented MVP program is compute-bound or I/O-bound. In the MVP, ADSPs (or the MP) submit data-transfer requests to the I/O controller (the TC) and check for the data-transfer completion by polling a predetermined register. Therefore, by custom-monitoring the ADSP polling instruction for the packet-transfer completion, we can determine if the requested data transfer has completed. For example, if an ADSP checkpoint was set to the ADSP polling instruction for the packet-transfer completion, Figure 5 would show that this polling instruction executed for a large number

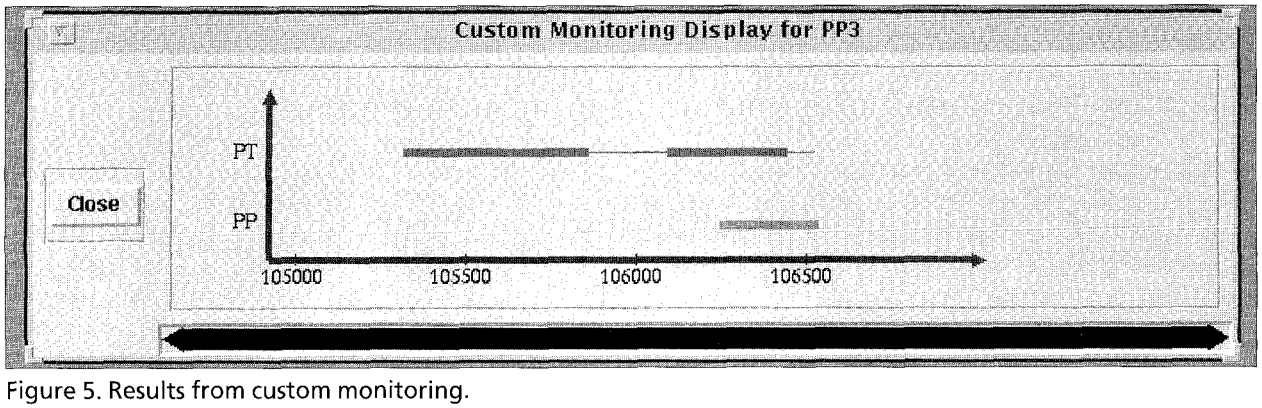


Figure 5. Results from custom monitoring.

Table 1. Four possible cases of an algorithm $s$ and its implementation $\Omega_s$.

| | Theoretical analysis | Monitoring result |
|---|---|---|
| Case I | I/O-bound | I/O-bound |
| Case II | I/O-bound | Compute-bound |
| Case III | Compute-bound | I/O-bound |
| Case IV | Compute-bound | Compute-bound |

of cycles (a solid line in the lower row). This would indicate that ADSP 3 was wasting a number of cycles just waiting for the requested packet transfer to complete.

### Performance-tuning strategies

An algorithm implementation on the ADSP is either compute-bound or I/O-bound, depending on whether the processing time or transfer time is dominant. Therefore, the strategy for improving an MVP program's performance is different for different cases.

Table 1 summarizes four cases of an algorithm $s$ and its ADSP implementation $\Omega_s$. We can classify the algorithm $s$ as compute-bound or I/O-bound based on pure theoretical analysis, using the pure processing time $t_{\text{pure processing}}$ and pure data-transfer time $t_{\text{pure i/o}}$. To compute the real data-transfer time for the MVP, we multiply $t_{\text{pure i/o}}$ by the number of ADSPs used in the algorithm implementation, because the single I/O controller needs to serve data-transfer requests from multiple ADSPs. Once the algorithm $s$ is realized into a specific implementation $\Omega_s$, we classify the implementation as compute-bound or I/O-bound, based on the monitoring results. We make this classification by custom monitoring with an ADSP checkpoint set to the address of the ADSP polling instruction that checks for the packet-transfer completion.

In all four cases, the first step should be checking whether significant cache-miss overhead exists, because the cache misses affect both the processing time and transfer time. Once the noncompulsory cache misses have been handled, more case-specific optimization can start.

