Vol.2010-ARC-190 No.24
2010/8/4

An Out-of-order Vector Processing Mechanism for Multimedia Applications

Ye Gao,†1 Ryusuke Egawa,†2,†3
Hiroyuki Takizawa†1,†3 and Hiroaki Kobayashi †2,†3

Nowadays, multimedia applications (MMAs) form an important workload for general purpose processors. The vector processing is considered as the most potential approach for MMAs due to plenty of data level parallelism involved in them. However, the tradition vector architectures obey an in-order issue policy (IIP). The IIP issue policy blocks the following instructions to be issued, no matter whether they are ready to be issued or not.

This paper proposes a media-oriented vector architectural extension with an out-of-order vector processing mechanism (OVPM). The OVPM overcomes the inefficiency on utilization of the memory bandwidth and vector functional units. As a result, the proposed architecture achieves a higher performance with lower hardware cost than the traditional one. This paper evaluates the proposed architecture with architectural design parameters and finds out the most efficient size for the vector architecture when performing MMAs.

1. Introduction

Various multimedia applications (MMAs), such as speech recognition, video encoder/decoder, provide extraordinarily fantastic services to consumers. The next generation MMAs have at least two requirements for the coming hardware devices. One is that they require a higher performance hardware device to match the demand of high speed and high quality media processing. The other is that different MMAs are required to perform on the same device because of rapid development on their varieties. Therefore, the next generation processor that executes MMAs should have both high performance and high programmability.

In order to meet these two demands, we expand the design space of the vector architecture to enhance the potential of general purpose processors (GPPs) on MMAs. The vector architecture is able to accelerate MMAs by effectively exploiting DLP [1], which is massively involved in MMAs. However, traditional vector processors have been mainly designed for scientific and engineering domains [2] [3]. They have extremely long vector registers and obey the in-order issue policy. The in-order issue policy leads to the inefficiency use of memory bandwidth and functional units, because it prevent the following instructions from being issued.

Regarding scientific applications, they have very long vector lengths in their algorithms, they can efficiently use the long vector registers to hide the latency of pipeline stalls caused by in-order policy. However, MMAs own shorter vector lengths than scientific applications. Short vector lengths lead to exposing the latencies caused by pipeline stalls due to in-order issue policy. Therefore, the traditional vector processor cannot execute MMAs efficiently.

This paper proposes a media-oriented vector architecture with an out-of-order (OoO) vector processing mechanism (OVPM) to overcome the inefficiency on processing short vectors. First, we analyze the behavior of the traditional vector architectures which obey an in-order issue policy and clarify their inefficiencies in performing MMAs. Second, a vector microarchitecture with OVPM is proposed to improve the inefficiency of the in-order issue policy. Third, two kinds of OoO issue policies are proposed in this paper. One is a load-forwarding policy (LFP), which means that only a memory access instruction can be issued in an OoO fashion. The other is a complete OoO policy (COP), meaning that all the vector instructions enable to be issued in an OoO fashion. Fourth, the proposed vector architecture is evaluated with several architectural design parameters to find out an efficient configuration.

The rest of the paper is organized as follows. Section 2 illustrates the related work of this paper. Section 3 discusses the limitations of the in-order policy when performing MMAs. Section 4 proposes a novel vector architecture with an OVPM for MMAs in order to overcome the inefficiencies of the in-order policy. Section 5 evaluates the proposed mechanism, and Section 6 gives the conclusions of this paper.

†1 Graduate School of Information Sciences, Tohoku University
†2 Cyberscience Center, Tohoku University
†3 JST CREST

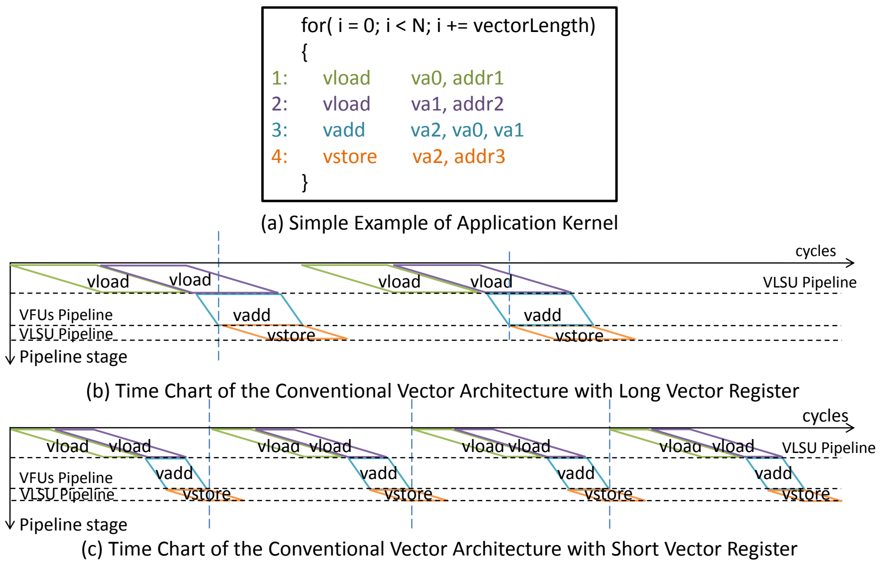
2. Related Work

A popular approach to high-performance computing of MMAs with high programmability is to extend GPPs with SIMD instructions. The modern SIMD extensions support 4-way parallel processing [4] [5], even though the next generation product such as AVX [6] only supports 16-way parallel processing. For this kind of approach, the limited parallel processing ability of hardware cannot satisfy the requirement of high performance for the next generation MMAs, which own a large amount of DLP.

