IPSJ Transactions on System LSI Design Methodology Vol. 3 47–56 (Feb. 2010)

Regular Paper

Performance Evaluation of a Dynamically Switchable SIMD/MIMD Processor by Using an Image Recognition Application

Shohei Nomoto,[†1] Shorin Kyo[†1]
and Shinichiro Okazaki[†1]

We have developed an "XC core" processor that achieves low cost, high performance, and low power consumption through the use of a highly parallel SIMD architecture (the SIMD mode), as well as achieves high flexibility by morphing into a MIMD architecture (MIMD mode). In this paper, we evaluate the effectiveness of the MIMD mode by using a white line detection algorithm for open roads. Our evaluation shows that the algorithm can be processed in real time (less than 33 ms) by using the MIMD mode to execute verification of white line segments, which is a part of the algorithm not suitable to be executed by the SIMD mode. We also show that the verification can be executed five times faster by using region of interest (ROI) transfer instructions to efficiently transfer the ROI of an image. Furthermore, we also measured the execution time in the MIMD mode with changing the number of processing units (PUs) used, from 2 to 4, 8, 16 and 32. The measured results show that the performance improvement rate slows down when using more than 16 PUs in the MIMD mode, mainly due to insufficient parallelism in the verification process. Overall, a 10.68 times speedup was achieved by using 32 PUs in the MIMD mode, compared with only using the SIMD mode.

1. Introduction

Vehicle safety systems based on image recognition technology are becoming widespread in recent years[1),2)]. These systems must be a sufficiently high performance to achieve real-time image recognition, and low power consumption to satisfy the stringent thermal design requirements for vehicle environments. They must also be highly programmable to support diverse image recognition algorithms, thus achieving faster time to market and higher maintainability.

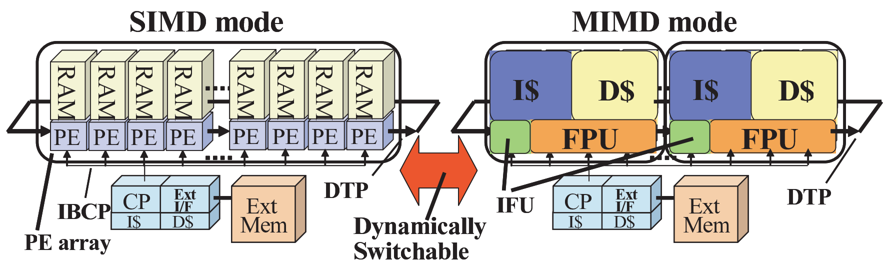
†1 System IP Core Research Laboratories, NEC Corporation

Various image recognition processors have been developed to satisfy these requirements. The Vchip (Vision/Video Chip), for example, implements basic operations for image recognition (such as an edge filter) as dedicated hardware[3)]. By using the dedicated hardware, the Vchip can cost-effectively achieve high performance and low power consumption for pre-defined operations which are supported by the hardware, but it lacks flexibility to support diverse algorithms. Visconti (vision-based sensing, control, and intelligence) consists of three 3-way VLIW-type image recognition cores, with each core being able to simultaneously execute one scalar instruction and two SIMD (8-bit, 8-way) instructions at each clock cycle[4)]. Visconti can extract and utilize various granularities of parallelism by using Multicore, VLIW, and SIMD instructions. Visconti can therefore flexibly and effectively execute various types of image recognition algorithms, but the peak performance of Visconti will be much lower than that of dedicated hardware that uses the same amount of hardware resources.

The authors have developed a series of image recognition processors called IMAP (Integrated Memory Array Processor)[5)]. IMAP processors employ a highly parallel SIMD architecture and consist of a linear array of processing elements (PEs), each of which is tightly coupled with a memory. Because all the PEs execute the same instruction, the size of the circuit required for control can be reduced and the size of the circuit used for operations (the number of PEs) can be increased. However, as image recognition algorithms have become more diversified and complicated in recent years, it is becoming more difficult to map them onto a pure SIMD architecture.

To solve these issues, we have designed a processor architecture called the XC core, which can dynamically switch between the SIMD and MIMD modes[6)]. In addition to the conventional SIMD mode, the XC core also implements a MIMD mode in which four PEs are reconfigured as one operation unit (PU: Processing Unit), as shown in **Fig. 1**. Because each PU of the MIMD mode can work independently, the XC core can flexibly and effectively execute algorithms consisting of multiple independent tasks.

In this paper, the effectiveness of the XC core's MIMD mode is shown by evaluating a white line detection algorithm for open roads implemented in the XC core. The remainder of the paper is structured as follows. Section 2 explains the



*PE = Processing Element, CP = Control Processor
IBCP = Instruction Broad Cast Path, DTP = Data Transfer Path
IFU = Instruction Fetch Unit, I$ = Instruction Cache, D$ = Data Cache
FPU = Floating Point Unit, Ext I/F = External Memory Interface

Fig. 1 Overview of the XC core architecture.

architecture of the XC core, and the region of interest (ROI) transfer instructions implemented to efficiently transfer the ROI of an image. Section 3 describes the detailed operations of the white line detection algorithm for open roads. In Section 4, the algorithm is implemented in the SIMD and MIMD modes, and the execution time in the SIMD mode is compared with that in the MIMD mode. The performance characteristics of the MIMD mode are also evaluated based on the number of PUs used to execute the algorithm. Finally, Section 5 presents the conclusion of this paper.