If the implementation is I/O-bound, as expected from the theoretical analysis (Case I), further optimization might be achieved by improving the program's dataflow portion so that the data transfers are more efficient. Crossbar-contention monitoring might also be necessary, because the transfer time can increase significantly if the TC stalls because of a large number of crossbar contentions between the TC and ADSPs. If multiple routines are to be integrated, combining an I/O-bound routine with other compute-bound routines might improve the overall execution time if the dataflow pattern of these routines is compatible and the combination does not introduce additional overhead.

When the implementation is compute-bound as analyzed theoretically (Case IV), the optimization options are limited to the improvement of the tight loops. Further parallelization or more efficient use of performance-enhancing features could shorten the execution time. The performance can also be improved by reducing the number of crossbar contentions with the other ADSPs and the TC if these contentions stall the ADSP.

When the I/O part of the implementation performs less efficiently than expected (Case III), a close examination of the current dataflow design is required. The program's dataflow portion might need redesigning. Reducing the number of TC stalls caused by crossbar contentions with the other ADSPs can also improve performance in this case. Combining some necessary computing steps from other routines with the tight loops, if possible, could decrease the effect of slower I/O performance, thereby shortening the overall execution time.

If the computing part of the implementation performs less efficiently than analyzed (Case II), examining the level of optimization in the tight loops is necessary. Further monitoring of the crossbar contentions with the other ADSPs and TC is also necessary, assuming the noncompulsory cache misses have been managed already.

## A performance-tuning example

We used the MPM to improve the performance of an 8×8 block-based 2D DCT on the MediaStation 5000, an MVP-based multimedia system.[11] (We have previously reported on a more complex example.[12]) The program first divides an $N \times M$ 8-bit input image into many ($N/8 \times M/8$) nonoverlapping 8×8 blocks. Then, the program performs a 2D DCT on each 8×8 block, via row-wise 8-point DCTs followed by column-wise 8-point DCTs. The output block has the same spatial resolution (8×8) as the input block. With the 8 bits-per-pixel input grayscale image, the 2D DCT output coefficients range from -2,048 to +2,047, because of the repeated multiplications (with the cosine values) and accumulations. Thus, the DCT coefficients are stored as 16-bit fixed-point numbers. Out of 16 bits, the four least significant bits represent the fractional part, while the upper twelve bits represent the integer part.

The 2D DCT program consists of two tasks: 8-bit to 16-bit conversion (`convert`) and 16-bit 8×8 block-based 2D DCT (`dct`). The separate `convert` task is necessary to prepare the input data properly for the `dct` task. Using 8-bit input pixels in `dct` to produce the 16-bit fixed-point DCT coefficients requires an extra shifting between a 16-bit cosine value and an 8-bit input pixel after every multiplication. The extra shifting changes the multiplication result back to a 16-bit number. Addi-

tionally, to perform two 16-bit arithmetic operations simultaneously by splitting the ALU, the program must pack two 16-bit multiplication results into a 32-bit word. With the preconverted 16-bit input pixels, however, the extra shifting and packing are unnecessary because the program uses the hardware swapper of the ADSP's multiplier unit, which cannot be used with 8-bit pixels. Of the various fast DCT algorithms, we used Lee's algorithm in `dct`.[13]

We highly optimized two processing cores for `convert` and `dct`, heavily using many advanced features of the ADSP. The `convert` processing core takes 1.25 cycles per output pixel, while the `dct` processing core takes 352 cycles per 8×8 block, or 5.5 cycles per output pixel. Four ADSPs running in parallel at 50 MHz, with each ADSP processing a quarter of the image, would take a total of 8.84 ms to perform the 8×8 2D DCT for a 512×512 input image. `Convert` would take 1.63 ms, while `dct` would take 7.21 ms. However, these estimates are pure processing times and do not include any overhead.

Using the two processing cores as building blocks, we implemented the first version of the integrated program by combining the two cores in a single ADSP-level function (see Figure 6). One 8×8 8-bit input block is brought into the on-chip memory at a time, and one 8×8 16-bit DCT coefficient block is written out to the external memory. The `convert` processing core and `dct` processing core share the same data flow. This version performed surprisingly poorly. It took 44.5 ms, 35.66 ms longer than the theoretical estimate of 8.84 ms.