Compared with the SIMD architectural extension, a vector architectural extension has higher ability to process data parallelism by using parallelized arithmetic pipelines and large capacity vector registers [7]. It employs a latency-tolerant load/store unit with an interleaved memory system that has a contribution to hide long memory latency. In this way, the vector architectural extension can be expected to achieve much higher performance than the SIMD extension.

Graphics processing units (GPUs) are the emerging high performance processors by using the SIMD and MIMD. GPUs employ context switching approach to hide the pipeline stalls. A group that constituted by several threads is a basic switching unit [8]. When a group is blocked in a GPU, the context will be switched to execute another group to hide the stall. The ability of hiding stalls depends on the number of groups. The more groups are, the higher the ability of hiding stalls is. The number of groups is determined by the number of each group's execution context size, which is assigned by hardware thread scheduler according to the number of registers in the group. Therefore, sometimes, the number of groups is not big enough to hide the pipeline stalls. In that case, GPUs will expose the pipeline stalls latencies and their performance will be decreased.

The vector architecture is one of the most potential candidates for media processing. Kozyrakis et al. [9] have proposed the Vector IRAM (VIRAM) architecture, which is a register-to-register vector architecture. The VIRAM architecture supports small data sizes and flexible vector length controlled by a dedicated register. It also provides various memory access patterns such as strided and indexed accesses. Although VIRAM is modified to include the memory latency in pipeline stages, it underutilizes the memory bandwidth due to the in-order issue policy, which makes the vector pipelines frequently stalled.



```
   for( i = 0; i < N; i += vectorLength)
   {
1:    vload      va0, addr1
2:    vload      va1, addr2
3:    vadd       va2, va0, va1
4:    vstore     va2, addr3
   }
```

(a) Simple Example of Application Kernel

(b) Time Chart of the Conventional Vector Architecture with Long Vector Register

(c) Time Chart of the Conventional Vector Architecture with Short Vector Register

Fig. 1 Time Chart of the Behavior of In-order Issue Policy.

3. Inefficiency of In-order Issue Policy

Most of the modern vector architectures such as [10] obey the in-order issue policy. Figure 1(a) presents an example of a simple loop to illustrate the inefficient memory accesses of in-order vector architectures, and its time chart is shown in Figure 1(b). Each parallelogram in Figure 1 shows a vector pipeline operation. The behavior of the traditional vector architecture is summarized as follows.

(1) *The first and second instructions*: Vector load instructions, *vload*, are decoded and executed by generating the memory addresses at the address generation unit (AGU).

(2) *The third instruction*: A vector addition instruction, *vadd*, is decoded and then stalled in the issue stage until *va*0 and *va*1 are ready.

(3) *The fourth instruction*: A vector store instruction *vstore* is stalled in the

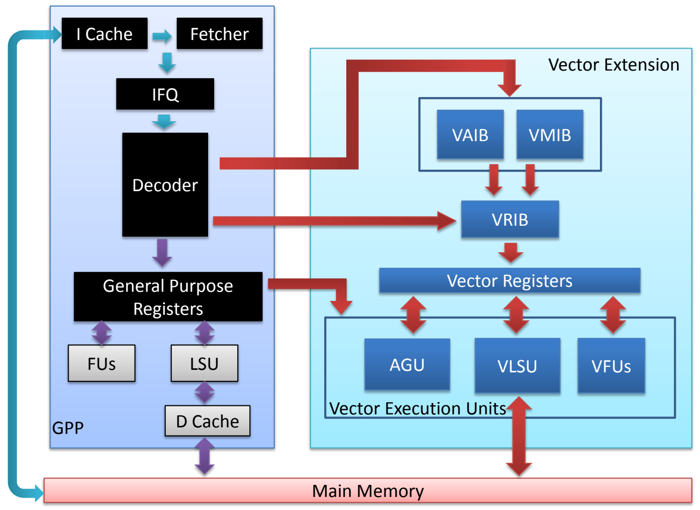


Fig. 2 Block Diagram of Proposed Vector Micro-Architecture.

decode stage unless the previous instruction *vadd* is issued, because of the in-order execution policy for the traditional vector architectures. Then, it is stalled in the issue stage to wait for the results from the *vadd* due to the input data dependency.

