Design of A Memory Latency Tolerant Processor(SCALT)

Naohiko SHIMIZU* Kazuyuki MIYASAKA** Hiroaki HARAMIISHI**
*Faculty of Eng.,Tokai Univ **Graduate School of Eng.,Tokai Univ.
1117 Kitakaname Hiratuka-shi Kanagawa 259-1292,Japan
TEL: +81-463-58-1211(ex.4084)
*email: nshimizu@keyaki.cc.u-tokai.ac.jp
**email: 0aepm049@keyaki.cc.u-tokai.ac.jp
**email: hiroaki@drive.co.jp

Abstract

In this paper, we introduce a design of a memory latency tolerant processor called SCALT. We present that adding the latency tolerant features on the conventional load/store architecture processor will not bring complexity to the design and the critical path of the processor will not be affected with those features. For the deviation of a latency problem, we have proposed a instruction to check the data arrival existence a buffer. This report describes about the SCALT, that uses buffer check instruction, and its performance evaluation results, obtained analyzing the SMP system through event-driven simulator.
Keywords: Latency Tolerant,Out of Order Execution,Software Prefetch,Decoupled Architecture

1 Introduction

While processor technology have been improving, the clock frequency is also increasing. With the increased requirement for the memory capacity, the DRAM is still using as main memory for its capacity. Though speed of data throughput increases thanks to the new technologies such as SDRAM or RAMBUS, the speed gap between main memory and processor still exists : the memory latency.

Fast cache memory is used to fill this gap. Indeed, performance goes up with cache capacity for small-scale programs, e.g. the larger the cache the higher is the cache hit-rate. But as we know, the overhead of the miss hit is significant. And the overhead will be higher in the clocks count when the clock frequency is higher. In the history, the architects approached this problem with decoupling the access issue with the data usage. We got 1) hit-under-the-miss, 2) miss-under-the-miss, 3) out-of-order conjuncted with many registers. The idea is simple, "issue the loads as fast as possible". But if we need more and more registers to fill the more latency, we must change the processor architecture, which introduces compatibility problems. Or we must use hidden registers which introduces big complexity to the design.

The cache prefetching is thought good idea for these problems. But cache is a software transparent resource, and it is not controllable from the software. The software should still assume that prefetched data may not be in the cache, or someone may rewrite the value. And if we have multiple of processors in a system, it will cause other performance problems.

If we have a reliable buffer which is under the control of a program, the processor design will be more simple. We don't need complex out of order core only to wait the memory latency. Or we don't need hundreds of registers for deep unrolling. But just issue decoupled buffer transfer requests and well-balanced number of registers will fill the latency for the buffer. This is what SCALT will do. The decoupled transfer requests is not restricted to the register's word size. And the number of the on-the-fly requests can be vast. Then in the architecture we can make a super computer's range of memory bandwidth processor. The buffer to the register transfer is still restricted by the processor's load instruction issue rate, but it is not a big problem than the main memory latency.

1.1 Problems related to the memory system

1.1.1 Physical limits at data throughput

Related to the LSI packaging, it have been difficult to assign pins for data I/O in large quantities. Recently this restriction became less serious when BGA packaging was introduced, which allows arranging pins on the face-up side of a chip. Also the upper limit of signal frequency for I/O signals using CMOS technology was comparatively low in the past. But also here transfer rates improved by using technology with impedance matching. As the data throughput is improving, the instruction processing performance is also improving. Ways to achieve higher processing performance are increasing clock frequency or super

scalar architectures. Reaching Gigahertz range these days further increase of clock frequency get technologically difficult. Also when it comes to super scalar architectures complexity diminishes speed improvement. Therefore a need for new innovations in processor architecture arose. VLIW is one attempt. It enables parallel data processing but has no effect on data transfer speed. Thus many VLIW based processors can efficiently process basic data types. But when it comes to large amount of data, memory throughput becomes a major bottleneck.

1.1.2 Memory latency

Another bottleneck is memory latency. If data cannot be provided from memory as fast as the processor requests it, the processor has to wait for memory and the performance goes down. Therefore ways of working around memory latency become a matter for designing high-performance architectures. In recent years multiprocessor systems with shared memory are commonly used for high-performance applications. Having multiple processors accessing the same memory , data respond time becomes even longer. Thus need of latency tolerant architecture became even stronger. Traditional high-performance computers (e.g., vector machine, etc.) can cope with long latency by processing multiple huge data vectors simultaneously. But processing small data vectors and the time needed for proper data vectorization creates an overhead that drops general performance. Furthermore this type of high-performance computers need long development period and is quiet expensive compared to microprocessors.