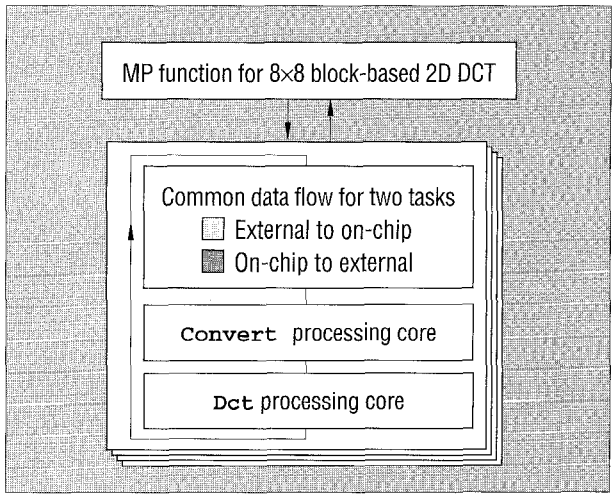


Figure 6. The structure of the 2D discrete cosine transform program, first version.

## Instruction-cache monitoring

Through cache monitoring in the two processing cores, we identified that each iteration (each 8×8 block processing) used approximately 1,100 to 1,200 extra cycles for servicing instruction-cache misses. Figure 7 illustrates this overhead for ADSP 2 with the four ADSPs operating in parallel. Figure 7a shows the cache-monitoring result for ADSP 2 after processing the third 8×8 block, while Figure 7b shows the result after the fourth 8×8 block. Between the processing of these two blocks, the total cache-service cycles increased by 1,208. We did not expect this large number of cache-miss service cycles, because the `convert` and `dct` processing cores use 75 ADSP assembly-language instructions (20 for `convert` and 55 for `dct`). This number is considerably lower than the maximum number (256) of instructions that can fit into the ADSP's 2-Kbyte instruction cache. (The 2-Kbyte instruction cache is divided into four blocks, each of which can store 64 64-bit instructions. Therefore, the instruction cache can store four different code segments simultaneously.)

The large number of cache misses happened between the successive subroutine calls to the `convert` and `dct` processing core routines. The MVP linker was putting these routines in the instruction cache without any

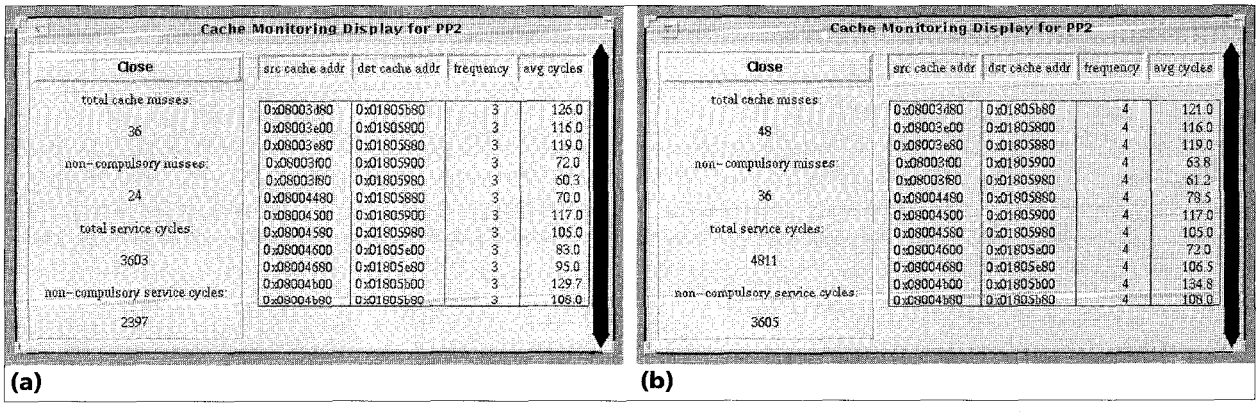


Figure 7. Cache-monitoring results for ADSP 2: (a) after processing the third 8x8 block; (b) after processing the fourth 8x8 block.