2. Overview of the XC Core Architecture

This section gives an overview of the XC core architecture. Firstly the basic structure of the XC core and an implementation that enables dynamic switching between the SIMD and MIMD mode are described. Secondly, the region of interest (ROI) transfer instructions that are implemented in each PU, and that can transfer the ROI of an image efficiently are explained. Finally, the software development environment for the XC core is described.

2.1 Basic Structures of the XC Core

As shown in Fig. 1, the XC core consists of a Control Processor (CP), and a PE array that rings a number of Processing Elements (PEs) tightly coupled with their own memory (RAM). Firstly, the basic structures of the CP and the PE array are described.

I) Control Processor (CP) The CP has its own instruction and data caches, and controls the operation of the whole XC core. In the SIMD mode, the CP issues instructions to each PE, and in the MIMD mode, manages the state of each PU (such as Run, Stop, or Finish). Moreover the CP has an instruction fetch unit which can issue up to six instructions per clock cycle. And up to six fetched instructions are issued to the CP, and up to five fetched instructions are issued to each PEs. The CP also has a VLIW execution unit which can execute up to six instructions simultaneously.

II) Processing Element (PE) Each PE has its own scratch-pad memory (RAM), and has a VLIW execution unit which can simultaneously execute up to five instructions sent from the CP instruction fetch unit. Each PE is also connected to its two neighboring PEs via a data transfer path (DTP). By using the DTP, each PE can exchange its own data with that of its neighboring PEs. Data burst transfer, called line transfer, is also available for transferring data between the external memory (Ext Mem) and each PE memory (RAM).

Next, we describe an implementation of the PUs, which work independently in the MIMD mode. To realize the MIMD mode at low cost, the hardware of four PEs is reused to realize one PU. The three main ways we reuse the hardware of four PEs are summarized below.

1) The cache system of each PU is configured by reusing the PE's memories and register files. Four PE's memories which are used as scratch pad memories in the SIMD mode, are reused as the memories of the PU's instruction and data caches. The register files of three PEs are also reused to store the PU's cache tags. A hardware based cache coherence mechanism is not implemented, due to the complexity of its implementaion and also the anticipated increase in memory traffic.

2) The control and data paths of the SIMD and MIMD modes are shared. By sharing a large number of instructions (about 70%) between the SIMD and MIMD modes, the instruction decode unit, register file unit, and instruction execution unit of one PE can be mostly reused for one PU.

3) The PU's floating point path is configured by reusing the execution units of the PEs. The execution units of three PEs that are not being used as the PU's

data path are reused to configure a single-precision floating-point path for the PU.

By reusing the hardware of four PEs in these ways, the additional hardware costs for implementing the MIMD mode, including the costs for the ROI transfer instructions described in Section 2.2, have been kept to approximately 10% of the costs for the whole XC core[6].

Finally, we describe in detail how the XC core dynamically switches between the SIMD and MIMD mode is described. The process for switching from the SIMD mode to MIMD mode consists of three steps. Firstly the CP's *cforkinit* instruction, which consumes three clock cycles, is issued to specify which PUs are to be activated, and to assign a memory space to each PU. Next, if more than one PU requires the common instruction code (or data set), the CP's *cforkp (or cforkd)* instruction can be used to fill the instruction (or data) cache of multiple PUs at the same time. Finally, the CP's *cfork* instruction specifying an initial program counter value is used to start execution of each PU. If the contents of the RAM attached to each PE in the SIMD mode must be saved, such operation should be executed before issuing the *cforkp (or cforkd)* and *cfork* instructions. In contrast, the process for switching from the MIMD mode to SIMD mode consists of only one step. When all PUs terminate their processing after explicitly writing back data cache contents by using the *cwb* instruction, the CP's *cwrs* instruction, which consumes only one clock cycle, is issued to inactivate all PUs.

2.2 Region of Interest (ROI) Transfer Instructions

Conventional SIMD architecture is not suitable for executing image recognition algorithms in which the size or content of the target region of an image is different from that of other regions. To execute this kind of algorithm efficiently, the SIMD architecture must reallocate the data within one target region among the PEs in a rather complicated way to make all PEs busy. However such reallocation will incur an additional processing time overhead. Furthermore, the execution time required for the entire algorithm is dominated by the longest execution time among the target regions.

Figure 2 shows one such algorithm, in which whether each target region of interest (ROI) in an image includes a pedestrian or not is verified. The XC core can execute such an algorithm efficiently, because each PU can independently

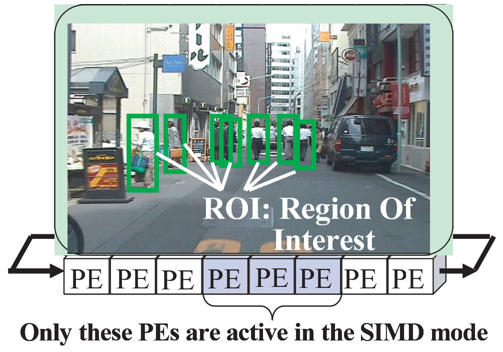


Fig. 2 Example of ROI-type operation.

verify each target region (ROI) in an image by using the MIMD mode. However, it is not appropriate to execute this kind of algorithm, which deals with ROIs, by using a normal cache system, because a normal cache system cannot efficiently transfer the data within each ROI, resulting in a degrading the performance of algorithm execution.