(4) *The fifth instruction*: The first instruction of the second iteration (*vload*) is stalled in the decode stage, unless the last instruction (*vstore*) of the first iteration is issued, no matter whether its operands are ready or not. This makes the first *vload* of the second iteration expose the memory latency. The exposure of memory latency occurs in each iteration. The following instructions show the same behavior as these five instructions.

As a result, the vector store instruction is held in the issue stage. It blocks the following instructions, such as the vector load instruction for the second loop, from being issued. The same stall will occur in each iteration, resulting in a large performance loss. In addition, as shown in Figure 1(c), since the problem size is fixed, a long vector register means that the number of data can be processed by one instruction increases and the number of iterations decreases. Therefore, for the traditional vector architectures, it is needed to use long vectors to keep the performance.

Table 1 New Hardware Units for Proposed Vector Architecture.

Hardware Unit	Abbreviation
Vector Arithmetic Instruction Buffer	VAIB
Vector Memory Instruction Buffer	VMIB
Vector Ready Instruction Buffer	VRIB
Vector Function Units	VFUs
Vector Load and Store Unit	VLSU
Address Generation Unit	AGU

However, long vector registers have several drawbacks for executing MMAs as follows.

- Not all of MMAs have a very long vector length. This means that the applications with a short vector length would underutilize the long vector registers.
- Long vector registers lead to a high hardware cost on occupation areas, power consumption and access latency [11].
- Long vector registers imply a small number of vector registers due to the restrictions of area, power consumption and so on. A small number of vector registers lead to register conflicts which would decrease the performance.

Therefore, a newly-designed vector architecture for MMAs is required to efficiently process short vectors in terms of hardware efficiency.

4. OoO Vector Processing Mechanism

This section proposes a novel media-oriented vector architecture as illustrated in Figure 2. In order to perform a vector instruction, the proposed architecture introduces new hardware units listed in Table 1. Those units are colored dark blue in Figure 2. In Figure 2, I Cache, IFQ, FUs, LSU, and D Cache represent an instruction cache, an instruction fetch queue, functional units, a load/store unit, and data cache, respectively.

4.1 Hardware of OVPM

The OVPM is proposed in order to overcome the problem caused by the in-order issue policy. There are three main hardware units that are renaming unit, reorder unit and commit unit to realize the OVPM. The renaming unit is used to remove the false data dependence including the write-after-write and write-after-read. It is also responsible for checking the true data dependence read-after-write among instructions [12].

Regarding the reorder unit, we add two new instruction buffers, VAIB and VMIB, in the vector datapath as the reorder buffers. The proposed OVPM adopts a physical register file based approach, which just contains a few bits to identify instructions and input/output registers without the value of vector registers. In this way, the VAIB and VMIB can be implemented at a small hardware cost.

The commit unit is used to retire the renamed register in a register alias table contained in the rename unit and instructions in VAIB and VMIB by instruction sequence if the corresponding instruction is finished.

In order to decrease the hardware overhead of OVPM, the proposed vector architectural extension use the GPP's the commit unit and rename unit to perform the renaming and commit process.

4.2 Instruction Issue Policies

Two kinds of issue polices are proposed in this paper: LFP and COP. LFP only supports memory instruction to be issued in an OoO fashion in order to improve the utilization efficiency of memory bandwidth. While, COP allows both memory instruction and arithmetic instruction to be issued in an OoO fashion, aiming at the efficient use of the vector function units as well as memory bandwidth.

The vector instructions are fetched, decoded, and renamed in the scalar. Then the vector memory instructions and arithmetic instructions are delivered to VMIB and VAIB, respectively. In the issue stage, for both LFP and COP, VMIB issues an instruction to VRIB as long as all of its operands are ready. The difference between LFP and COP is that the former policy issues arithmetic instructions in order while the latter does out-of-order. In this way, these policies can prevent underutilization of memory bandwidth and vector function units that is caused by the conventional in-order policy. Figure 3 uses the same example as shown in Figure 1 to compare the behavior of the LFP OoO mechanism with that of in-order execution. The procedure is described as follows.

(1) *The first and second instructions*: A vector load instruction *vload* is decoded and executed as long as the memory addresses for *vload* are generated by AGU.

(2) *The third instruction*: A vector addition *vadd* is decoded and stalled in the issue stage due to the dependency between *va*0 and *va*1. The *vadd* is not issued until its input data from two *vload* instructions become ready.

(3) *The fourth instruction*: A vector store instruction *vstore* is decoded and stored in VMIB. The *vstore* stays in VMIB to wait for the results from the *vadd* due to data dependency.

(4) *The fifth instruction*: The first instruction of the second iteration (*vload*) is decoded and dispatched to VMIB. As long as its operands are ready, it is delivered to VRIB, no matter whether it is the first element in VMIB or not. In this way, the first *vload* of the second loop can be issued rather than be stalled by the previous instruction that has not been issued yet.

As mentioned above, the proposed vector architecture can keep executing memory access instructions and efficiently hide the memory latency. Therefore, as Figures 3 (a) and (b) show, the proposed load-forwarding mechanism can handle the short vectors efficiently to take advantage of memory bandwidth.