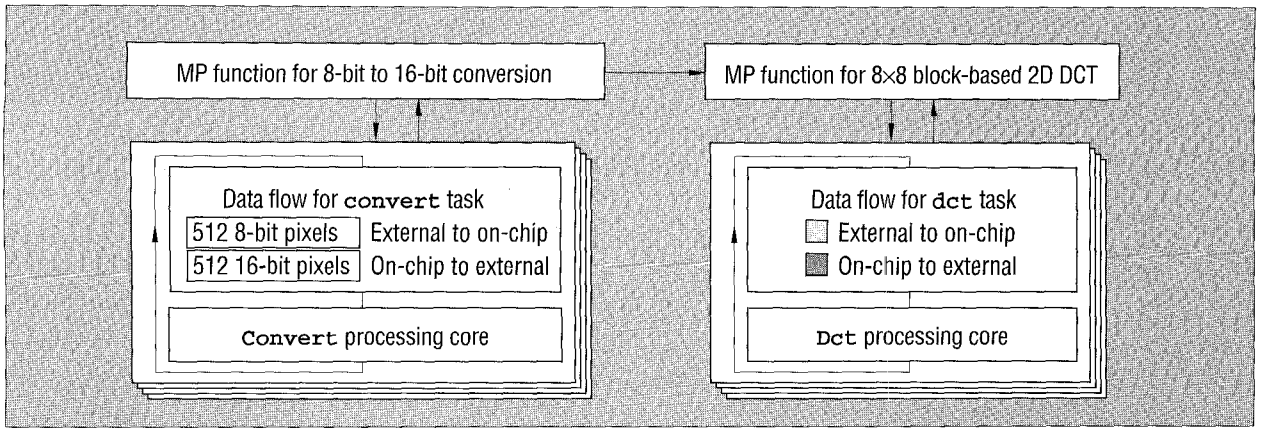


Figure 8. The structure of the 2D DCT program's second version.

64-instruction block consideration—for example, starting a routine in the beginning of a cache block. Furthermore, the linker did not place these two routines close to each other when building an executable module. So each core routine required two 64-instruction cache blocks. Four 64-instruction blocks for two core routines and two additional 64-instruction blocks for the ADSP-level C code (that is, **for** statement blocks) caused cache misses for each subroutine call. The overhead from these noncompulsory cache misses caused the performance loss of between 22.5 and 24.6 ms.

To reduce the number of cache misses between **convert** and **dct**, the program's second version placed the two processing cores into separate ADSP-level functions (see Figure 8). The program stored the converted image in the MS5000's external memory before the second function performed the DCT on the image. For more efficient data movement, the **convert** task for each ADSP brings 512 8-bit pixels (or four rows of 128 pixels) into the on-chip memory at a time, instead of a single 8×8 8-bit input block. For this organization, cache monitoring showed that using two ADSP-level func-