Beneath the vector machine there are not many architectures that secure high data throughput rates at instruction level. Recently there are other architectures providing SIMD (Single Instruction Multiple Data stream) instructions, but these are restricted to basic data types.

There are two major methods of providing high performance with latency tolerance. One is processing independent data while the processor is waiting for data from memory . A simple example is thread switching: when a thread waits for data from memory another thread which uses available data could be executed in the meantime. Another method to work around memory latency is to prefetch data to the cache so that it is available when requested. This is implemented by analyzing the program flow at compile time and insert prefetch instructions at proper places.

The problem at implementing prefetches is to provide consistency between cache and memory. Imagine you have a store-instruction between a prefetch and the corresponding load-instruction and all three refer to the same memory address. The memory control unit of the processor has to deal with such overlapping cases. Data exchange between memory and processor has to be buffered. The more overlaps a processor can handle simultaneously the more complex it is. Today's normal end-user processors can usually handle up to two or four overlaps. Another problem is cache-thrashing caused by prefetches. If too many prefetches occur, cache capacity or memory association frames are exceeded, thus already cached data is deleted from the cache. If this data is requested again, performance drops because it has to be fetched again from memory. Also, on multiprocessor systems performance drops occur when two or more processors compute data from the same memory area: Given two processors have the same memory area in cache, one processor has to refill the cache every time the other processor makes changes to that area. Because cache has to be consistent with memory, a processor with cache prefetches has to provide special treatment for data-access to memory locations which got prefetched but not yet used. Thus prefetched data which got invalid has to be discarded.

2 The Basics of SCALT

We propose an architecture called SCALT, which does data transfer between memory and processor asynchronously to the instructions requesting it. Furthermore it provides buffers for resources which do not depend on consistency with memory.

The buffer consists of many entries of lines. Each line will be as same length and boundary as the cache line. And we define three instructions to manipulate the buffer entries. We designed the instructions carefully that they will not prevent higher clock rating.

- The instructions for the asynchronous transfer designate one memory location and one buffer entry number. Which means we can use usual address mapping scheme to issue the transfer. The buffer entry number directly designates the location within the buffer. And the processor will issue the memory requests with the entry number information as a tag in a register. The memory system can process the requests as the out of order with attaching the tag. When a request returns the data, processor store the data into the associated entry by referencing the tag. There are two instructions "B.FETCH" and "B.STORE" for the transfers.

- The required number of the buffer is defined by the processing performance and the latencies. For example, assume that we have 10GHz processor and 200nS memory, if the processor issues one load at a clock, the required outstanding buffer should be 16Kbytes, that is $10 \times 10^9 \times sizeof(double) \times 200 \times 10^{-9}$. If we use 64byte lines, only 256 entries are required.

Since these entries do not have to be consistent with memory, the processor does not have to care about possible instructions which alter memory locations

fetched or prefetched from. Also processor design becomes compact as no address checking or special buffering is required. Size and number of buffer-entries is configurable to fit needs of the target application. For small scale applications as in desktop computers the SCALT architecture can compensate low memory bus-capacities. Number and size of buffer entries is software-independent, thus no recompilation is needed to run the well designed software on different versions of a SCALT system.