As shown in **Fig. 3**, a normal cache system transfers a rectangular ROI data area in a cache manner, in which all the data included in each cache line from CL1 to CL5 is transferred. Therefore the data around the ROI will also be transferred. Such unnecessary data transfer wastes the memory bandwidth between the cache memory and the external memory, and lowers the cache utilization efficiency by storing unnecessary data in the cache memory. Furthermore, because the PU's cache system is constructed in a cost-effective way by simply reusing the PEs' register files to store the PU's cache tags, the PU's cache does not have sufficient cache lines. Actually, the size of one PU's cache line is 512 bytes, which is quite larger than conventional microprocessors. It can therefore be determined that the PU's cache system is not suitable for ROI-type of data transfer.

To solve this issue, the XC core implements two ROI transfer instructions, called *roiread* and *roiwrite*, which efficiently transfer just the data within each ROI region. By using these ROI instructions, performance degradation owing to unnecessary data transfer can be eliminated. As shown in **Fig. 4**, the two ROI transfer instructions can transfer just the data within a ROI region (from d0 to

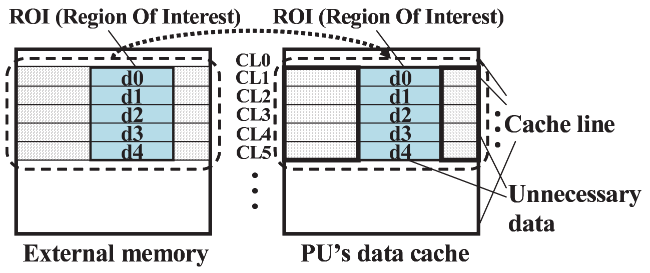


Fig. 3 Transferring ROI in a cache manner.

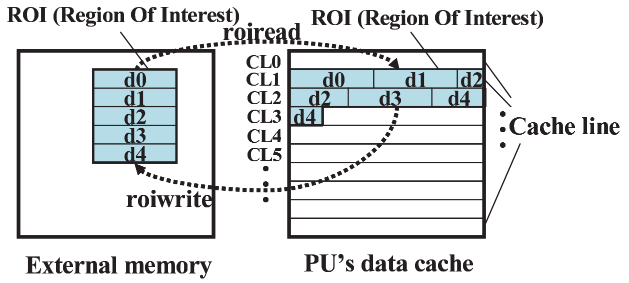
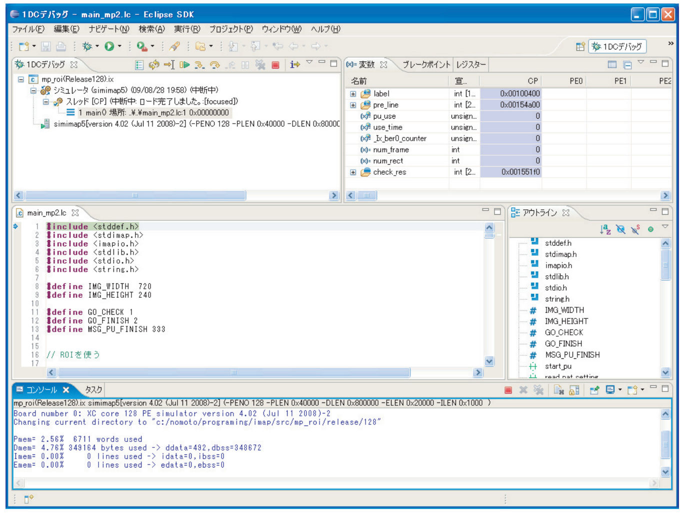


Fig. 4 Transferring ROI in a ROI manner.



```
mp_roi(Release128).ix
  シミュレータ (simimap5) (09/08/28 19:58) (中断中)
    スレッド [CP] (中断中: ロード完了しました。:[focused])
      1 main() 場所: .¥.¥main_mp2.lc:1 0x00000000
  simimap5[version 4.02 (Jul 11 2008)-2] (-PENO 128 -PLEN 0x40000 -DLEN 0x8000C
```

名前	宣...	CP	PE0	PE1	PE2
label	int [1...	0x00100400			
pre_line	int [2...	0x00154a00			
pu_use	unsign...	0			
use_time	unsign...	0			
_Ix_ber0_counter	unsign...	0			
num_frame	int	0			
num_rect	int	0			
check_res	int [2...	0x001551f0			

```c
#include <stddef.h>
#include <stdimap.h>
#include <imapio.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define IMG_WIDTH  720
#define IMG_HEIGHT 240

#define GO_CHECK 1
#define GO_FINISH 2
#define MSG_PU_FINISH 333


// ROIを使う
```

```
mp_roi(Release128).ix simimap5[version 4.02 (Jul 11 2008)-2] (-PENO 128 -PLEN 0x40000 -DLEN 0x800000 -ELEN 0x20000 -ILEN 0x1000 )
Board number 0: XC core 128 PE simulator version 4.02 (Jul 11 2008)-2
Changing current directory to "c:/nomoto/programing/imap/src/mp_roi/release/128"

Pmem= 2.56%  6711 words used
Dmem= 4.76% 349164 bytes used -> ddata=492,dbss=348672
Imem= 0.00%      0 lines used -> idata=0,ibss=0
Emem= 0.00%      0 lines used -> edata=0,ebss=0
```

Fig. 5 Integrated software development environment of the XC core.

d4), and store the data in consecutive locations in the cache memory. Two address pointers are used during execution of the ROI transfer instructions, one for the external memory, and the other for the PU's data cache, each being updated independently to each other. After the ROI transfer, the transferred data can be accessed by using the address pointer for the PU's data cache. Consequently, the wasted memory bandwidth is reduced, and cache memory utilization is improved.