Table 2 Processor Configuration.

Vector ALU operation latency/issue latency	10/1 cycles
Vector multiplier operation latency/issue latency	10/1 cycles
Vector division operation latency/issue latency	15/1 cycles
Vector ALU number/pipeline number	1/8 cycles
Vector multiplier number/pipeline number	1/8 cycles
Vector division number/pipeline number	1/8 cycles
Vector register num/size	8/8192 bits
Frequency / Peak Performance	1 GHz / 16 Gflop/s

5. Performance Evaluations

5.1 Experimental Setup

5.1.1 Processor Model

We developed a simulator of the proposed architectural extension based on the SimpleScalar toolset [13] to investigate its performance on media workloads. The vector instruction set is implemented by manually inserting vector instructions as instruction annotations to PISA instruction set. Table 2 summarizes the parameters of the vector architectural extension. The peak computational

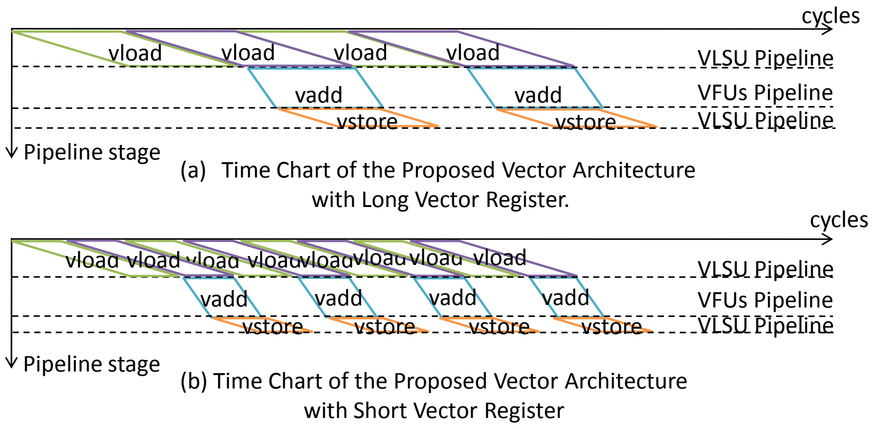

cycles
vload vload vload vload
VLSU Pipeline
vadd vadd
VFUs Pipeline
vstore vstore
VLSU Pipeline
Pipeline stage
(a) Time Chart of the Proposed Vector Architecture with Long Vector Register.
cycles
vload vload vload vload vload vload vload vload
VLSU Pipeline
vadd vadd vadd vadd
VFUs Pipeline
vstore vstore vstore vstore
VLSU Pipeline
Pipeline stage
(b) Time Chart of the Proposed Vector Architecture with Short Vector Register

Fig. 3 Time Chart of Load-Forwarding Mechanism.

performance of the vector extension is achieved when both vector adder and multiplier are working at the same time by a chaining mechanism. Accordingly, the peak computational performance of a vector extension is 16 Gflop/s.

The baseline processor whose frequency is 3.2 GHz is a 4-way superscalar processor without vector architectural extensions. We assume that it can achieve peak computational performance when four floating point function units work simultaneously. Therefore, the peak computational performance of the baseline scalar processor is 12.8 Gflop/s.

5.1.2 Benchmark Programs

Six multimedia benchmarks are introduced to evaluate the architectural implications of GPPs with vector extensions. Table 3 lists the benchmarks along with their brief descriptions. Those benchmark programs are selected from the PARSEC benchmark suite [14] and ALPbench benchmark suite [15]. Both of them include emerging MMAs containing massive DLP.

The evaluation is performed in the following steps. First, in advance of the performance evaluation, hot kernels of each benchmark program are detected by using the profiling tool, Gprof. Second, the kernel codes efficient for vector processing are extracted from the benchmark programs. Third, these kernels are compiled to assembly codes by the sxcc [16], which is a C compiler for NEC SX series vector supercomputers. Using the optimized assembly codes generated by sxcc, the key computation part of each kernel is manually replaced with the embedded assembly code of vector extension instructions. Fourth, the source code is cross-compiled to the binary code by using the Simplescalar's C compiler. Finally, the binary codes with embedded vector instructions are used for the simulation.

Table 3 The Descriptions of Benchmarks.

Benchmarks	Domain	Vectorization Ratios	Vector Length
sphinx3	speech recognition	99.4%	4096
faceRec	face recognition	98.3%	173
raytracer	animation	99.6%	1080
vips	image processing	98.3%	79
M x M	matrix-matrix multiplication	98.9%	1000
V x M	vector-matrix multiplication	99.1%	1000

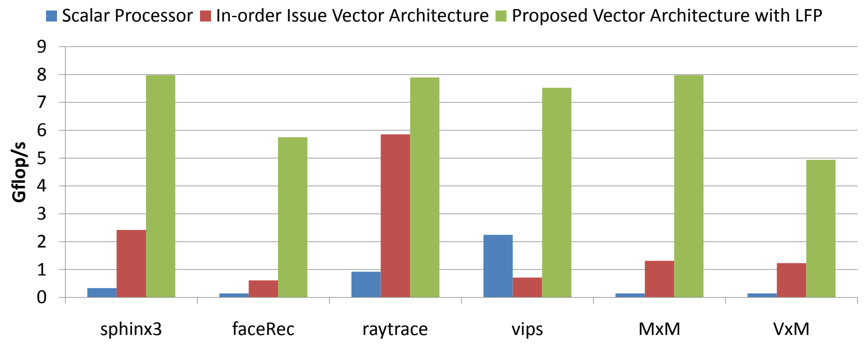


Fig. 4 Sustained Performance (Gflop/s) Evaluation.