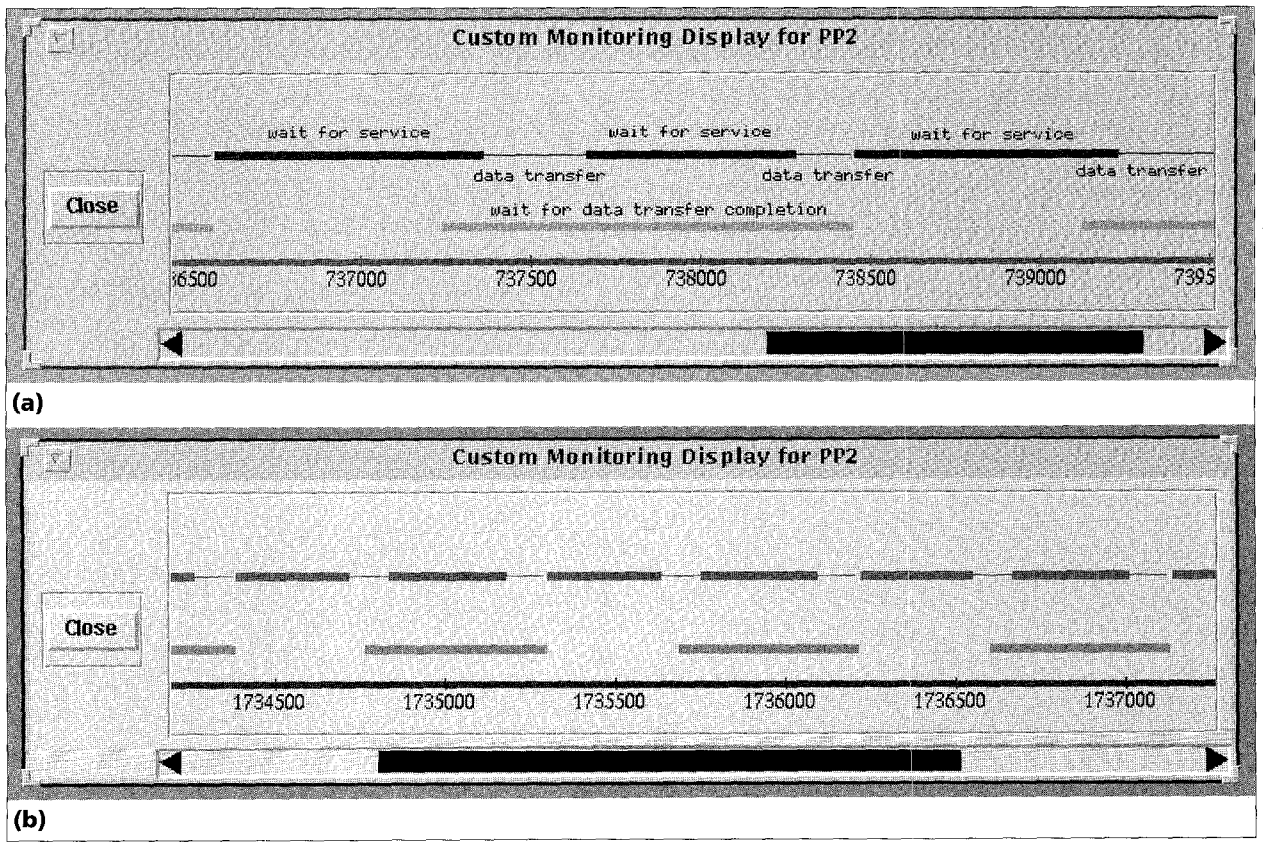


Figure 9. Custom-monitoring results for ADSP 2: (a) for the 8-bit to 16-bit conversion ADSP-level function; (b) for the 16-bit 8x8 2D DCT ADSP-level function.

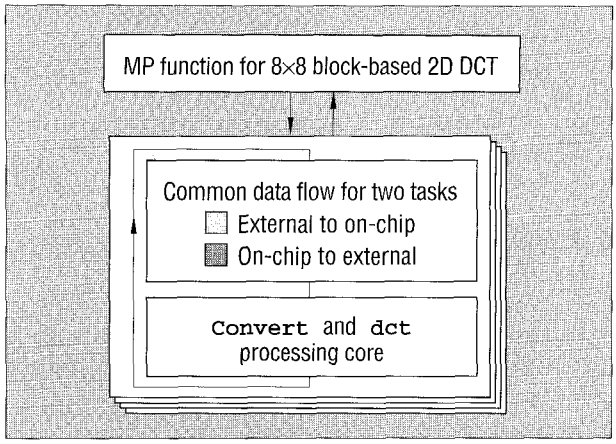


Figure 10. The structure of the 2D DCT program's third version.

tions caused no extra cache misses. The execution time decreased from 44.5 ms to 16.3 ms, an improvement of 28.2 ms. The improvement was about 4 to 6 ms larger than you might expect from a reduction of cache misses. This is because unnecessary cache misses affect the transfer time as well as the processing time, and the significant reduction in cache misses also improves the overall I/O performance.

### I/O-BOUNDNESS MONITORING

However, both ADSP-level functions became I/O-bound, as the custom-monitoring results for ADSP 2 show (see Figure 9). As we mentioned before, we set an ADSP checkpoint to be the address of the ADSP polling instruction that checks for the packet-transfer completion. The lines in the lower row in Figure 9 indicate that ADSP 2 executed this polling instruction for a large number of cycles before the requested packet transfer completed. The long thick lines in the upper row show that the TC's unavailability delayed the packet-transfer service for many cycles. The thin lines indicate when the TC serviced the requested packet transfer. Because of `convert`'s simple computation steps in the processing core, it spent about two thirds of its computing cycles waiting for the packet-transfer completion (see Figure 9a), while `dct` spent more than 50% of its cycles waiting for the packet-transfer completion (see Figure 9b).