2.3 Software Development Environment

We have developed an integrated software development environment (IDE) based on Eclipse[7] for programmers of the XC core. This environment is shown in **Fig. 5**. The IDE combines all the tools necessary to develop applications, such as a source code editor, an optimized XC compiler, and a graphical debugging tool, etc. And all tools are seamlessly integrated in the IDE, so programmers can easily develop their applications. Furthermore, two different programming models are assumed to develop programes in the XC core, one for the SIMD mode, and one for the MIMD mode. These models are described as follows.

I) SIMD mode programming model

In the SIMD mode, a data parallel extension of C called 1DC (one-dimensional C)[8], is used to explicitly specify the code to be executed using the PE array. In 1DC, entities associated with the PE array are declared by using a sep (or separate) keyword. A sep data item possesses as many scalar elements as a multiple of the number of PEs. Explicitly parallel operations are specified using sep variables in 1DC expressions. **Figure 6** shows the six primitive extensions of 1DC from C. The SIMD programming model is based on the use of the collection of all PE local data RAMs collected to form a 2-D memory plane, where the source, destination, and work spaces are assumed to be stored. This 2-D memory plane is used as a large scratch-pad working area for the 1-D PE array to work on in parallel or in a systolic way.

II) MIMD mode programming model

In the MIMD mode, the normal ANSI C is used for programming the CP and the PUs. Also conventional POSIX Pthreads API-compatible functions, or native thread functions are available for user thread programming. Message passing is

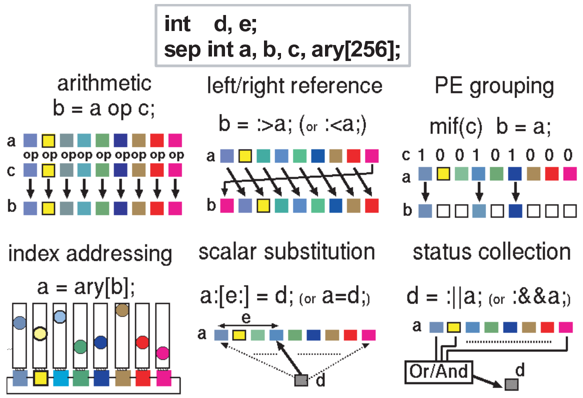


Fig. 6 Six primitive 1DC extended syntax forms.

supported by the typical "send" and "recv" instructions of the CP and the PUs. Message types such as direct (one-to-one), broadcast (one-to-many), and interruption are available. Direct-messages are useful for synchronizing two PUs for exchanging data. For example, in performing ROI region verification for detected object candidates, the CP can post tasks as any-messages, each containing the location information of the region, to the PUs. These any-messages are then received by the PUs, which are waiting for new workloads. An interruption-message can force the receiver PU or CP to branch to a program address specified in the message body. Interruption-messages sent by the PUs to the CP can also be used to implement semaphores, so as to centralize as well as sequentialize accesses to shared resources or variables between tasks.

3. White Line Detection Algorithm for Open Roads

This section firstly provides an overview of the white line detection algorithm for open roads which is used for evaluating the effectiveness of the XC core architecture. Secondly, the process of verifying the white-line segment candidates, which is a part of the white line detection algorithm, and which is not suitable to be executed by a pure SIMD architecture, is explained.

3.1 Overview of the White Line Detection Algorithm

Figure 7 shows a flow chart of the algorithm, and **Fig. 8** shows the image

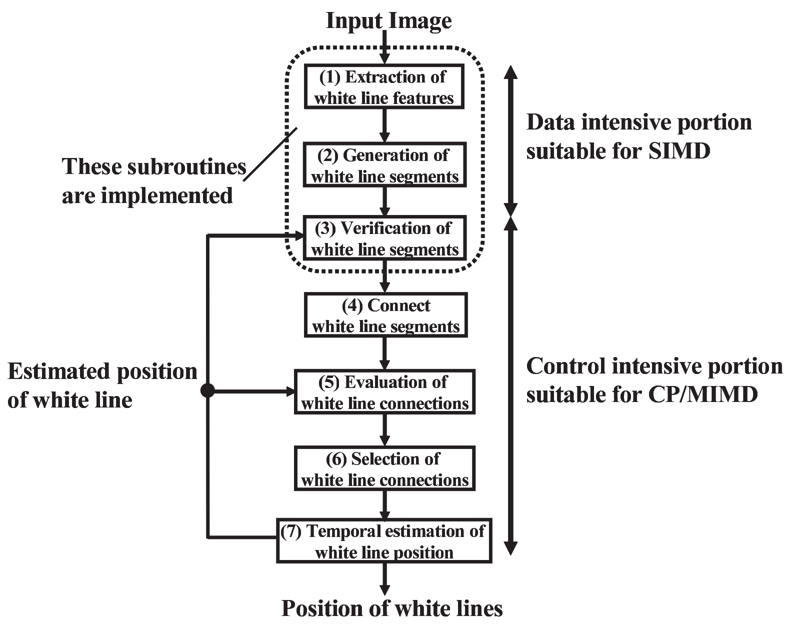

Input Image
(1) Extraction of white line features
(2) Generation of white line segments
(3) Verification of white line segments
(4) Connect white line segments
(5) Evaluation of white line connections
(6) Selection of white line connections
(7) Temporal estimation of white line position
Position of white lines
These subroutines are implemented
Estimated position of white line
Data intensive portion suitable for SIMD
Control intensive portion suitable for CP/MIMD

Fig. 7 White line detection algorithm for open roads.

processing results. As shown in Fig. 7, the white line detection algorithm consists mainly of the following seven subroutines.