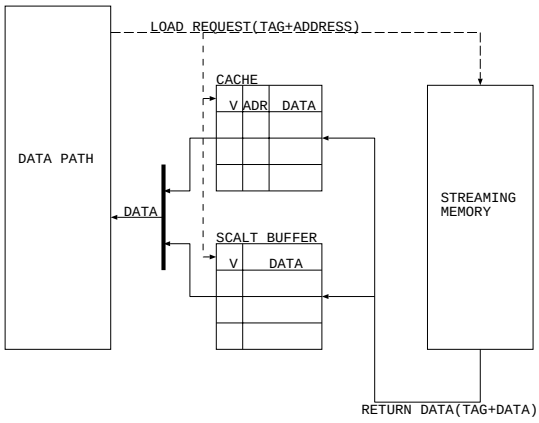

Figure 1: An example load path configuration with SCALT

3 The data flow in SCALT

The figure2 is an example of general RISC processor architectures and SCALT architectures assembly code. Here, constant (a) of DAXPY is in register r1. Corresponding data flow to the SCALT assembly code is showed in the figure3. "B.FETCH" is an

```
                y[i]=y[i]+a*x[i]

RISC              SCALT                SCALT(B.CHECK)

load  r2  #1000   B.FETCH  0 #1000      B.FETCH  0 #1000
load  r3  #2000   B.FETCH  1 #2000      B.FETCH  1 #2000
mult  r2 r1 r2
add   r4 r2 r3      load  r2  #4000       B.CHECK  0
store r4  #2000     load  r3  #4032       B.CHECK  1
                    mult  r2 r1 r2
                    add   r4 r2 r3          load  r2  #4000
                    store  r4  #4032        load  r3  #4032
                          ...               mult  r2 r1 r2
                    B.STORE  1 #2000        add   r4 r2 r3
                                            store  r4  #4032
                                                 ...

                                            B.STORE  1 #2000
```

Figure 2: An example of the scalt assembly code

instruction to fetch the data from the main memory to the specified entry of SCALT buffer. The data size fetched in each time has the same length as the cache line. "B.CHECK" checks the valid bit of the specified buffer entry, and it is an instruction which returns the value. "B.STORE" is an instruction which stores data from buffer entry to the main memory. The figure2 has described only fetch to the 0 and 1 buffer entries number. However, the fetch instructions can issue more buffer entries. Arrival data at the buffer entry is read from the virtual mapped address and loaded into a register, using the usual load instruction, and used to the arithmetic operation. Arithmetic results are stored in the buffer entry, which is mapped to a virtual address, by usual store instruction. When the usual store instruction fill a buffer entry with data, the B.STORE instruction stores specified entry's data to main memory.

A sparse matrix is important for many real applications. There are some implementation methods for this matrix. One of them is a row major compaction which is represented as : A[L[i]]

But the stride prediction cannot predict the stride in this case. Because, L[i] doesn't increase consistently. But the SCALT treats a sparse matrix as below.

The SCALT prefetch the portion of L, and then the SCALT issues prefetch instruction to the each value of A[L[i]].

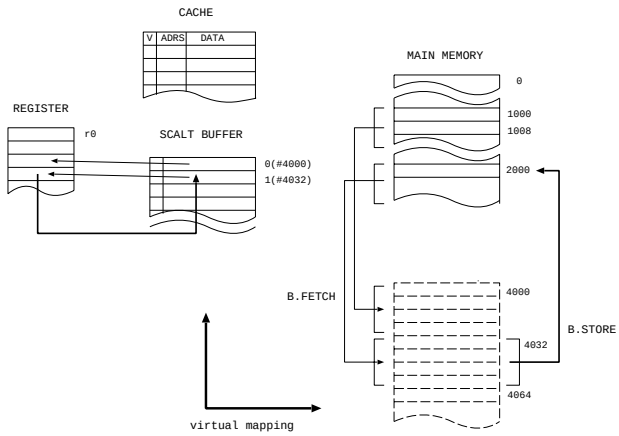

Figure 3: Data flow

4 SCALT Buffer Check Instruction

When data has not arrived to a buffer entry, a processor must waits for the data arrival(i.e. a processor stalls figure4). The program investigates the data arrival using "B.CHECK" instruction, and if the data doesn't arrive, the program investigate the next buffer entry (for example, entry 2, entry 3, etc). The program executes only the data has already arrived at the buffer entry. When the data hasn't arrived, the processing is advanced to the next buffer entry, and it checks data arrival again later. And if the data has arrived, it is executed, and so on.

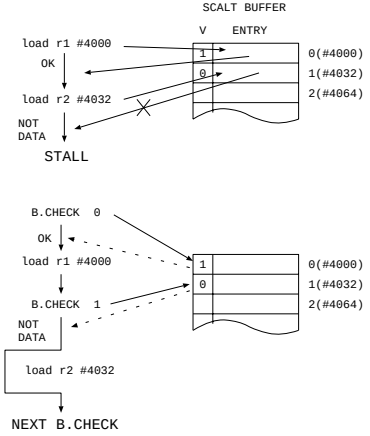

Figure 4: The example of B.CHECK instruction

The figure5 shows an example program flow with SCALT buffer check instruction(B.CHECK). With the deviation of the memory latency, the data in a buffer entry may not arrived yet. In this case, buffer entries of non-arrived data is distinguished by buffer check instruction, and the arithmetic operation is processed only for the ready entries.

Buffer entries with non-arrived data are managed by non-arrived data list and they are checked in every loop until the data arrival. When the data arrives, it is processed and it respective buffer entry is removed from the non-arrived data list. The non-arrived data list is controlled by the software.