To reduce the overall I/O time, the program's third version combined the two processing cores into a single core (see Figure 10). With the same data flow, the decreased I/O time reduced the execution time to 11.6 ms. The new combined processing core did not have any noncompulsory cache misses. However, custom monitoring indicated that the implementation was still I/O-bound. Figure 11 illustrates this I/O-boundness for ADSP 2, although combining the two I/O-bound routines into a single routine achieved a better balance between the I/O time and the processing time. The total length of the lines in the lower row is much shorter in Figure 11 than in Figure 9, meaning that ADSP 2 spent much less time checking for the packet-transfer completion. However, the ideal 2D DCT implementation would not have any lines in the lower row.

Because the third version was still I/O-bound, we had to analyze the data flow in depth. We did not expect this version to be I/O-bound, because the combined processing core was supposed to take only 6.75 cycles per output pixel, while the number of pure I/O cycles was only 48 cycles/64 pixels, or 0.75 cycles per output pixel. (Using two cycles per memory access and 64-bit data width, reading in 64 8-bit data—64 bytes—takes 16 cycles, and writing back 64 16-bit data takes another 32 cycles, a total of 128 bytes.) With four ADSPs submitting data-transfer requests at the same time, the effective data-transfer rate for the MVP is four times larger than 0.75 cycles per output pixel. On close examination, we found that the row-time access overhead contributed greatly to the

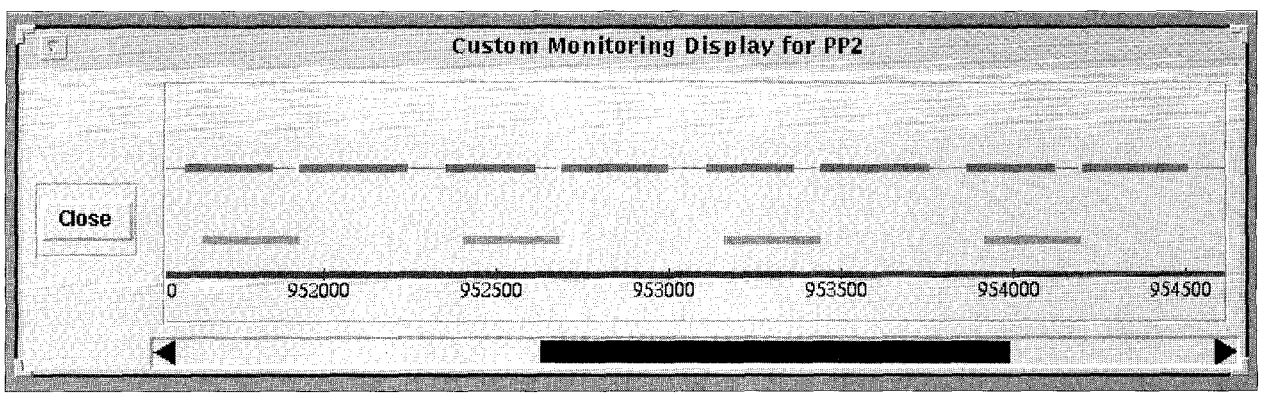


Figure 11. Custom-monitoring result for ADSP 2 for the ADSP-level function with the two processing cores combined into one.

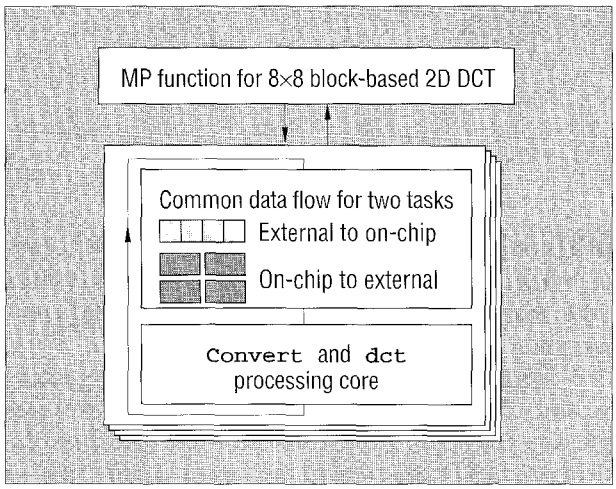


Figure 12. The structure of the 2D DCT program's fourth version.