(1) From each pixels of image data as shown in Fig. 8-a, white line features are extracted as shown in Fig. 8-b, (2) Neighboring pixels whose extracted white line features are higher than a threshold value are grouped and then the grouped pixels are divided into white line segments as shown in Fig. 8-c, (3) Some of the generated segments that are further judged to be non-white-line are eliminated, as shown in Fig. 8-d, (4) Some of the verified segments are connected with each other, based on the rule of the white line shapes defined by the road traffic regulations, to generate white line connections, (5) The connected segments are evaluated, based on how smoothly the neighboring segments are connected with each other, the sum of the white line features, and the shape similarity between the connected segments and the white lines estimated from the detection results of previous frames, (6) Connected segments whose evaluated score is higher than

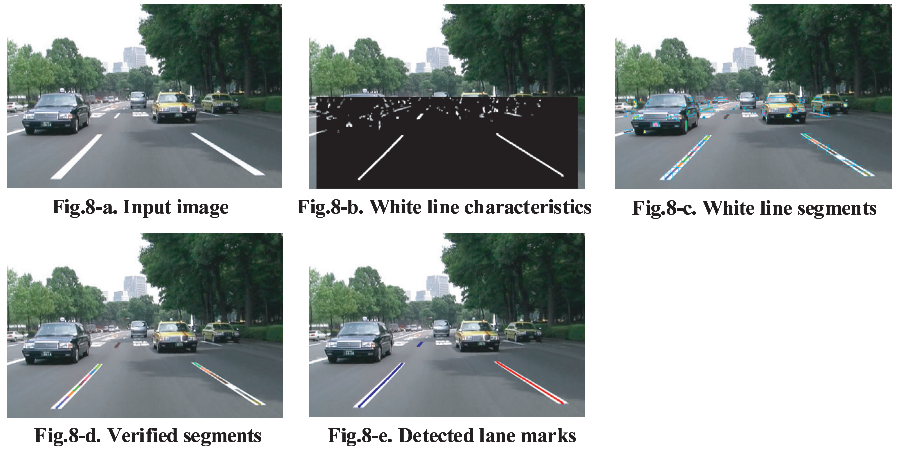


Fig.8-a. Input image
Fig.8-b. White line characteristics
Fig.8-c. White line segments
Fig.8-d. Verified segments
Fig.8-e. Detected lane marks

Fig. 8 Sample images of a proposed algorithm.

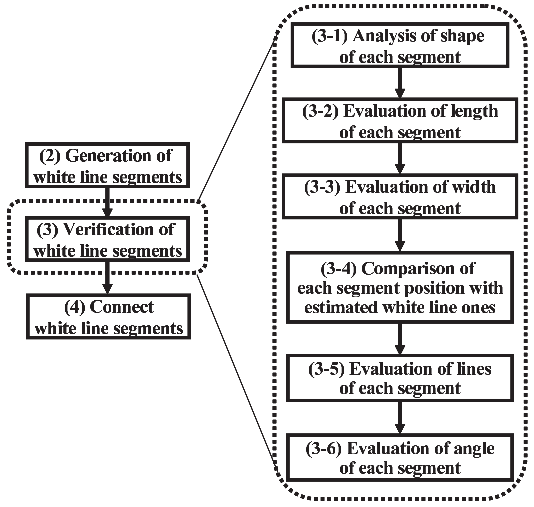

(2) Generation of white line segments
(3) Verification of white line segments
(4) Connect white line segments
(3-1) Analysis of shape of each segment
(3-2) Evaluation of length of each segment
(3-3) Evaluation of width of each segment
(3-4) Comparison of each segment position with estimated white line ones
(3-5) Evaluation of lines of each segment
(3-6) Evaluation of angle of each segment

Fig. 9 Verification process of white line segment candidates.

a threshold value, and higher than the other connected segments, are selected as white lines (one for the left lane, and the other for the right lane) as shown in Fig. 8-e, (7) The position of the detected white lines is used to update the Kalman filter[9] coefficient used to estimate the future position of white lines. The estimated positions of the white lines are used to verify the white line segments or to evaluate the connected segments in the next image.

These seven subroutines are classified as either a data-intensive portion or a control-intensive portion. Subroutines (1) and (2) are classified as data-intensive portions, because a large amount of data has to be processed in the same way. Thus subroutines (1) and (2) are suitable to be implemented in the SIMD mode. On the other hand, subroutines (3) to (7) are classified as control-intensive portions, because they include many branching sequences, which causes the differences in the processing time and processing code for each input data. Therefore these subroutines are suitable to be implemented by the CP or in the MIMD mode.

3.2 Verification of White-Line Segment Candidates

This sub-section provides a detailed explanation of subroutine (3) of the seven subroutines of the white line detection algorithm shown in Fig. 7. Subroutine (3) is used to verify the white-line segment candidates. The reason for this detailed explanation is that subroutine (3) performs a very important operation within the control-intensive portion of the proposed algorithm. Actually subroutine (3) takes up to 99 percent of the execution time in the control-intensive portion, which consists of subroutines (3) to (7).

As shown in **Fig. 9**, subroutine (3) consists of the following six operations. (3-1) The shape of each white-line segment candidate is analyzed to obtain the length, width, and gravity center, (3-2, 3-3) Some of the candidates whose length or width is less than a threshold value are eliminated, (3-4) Some of the candidates whose position is far from the estimated white line position are eliminated, (3-5) After calculating the linearity of each white-line segment candidate, some of the candidates are eliminated if their linearity are less than a threshold value, After calculating each candidate angle made by the candidate axis and camera one, some candidates are eliminated if the angle is not allowed at that position when considering the road shape defined by the road traffic regulations.