5.2 Evaluation Results

5.2.1 Base Performance

We compared the sustained performance of proposed architectural extension with LFP, the baseline scalar processor and the traditional one. The maximum vector length (MVL) is 256, which value is used in most of traditional vector processors. In this paper we use MVL to represent length of a vector register, because their values are often the same. VAIB and VMIB store up to 512 entries

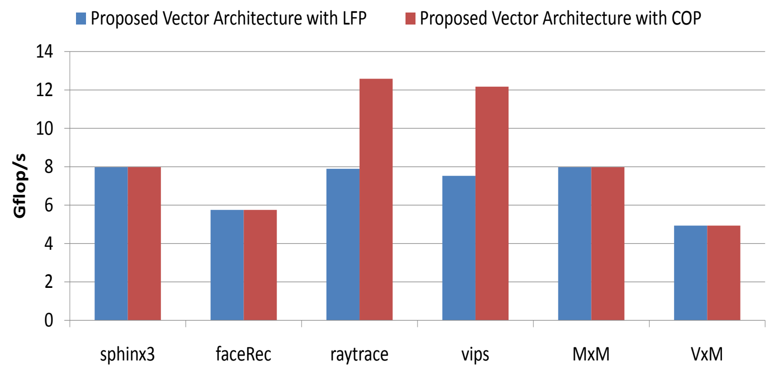


Fig. 5 Impact on Different Issue Policies.

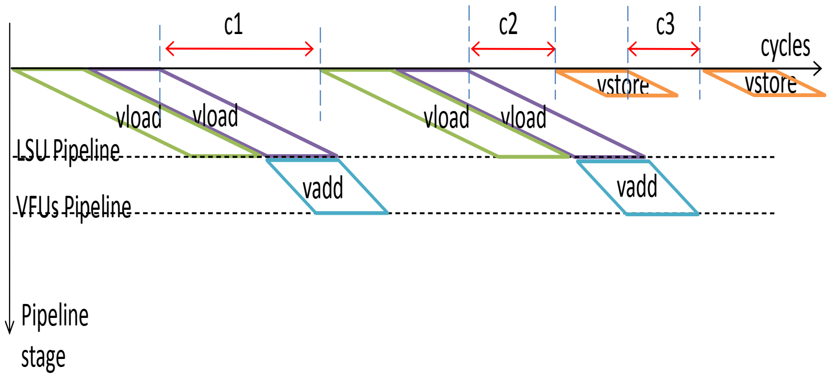


Fig. 6 The Defination of VMBC.

in the proposed vector architecture.

Figure 4 shows Gflop/s for each benchmark program and each architecture. The proposal achieves 28x higher performance than the baseline scalar processor and 5x higher performance than the traditional one on average. This is not only because the proposed architectural extension has a higher peak performance, but also because it achieves a higher computational efficiency as shown in Figure 4. This indicates that the proposal takes advantage of massive DLP involved in MMAs and utilizes memory bandwidth efficiently. The benchmark `vips` has performance reduction on the traditional vector architectural extension. This is because the vector length of the kernel is so short (the vector length is 79) that the kernels cannot hide the start-up overhead, which implies the traditional vector architecture cannot handle short vectors efficiently. On the other hand, proposed vector architectural extension still attains significant performance gain, compared to the traditional one.

The evaluation results imply that the vector architectural extension can achieve a high performance to execute MMAs even with a low frequency. The proposed vector architecture achieves a higher computational efficiency than the traditional one due to efficient utilization of memory bandwidth, which will be analyzed in subsection 5.2.3.

5.2.2 Impact on Issue Policies

Figure 5 depicts the sustained performance of OoO vector architectural extension with LFP and COP. These simulations compare the performance (Gflop/s) of the different instruction issue policies for OVPM.

The results show that there is no performance improvement in the cases of $sphinx3$, $faceRec$, MxM and VxM, even if the issue policy is changed from LFP to COF. These benchmarks are memory -intensive programs, whose performances are mainly influenced by memory access instructions. Therefore, OoO issue for arithmetic instruction has few effects on them. On the other hand, in the cases of *raytrace* and *vips*, the COP works effectively on them. These benchmarks are computation-intensive programs, which involve a large amount of arithmetic instructions. Therefore, OoO issue for arithmetic instructions allows the benchmarks to efficiently use vector function units. Therefore, COP improves the performance for the benchmarks.