Table 2. The performance of the four versions of the 8×8 2D DCT program on a 512×512 input image.

| Program organization | Performance (ms) | Speedup over the first version |
|---|---|---|
| First version | 44.5 | — |
| Two ADSP-level functions | 16.3 | 2.73 |
| Combined processing core | 11.6 | 3.84 |
| Four 8×8 blocks at a time | 9.53 | 4.67 |

large I/O time because accessing an 8×8 block requires frequent memory page boundary crossings.

(The MS5000's main memory subsystem supports 2-Kbyte pages. With 512×512 images, two row-time accesses are necessary to read an input 8×8 block [8 bits per pixel], and four row-time accesses are necessary to write back the 8×8 results [16 bits per pixel]. With two cycles per memory access and 64-bit data width, pure data transfer would take 48 cycles. Approximately 90 extra cycles are necessary for six row-time accesses. So, it takes 2.16 cycles per output pixel on each ADSP. With four ADSPs running in parallel, the total data-transfer rate for the MVP becomes 8.63 cycles per output pixel, which is greater than the combined processing core's 6.75 cycles per output pixel.)

To reduce the row-time access overhead, the program's fourth version brought in and processed four 8×8 blocks at a time, instead of one. Figure 12 shows the overall program structure for this version, which became compute-bound (see Figure 13). An ADSP checkpoint executed only once (indicated in Figure 13 by a single point in the middle of the lower row). This execution occurred well after the requested packet transfer completed (indicated by the end of the thin line in the upper row). This version executed in 9.53 ms.

Table 2 summarizes the four versions of the 2D DCT program. We achieved the overall speedup of 4.67 by tuning the performance based on the monitoring results from the MPM. The final version's execution time of 9.53 ms is slightly larger than the theoretical minimum of 8.84 ms. However, further performance improvement would be much more difficult, and the achievable performance gain would be small compared to the necessary effort.

Achieving good performance on high-performance single-chip multiprocessor DSPs is challenging. There is almost no end to optimizing any complex algorithm. However, tools such as the MPM can simplify this task.

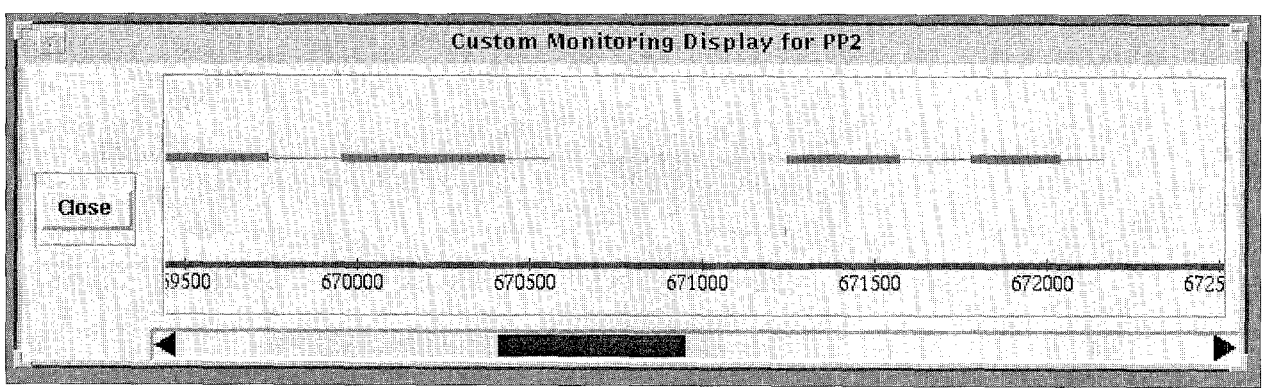


Figure 13. Custom-monitoring result for ADSP 2 for the ADSP-level function, combining two processing cores into one and processing four 8x8 blocks at a time.

With the MPM, MVP programmers identify the DSP-specific performance bottlenecks. It presents the monitored results so that the programmer gains a clear view of program execution and areas of potential improvement. The tight integration between the familiar functional debugger and performance monitor allows the MPM to easily and efficiently fine-tune DSP applications. So, as our example shows, with judicious use of the MPM and experience, intuition, and reasonable effort, programmers can considerably improve and optimize the performance of image-computing algorithms.