4. Evaluation of the Performance of the XC Core

In this section, the performance of the XC core's MIMD mode is evaluated by using a white line detection algorithm for open roads implemented in the XC core. Firstly we consider how to implement the algorithm using the XC core. Next, the performances of implementations using and without using the MIMD mode are compared to show the effectiveness of the MIMD mode.

4.1 How to Implement a White Line Detection Algorithm for Open Roads in the XC Core

Among seven subroutines of the white line detection algorithm shown in Fig. 7, (1) Extraction of white line features, (2) Generation of white line segment candidates, and (3) Verification of white line segment candidates, were selected to be implemented using the XC core, and their execution times were estimated. As these three subroutines dominate a large portion (about 99%) of the whole execution time consumed by the entire algorithm, it is possible to recognize the characteristics of the XC core with respect to the algorithm, by analyzing the results of implementing these subroutines in the XC core. How we implemented subroutines (1) to (3) in the XC core is explained below.

Subroutine (1): Extraction of white line features applies the same filter to all the pixels of a camera image to extract white line features. Because the same operation is applied to a large amount of data, it is suitable to implement subroutine (1) in the SIMD mode, where many pixels can be filtered in parallel by allocating each pixel to each PE of the SIMD array.

Subroutine (2): Generation of white line segments, checks whether all the pixels of the image resulting from subroutine (1) are higher than a threshold value. Neighboring pixels that have passed the threshold test are grouped together to generate white line segment candidates. Because the same operation is applied to a large amount of data, it is suitable to implement subroutine (2) in the SIMD mode, where many pixels can be threshold-tested and grouped in parallel, by allocating each pixel to each PE of the SIMD array.

Subroutine (3): Verification of white line segments, checks whether the shapes of all the white line segment candidates comply with the road traffic regulations, and eliminates those candidates whose shapes do not comply. In this kind of verification, each candidate requires a different execution time, because the operation for each candidate is terminated halfway, according to its shape characteristics (size, angle, shape, etc.). Meanwhile, because each white line segment candidate can be verified independently, it is suitable to implement subroutine (3) in the MIMD mode. In the MIMD mode, each PU verifies each candidate independently from the other PUs, and the CP dynamically manages the assignment of candidates to PUs and collects the execution result from the PUs by using the native thread functions, explained in Section 2.3.

In order to demonstrate the effectiveness of the MIMD mode of the XC core, subroutine (3) was also implemented in the SIMD mode. The estimated execution time using the SIMD mode was compared with that using the MIMD mode. In the SIMD mode, it would be preferable to allocate each white line segment candidate to each PE of the SIMD array in order to execute as many candidates in parallel as possible. However, according to our estimation, transferring the data of each candidate to each PE would incur a considerable overhead with this kind of implementation. Consequently, for the SIMD mode, we chose an implementation in which each candidate is verified sequentially by using the CP.

4.2 Results of Performance Evaluation

The white line detection algorithm for open roads was implemented in the SIMD and MIMD mode by using the above methods. By using a cycle-based simulator of the XC core and four different test scenes (VGA size) as shown in **Fig. 10**, each execution time necessary for each scenes in the SIMD and MIMD mode, were estimated and averaged. And each estimated execution time was compared with the other ones. Furthermore, to efficiently verify the white line segment candidates, which is a subroutine of the white line detection algorithm, it was preferable to use the ROI transfer instructions explained in Section 2.2, because verification is executed for each image region (ROI) of a verification candidate. To demonstrate the effectiveness of the ROI transfer instructions, the verification process was implemented using and without using these instructions, and their execution times were compared. **Table 1** shows the hardware specifications of the XC core used for this evaluation.

Figure 11 shows the averaged execution time distribution for four test scenes, of the white line detection algorithm implemented using the XC core. As shown

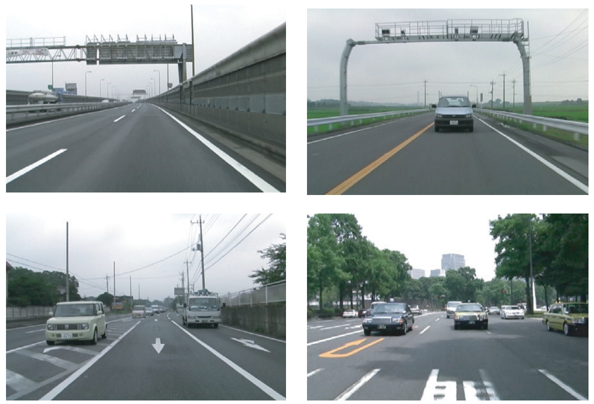


Fig. 10 Four different test scenes for performance evaluation.

Table 1 Performance specifications of the XC core.