Although COP enables to improve the performance for some benchmarks, it also takes more power consumption than LFP. Compared with LFP, it needs to more frequently wake up VAIB so as to find out an operand-ready instruction to be issued, to access the renaming unit, and to use the commit unit. All of these behaviors will lead to power dissipation. Therefore, it is necessary to evaluate trade-off between performance and power dissipation for COP in the future.

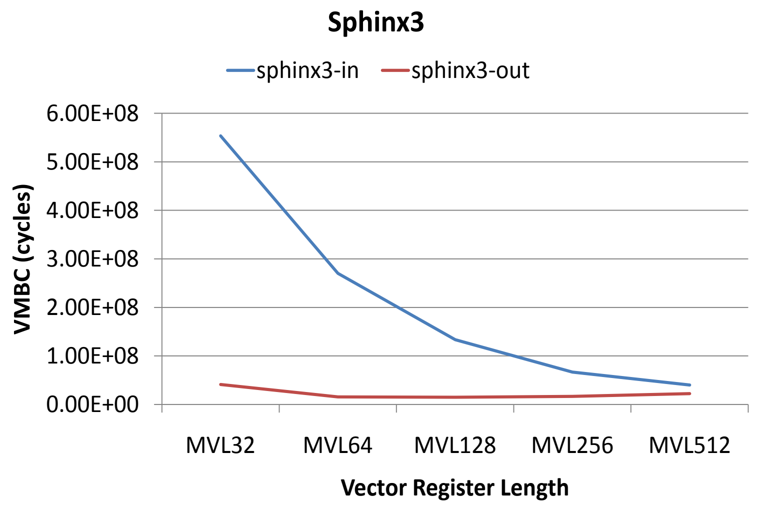


Fig. 7 VMBCs Evaluation.

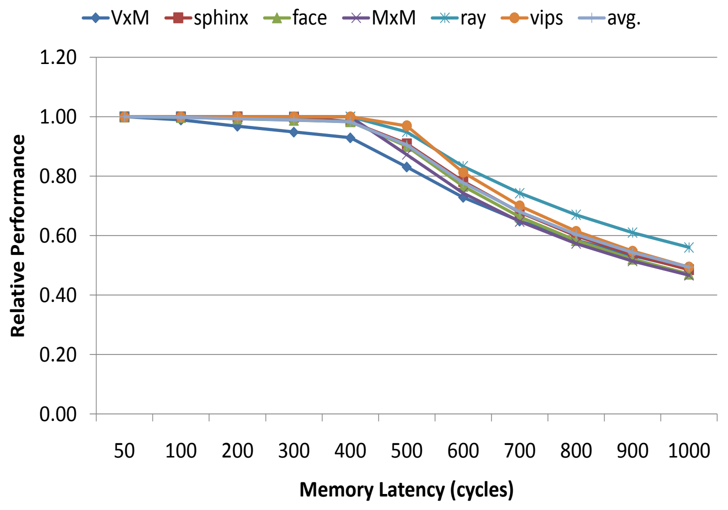


Fig. 8 Memory Latency Impact on Performance Degradation.

5.2.3 Memory Bandwidth Utilization

In this paper, vector memory blank cycles (VMBCs) are evaluated to prove that the main contribution of performance improvement of the proposed architecture comes from the efficient utilization of memory bandwidth. VMBCs are the accumulation of cycles between every two memory access instructions. For example, in Figure 6, VMBCs are obtained by summing up $c1$, $c2$, and $c3$. VMBCs indicate the utilization efficiency of memory bandwidth and VLSU pipelines. For the same program, smaller VMBCs means that the underutilizing cycles of memory bandwidth are short. Therefore, smaller VMBCs imply the more efficient utilization of memory bandwidth.

Figure 7 shows that VMBCs for a benchmark, `sphinx3`, change as MVL increases for example. In the figure, *sphinx3-in* and *sphinx3-out* mean VMBCs of the traditional vector architecture and the proposed architecture, respectively. The VMBCs of the proposed architecture are always smaller than those of the traditional architecture with any MVL. This implies that the proposed architecture is more efficient to use the memory bandwidth and the VLSU pipelines. Moreover, VMBCs decrease as the speedup ratio increases. This proofs that the main factor of performance improvements on the proposed architecture is the efficient use of the memory bandwidth.

5.2.4 Tolerance of Memory Latency

Over the years, CPU frequencies have been rising faster than those of the memory, so memory access latencies in CPU cycles are increasing. The long memory latencies will degrade the CPU performance. Therefore, the processors that execute the upcoming MMAs should be able to tolerate long memory latencies in order to reduce the performance loss and improve the efficiency of using hardware resources.

Figure 8 depicts the performance with increasing memory latencies from 50 cycles to 1000 cycles. The y-axis shows the performance relative to the case where memory latency is 50 cycles.