We plan to add more functions to the MPM. For example, in the current version, only one type of user-defined event can be specified for a single monitoring session. The MPM can be extended to support a multiple number of user-defined events simultaneously within one monitoring session.

## References


1. T. Anderson and E. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance," *Performance Evaluation Review*, Vol. 18, No. 1, May 1990, pp. 115–125.

2. D. Reed et al., "Scalable Performance Analysis: The Pablo Performance Analysis Environment," *Proc. 1993 Scalable Parallel Libraries Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 104–113.

3. B. Miller et al., "The Paradyn Parallel Performance Measurement Tool," *Computer*, Vol. 28, No. 11, Nov. 1995, pp. 37–46.

4. J. Leonard, "EDN's 1994 DSP-Chip Directory," *EDN*, Vol. 39, No. 12, Dec. 1994, pp. 75–135.

5. M. El-Sharkawy, *Signal Processing, Image Processing and Graphics Applications with Motorola's DSP96002 Processor*, PTR Prentice-Hall, Upper Saddle River, N.J., 1994.

6. K. Guttag, R. Gove, and J. Van Aken, "A Single-Chip Multiprocessor for Multimedia: The MVP," *IEEE Computer Graphics and Applications*, Vol. 12, No. 6, Nov. 1992, pp. 53–64.

7. D. Reed, "Performance Instrumentation Techniques for Parallel Systems," in *Performance Evaluation of Computer and Communications Systems*, L. Donatiello and R. Nelson, eds., *Lecture Notes in Computer Science 729*, Springer-Verlag, Berlin, 1993, pp. 463–490.

8. D. Krummc and A. Couch, "Integrated Debugging and Performance Monitoring for Parallel Programs," *Proc. 15th Ann. Int'l Computer Software and Applications Conf.*, IEEE CS Press, 1991, pp. 317–318.

9. R. Gove, "The MVP: A Highly-Integrated Video Compression Chip," *Proc. Fourth IEEE Data Compression Conf.*, IEEE CS Press, 1994, pp. 215–224.

10. *TMS320C8x (MVP) Online Reference* (Release 1.00), Texas Instruments, Dallas, 1994.

11. W. Lee et al., "MediaStation 5000: Integrating Video and Audio," *IEEE MultiMedia*, Vol. 1, No. 2, June 1994, pp. 50–61.

12. J. Kim and Y. Kim, "UWICL: A Multi-Layered Parallel Image Computing Library for Single-Chip Multiprocessor-Based Time-Critical Systems," *Real-Time Imaging*, Vol. 2, No. 3, June 1996, pp. 187–199.

13. B. Lee, "A New Algorithm to Compute the Discrete Cosine Transform," *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol. ASSP-32, No. 6, Dec. 1984, pp. 1243–1245.



**Jihong Kim** is a member of the technical staff at Texas Instruments' Corporate Research Laboratories. His research interests include image computing, multimedia systems, algorithm design and software tools, performance analysis, real-time systems, and simulation. He received his BS in computer science and statistics from Seoul National University in 1986, and his MS and PhD in computer science and engineering from the University of Washington in 1988 and 1995. He is a member of the IEEE Computer Society and the ACM. Contact him at Texas Instruments, PO Box 655303, M/S 8374, Dallas, TX 75265; jihong@csc.ti.com.

**Yongmin Kim** is a professor of electrical engineering and an adjunct professor of bioengineering, radiology, and computer science and engineering at the University of Washington. His research interests are multimedia, image processing, computer graphics, medical imaging, machine vision, and computer architecture. He also directs the Image Computing Systems Laboratory and the University of Washington Image Computing Library Consortium. He received his BS in electronics engineering from Seoul National University in 1975 and his MS and PhD in electrical engineering from the University of Wisconsin in 1979 and 1982. He received the Early Career Achievement Award from the IEEE/EMBS in 1988. He is a fellow of the IEEE and of the American Institute for Medical and Biological Engineering. He also serves as an Accreditation Board of Engineering and Technology evaluator for computer engineering. Contact him at the Dept. of Electrical Engineering, Box 352500, Univ. of Washington, Seattle, WA 98195-2500; kim@ee.washington.edu.