	SIMD mode	MIMD mode
Number of PE (PU)	128PE	32PU
Instruction provided	Instruction cache (32 KB, 2way-64entry) @CP	Instruction cache (8 KB, 2way-8entry) @PU
Data provided	Data cache (4 KB, 2way-16entry) @CP, Scratchpad memory (4 KB) @PE	Data cache (8 KB, 2way-8entry) @PU
Data path	5way-VLIW	3way-VLIW
Floating point path	Not supported	Supported
External memory	Amount: 256 MB, Bandwidth: 13.8 Gbps	
Operating frequency	108 MHz @ 90 nm CMOS	

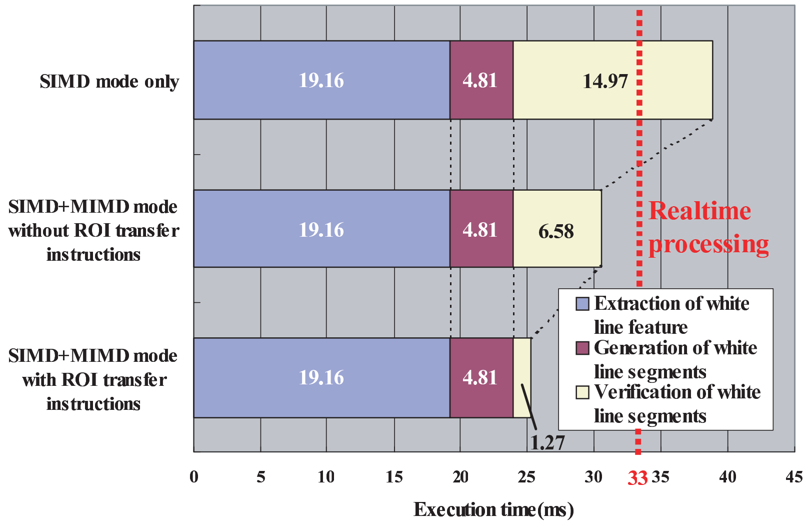


Fig. 11 Execution time of white line detection algorithm.

in Fig. 11, the algorithm can be processed in real time (less than 33 ms) by using the MIMD mode to execute verification of white line segment candidates. The execution time of verification using the SIMD mode was 14.97 ms, but using the MIMD mode, this could be reduced to 6.58 ms when the ROI instructions were not used, and 1.27 ms when they were used. Figure 11 demonstrates how the use of the MIMD mode can improve the performance of the XC core. Furthermore, verification process was executed five times faster by using the ROI instructions, demonstrating that the ROI instructions improved the performance of the XC core. We also evaluated the performance characteristics of the MIMD mode, with changing the number of PUs used to execute verification of white line segment candidates. The verification execution time of the verification process was measured with changing the number of PUs from 2 to 4, 8, 16 and 32.

Figure 12 shows the performance improvement in the MIMD mode compared with the SIMD mode according to the number of PUs used, and when using and not using the ROI instructions. Figure 12 also shows the performance in the MIMD mode when a zero latency memory access is assumed, enabling us to examine how data transfer contention affected the performance of MIMD mode execution. Figure 12 shows that a significant performance improvement can be achieved by using the MIMD mode. For example, the performance was approximately 8.95 times faster when using 16 PUs, and 10.68 times faster when using 32 PUs. Also the performance was up to five times faster when using the ROI instructions. It is assumed that this is because the ROI instructions can substantially reduce performance degradation caused by data transfer contention, which increases as the number of PUs increases. The performance improvement when using the ROI instructions scaled well up to 8 PUs, but for 16 PUs or

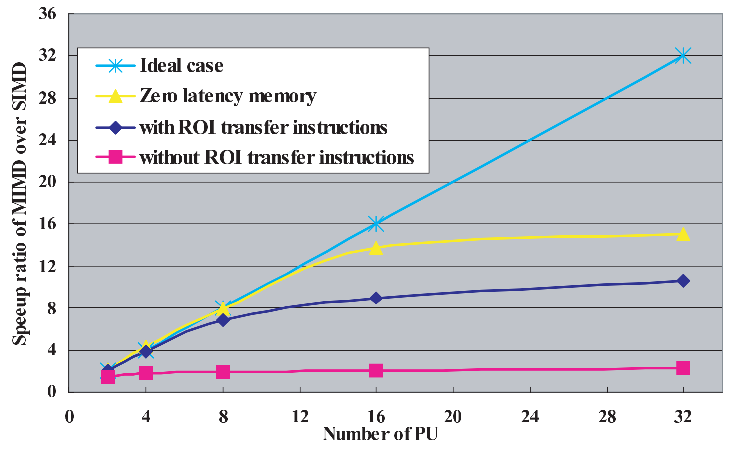


Fig. 12 Performance improvement by using the MIMD mode.

Table 2 Average percentage of detailed tasks configuring PU's operation.

	2PUs	4PUs	8PUs	16PUs	32PUs
I$ miss stall	15.39%	13.92%	13.04%	7.01%	2.98%
D$ miss stall	0.00%	0.00%	0.00%	0.00%	0.00%
Job waiting	2.87%	10.82%	13.60%	37.49%	51.22%
Program body	81.74%	75.26%	73.36%	55.50%	45.80%

32 PUs the improvement rate slowed down substantially. This is likely to be due to the capacity of the memory bandwidth between the external memory and PUs, which becomes insufficient to provide data to all PUs as the number of PUs increases. But as shown in Fig. 12, when the number of PUs increases to more than 16, the improvement rate falls away from that of the ideal case, in spite of using a zero latency memory. In order to investigate the reason for such a slow down in the performance improvement rate, the number of cycles consumed by the PUs, was examined in detail, with the results shown in **Table 2**.

Table 2 lists the average percentage of cycles consumed by various detailed tasks making up the PUs' operations. These tasks include "instruction cache (I$) miss stall", "data cache (D$) miss stall", "job waiting", and "program body". Table 2 suggests that the percentage of time consumed by "job waiting" increases, as the number of PUs increases. This is especially true when 16 and 32 PUs are used.

Table 3 Percentage of cycles consumed by "broadcast candidates" task of CP.