The results show a clear trend: although the performance decreases with the memory latency increasing, the proposal can tolerate a quite long memory latency. In the case where the latency is 500 cycles, the performance degradation

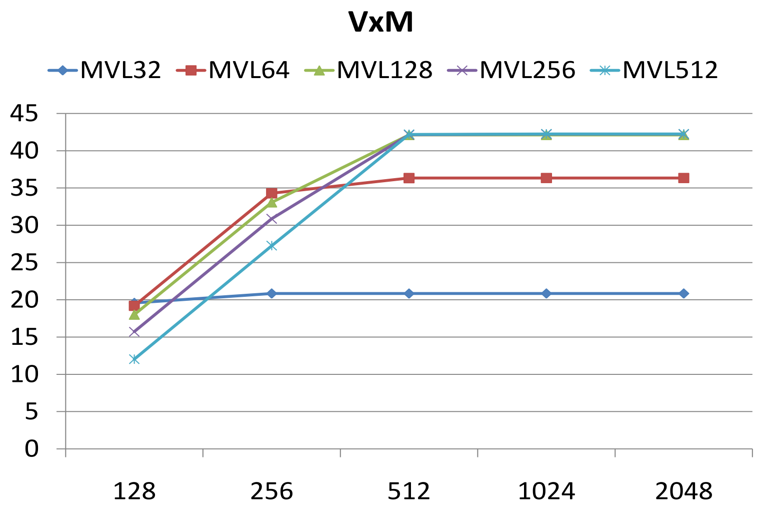
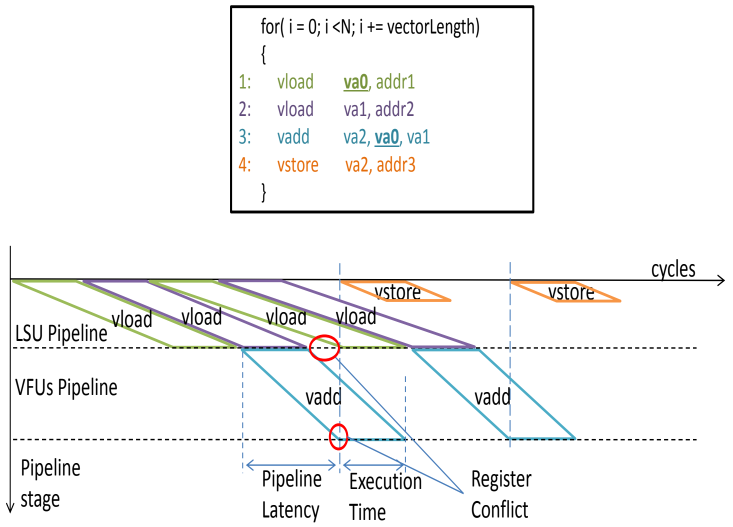


Fig. 9 Effects of the Instruction Buffer Size.



```
for( i = 0; i <N; i += vectorLength)
{
    vload     va0, addr1
    vload     va1, addr2
    vadd      va2, va0, va1
    vstore    va2, addr3
}
```

Fig. 10 Register Conflict.

is less than 17% of the performance achieved with the latency of 50 cycles. Even in the case of a very long latency of 1000 cycles, performance degradation is 51% on average. Therefore, these results suggest the proposed architecture is tolerant of memory latency because vector processing can efficiently hide the latency.

5.2.5 Effects of Instruction Buffer Size and MVL

In this subsection, different instruction buffer sizes and MVLs are evaluated in order to find out an appropriate architectural design configuration. Figure 9 shows the relative performance of proposal with LFP using `VxM` as an example, when the buffer sizes and MVLs are changed. The Y axis indicates the speedup of proposed vector extension normalized by the baseline scalar processor. The performance improves with the increase in the instruction buffers size, until the buffers size reaches 512 entries. When the buffers size is larger than 512, no more performance gain is obtained in any length of vector register. When the length of the vector registers equals to 128, and buffer size is 512 entries, the proposed vector architecture attains a comparable or better performance in comparison with the other situations.

When the MVL is shorter than 128, the vector register conflicts occur and affect the performance improvement. Figure 10 illustrates the vector register conflicts by using an simple example in the case of MVL32. The *execution time* of vector instructions is four cycles because the MVL is 32 and the number of vector pipelines is eight. The latency of the vector pipeline for an addition is assumed as 10 cycles. Therefore, the *vload* operation in the second iteration would have stored to the vector register *va*0 before the *vadd* operation in the first iteration. However, the *vadd* operation does not release the *va*0 registers until it is finished. Thus, the VLSU pipeline has to stall several cycles (6 cycles in this case) to wait for the completion of the *vadd* operation. In every iteration, this kind of stall happens when the execution time is shorter than the pipeline latency. Although the renaming mechanism is able to alleviate the influnce of vector register conflicts on the performance, it also occurs frequently when the MVL is short, due to only eight vector register in the proposed vector architecture.

When the buffer size is smaller than 512, the performance decreases because of the limited capacity of instruction buffers. The vector instructions are decoded at one time and stored in the instruction buffers. Small size of instruction buffers indicates that the instruction buffers are easily to be filled up. If the instruction buffers become full, the decode stage stalls until it can send instructions to the instruction buffers again. Therefore, the performance decreases.

5.2.6 Hardware Cost

The additional hardware cost is mainly required for implementing VMIB and VAIB. The rough calculation of the components of VAIB and VMIB mentioned in Section IV is as follows. ID costs 10 bits in the case of 512 entries. *opcode* uses 7 bits. Checking the *output_dependency* and *input_dependency* are 1 bit and 2 bits, respectively. Since VAIB and VMIB own the same components, it takes (10+7+2+1)*512*2 = 20480 bits. Actually, the proposal employs shorter vector registers than traditional ones. The hardware cost reduced from shorter vector registers is larger than that increased caused by VAIB and VMIB.

6. Conclusions

The media-oriented vector architecture with OVPM as presented above aims at realizing a vector extension to enhance the potential of GPPs on MMAs. It overcomes the inefficiencies in MMAs for the conventional vector architectures which obey in-order issue policy. The OVMP is implemented by using two instruction buffers. It overcomes the inefficiencies in MMAs for the traditional vector architectures, which obey an in-order policy.

With this approach, the proposed vector architecture obtains up to 28x speedup on average. The evaluations of VMBC prove that the efficient utilization of memory bandwidth is the main contribution to the high performance on proposed architecture. Finally, the evaluation shows the proposed vector architecture whose MVL is 128 and instruction buffer sizes are 512 attains a comparable or better performance.

The ability of data transportation between the off-chip memory and the chip has a large influence on the performance. It is difficult to increase off-chip memory bandwidth because of limited pin count of the chip. In the future work, we will explore a way to efficiently use on-chip memories (or caches) and their high on-chip memory bandwidth. Moreover, the hardware cost of proposed vector extension is not will be evaluated, in terms of power dissipation and chip area.

Acknowledgments

The authors would like to thank Dr. Akihiro Musa for valuable discussions.
This research was partially supported by Core Research for Evolutional Science and Technology (CREST), Japan Science and Technology Agency (JST).

References

1) Smith, J.: The Best Way to Achieve Vector-Like Performance, *the 21st Intl. Symposium on Computer Architecture* (1994).
2) Soga, T., Musa, A., Shimomura, Y., Egawa, R., Itakura, K., Takizawa, H., Okabe, K. and Kobayashi, H.: Performance evaluation of NEC SX-9 using real science and engineering applications, *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp.1–12 (2009).
3) Abts, D., Bataineh, A., Scott, S., Faanes, G., Schwarzmeier, J., Lundberg, E., Johnson, T., Bye, M. and Schwoerer, G.: The Cray BlackWidow: a highly scalable vector multiprocessor, *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pp.1–12 (2007).
4) Thakkar, S. and Huff, T.: The Internet Streaming SIMD Extensions, *IntelTechnology Journal*, pp.26–34 (1999).
5) Gschwind, M.: Chip Multiprocessing and the Cell Broadband Engine, *CF '06: Proceedings of the 3rd conference on Computing frontiers*, New York, NY, USA, ACM, pp.1–8 (2006).
6) Intel: Intel Advanced Vector Extensions Programming Reference, http://www.intel.com/.
7) Gebis, J. and Patterson, D.: Embracing and Extending 20th-Century Instruction Set Architectures, *IEEE Computer*, Vol.40, No.4, pp.68–75 (2007).
8) Lefohn, A., Houston, M., Andersson, J., Assarsson, U., Everitt, C., Fatahalian, K., Foley, T., Hensley, J., Lalonde, P. and Luebke, D.: Beyond programmable shading (parts I and II), *SIGGRAPH '09: ACM SIGGRAPH 2009 Courses*, pp.1–312 (2009).
9) Kozyrakis, C.: Vector vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks, *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.283–293 (2002).
10) Christoforos, K.: A Media-Enhanced Vector Architecture for Embedded Memory Systems, Master's thesis, University of California at Berkeley (1999).
11) Rixner, S., Dally, W.J., Khailany, B., Mattson, P.R., Kapasi, U.J. and Owens, J.D.: Register Organization for Media Processing, *HPCA*, pp.375–386 (2000).
12) Hennessy, J.L. and Patterson, D.A.: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003).
13) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An Infrastructure for Com-

puter System Modeling, *Computer*, Vol.35, No.2, pp.59–67 (2002).

14) Bienia, C., Kumar, S., Singh, J.P. and Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications, Technical report, Princeton University (2008).

15) Li, M., Sasanka, R., Adve, S.V., Chen, Y. and Debes, E.: The ALPBench Benchmark Suite for Complex Multimedia Applications, *Proceedings of the IEEE International Workload Characterization Symposium* (2005).

16) Yokoya, Y., Kudoh, Y., Hayasaka, T., Traeff, J., Ritzdorf, H. and Hayashi, Y.: The Compilers and MPI Library for SX-9, Technical report, NEC Corporation (2008).