	2PUs	4PUs	8PUs	16PUs	32PUs
Broadcast candidates	2.00%	3.00%	7.00%	12.00%	17.00%

Therefore the following two possibilities can be considered, (1) Although there are enough white line segment candidates to be simultaneously executed by the PUs, the CP cannot broadcast enough candidates to make all the PUs busy, and (2) Although the CP can broadcast enough white line segment candidates to make all the PUs busy, there are not enough candidates to be broadcast to all the PUs.

To verify which of these is correct, we measured the percentage of time consumed by "broadcast candidates" task of the CP. The results are shown in **Table 3**. Table 3 shows that the percentage of time consumed increases, as the number of PUs that are used increases. But the percentage accounts for only 17% of all time consumed by the CP, even when the number of PUs used is 32. This shows that the CP has enough ability to broadcast candidates to all the PUs. Thus, there are not enough candidates to keep 16 or 32 PUs busy. We can therefore assume that possibility (2) is the reason why the performance improvement rate slows down substantially when 16 or 32 PUs are used.

5. Conclusion

In this paper, we implemented a white line detection algorithm for open roads in the XC core, and evaluated the performance of executing the algorithm to show the effectiveness of the MIMD mode of the XC core. Our evaluation showed that the algorithm could be processed in real time (less than 33 ms) by using the MIMD mode to execute verification of white line segment candidates, which is a part of the algorithm that is not suitable to be executed by the SIMD mode. Meanwhile, it was demonstrated that verification could be executed five times faster by using ROI instructions, which can efficiently transfer the ROI of an image. Furthermore, the verification execution time in the MIMD mode was measured with changing the number of PUs from 2 to 4, 8, 16 and 32. The measured results showed that the performance improvement rate when using the ROI instructions could scale well up to 8 PUs, and that the execution time was approximately 8.95 times faster when using 16 PUs, and 10.68 times faster when using 32 PUs. Meanwhile, we investigated why the improvement rate slowed down when more than 16 PUs were used. The investigation demonstrated that the reason was that the verification of white line segment candidates does not have sufficient parallelism to keep all the PUs busy. The above results showed that the performance of image recognition applications can be significantly improved by using the MIMD mode, in addition to the SIMD mode.

References

1) Kimura, Y., Kato, T., Ohta, M., Ninomiya, Y., Takagi, Y., Usami, M. and Tokoro, S.: Stereo vision for obstacle detection, *The 13th Intelligent Transportation Systems World Congress* (2006).
2) Tsuji, T., Hattori, H., Watanabe, M. and Nagaoka, N.: Development of night-vision system, *IEEE Trans. Intelligent Transportation Systems*, Vol.3, Issue.3, pp.203–209 (2002).
3) Muramatsu, S., Otsuka, Y., Takenaga, H., Kobayashi, Y., Furusawa, I. and Monji, T.: Image processing device for automotive vision systems, *IEEE Intelligent Vehicle Symposium*, Vol.1, pp.121–126 (2002).
4) Tanabe, J., Taniguchi, Y., Miyamori, T., Miyamoto, Y., Takeda, H., Tarui, M., et al.: Visconti: multi-VLIW image recognition processor based on configurable processor, *IEEE Custom Integrated Circuits Conference*, pp.185–188 (2003).
5) Kyo, S., Okazaki, S. and Arai, T.: An integrated memory array processor architecture for embedded image recognition systems, *IEEE Trans. Comput.*, Vol.56, Issue.5, pp.622–634 (2007).
6) Kyo, S., Koga, T., Hanno, L., Nomoto, S. and Okazaki, S.: A low-cost mixed-mode parallel processor architecture for embedded systems, *International Conference on Supercomputing*, pp.253–262 (2007).
7) http://www.eclipse.org/.
8) Kyo, S., Okazaki, S. and Arai, T.: An Integrated Memory Array Processor Architecture for Embedded Image Recognition Systems, *International Symposium on Computer Architecture*, pp.132–145 (2005).
9) Kalman, R.E.: A new approach to linear filtering and prediction problems, *Transactions of the ASME, J. Basic Engineering*, Vol.82, pp.35–45 (1960).

(Received May 21, 2009)
(Revised September 4, 2009)
(Accepted October 31, 2009)
(Released February 15, 2010)

(Recommended by Associate Editor: *Shinsuke Kobayashi*)

Shohei Nomoto received his B.E. and M.E. degrees in systems & information engineering from Tsukuba University in 2003 and 2005, respectively. He joined NEC Corporation in 2005 and is currently a research staff member at the System IP Core Research Laboratories. His research interests include parallel processing systems and processor architecture. He is a member of the Information Processing Society of Japan (IPSJ).

Shorin Kyo received his B.E., M.E., and Ph.D. degrees in precision engineering from the University of Tokyo in 1987, 1989, and 2004, respectively. He joined NEC Corporation in 1989 and is currently a principal researcher in the System IP Core Research Laboratories. He has been involved in research on parallel processing system and processor architecture, parallel language and compiler design, and image processing. Between 1994 and 1995, he was a visiting researcher in the Department of Applied Physics at Delft University of Technology. He is a member of the Information Processing Society of Japan (IPSJ) and the Institute of Electronics, Information and Communication Engineers (IEICE).

Shinichiro Okazaki received his B.E. and M.E. degrees in electronic engineering from Osaka University in 1982 and 1984, respectively. He joined NEC Corporation in 1984 and is currently a senior principal researcher in the System IP Core Research Laboratories. He is engaged in research on image processing and parallel processing architecture. He is a member of the Institute of Electronics, Information and Communication Engineers (IEICE).