The B.CHECK take 1 clock cycle to finish operation. The wait time for the data arrive turns insensitive because the processor doesn't stall.

5 Performance evaluation

In the hardware performance evaluation, we used the event-driven simulator, coded in C language. We used the Livermore Loops, and the address generation patterns was specified by a special compiler for this simulator.

The simulation model was showed in the figure6. The program of Livermore Loops is in INST MEM on the model. Processor fetches one instruction per one cycle from INST MEM, and it performs the instruction. This model is the same as the general processor model. The number of processor, used as a parameter, was from 1 to 64. The connection between a main memory and a processor was crossbar connection. The number of memory banks was made to increase in $2^N (N = 0, 1, 2, ..., 9)$.

When the number of memory banks changed, we investigated the variation of the average rate of bank utilization. The memory banks number is set constant, and when changing the number of processors and the number of buffer entries, it was investigated how many times the loop execution occurred. The parameter used in the simulation was shown in the

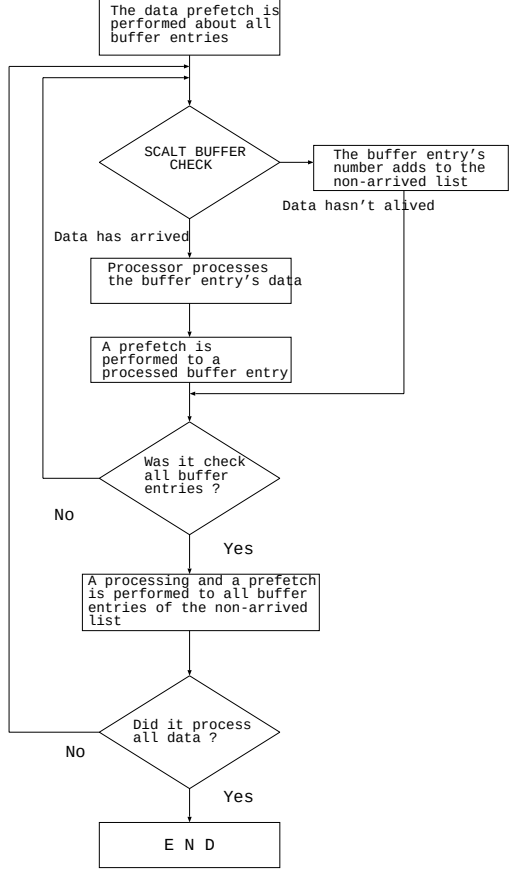

Figure 5: The flow chart using the Buffer Check Instruction

table1.

Table 1: Simulation Parameter

CPU Cycle Time	5nS
Memory banks number	L
Data transmission time between LSI	5nS
SC Request Pitch	10nS
Length of Memory Bank	16B
Length of Buffer Entry	32B
Access Cycle of Memory	250nS
Memory Access Time	M nS
Num of Buffer Entry	N

The Livermore Loops was executed, repeatedly, during a constant time. Memory bank conflicts in the SMP system was caused with frequently, and the memory latency in SCALT fetch was made to change dynamically. SCALT buffer check instruction finishes a processing in one clock. Access to Non-arrived data list are cache access on an actual hardware. In simulation, SCALT accesses non-arrived data list in one clock. SCALT adds buffer entry of non-arrived data to list in one clock, and it removes buffer entry of non-arrived data from list in one clock. These are shown as overhead on low rate of memory utilization in following simulation result.

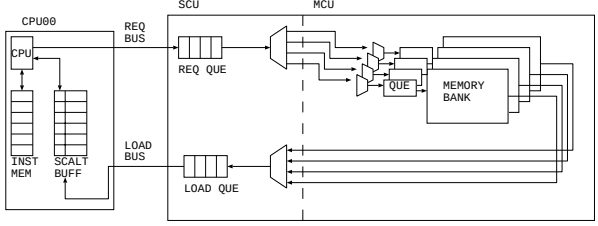

Figure 6: Simulation model

6 Simulation result

The simulation was evaluated to the Livermore Loops kernel 1 and 4. The simulation results was showed from the figure7 to the figure14.

From the figure7 to figure10, we compared the performance results between common RISC processor and the SCALT. The load requests number to the memory system of the RISC processor's cache miss was set to 2 or 4.

The figure7 and the figure8 show that the performance of SCALT was almost proportional at the memory banks. This shows that different performance system can be built changing the memory banks number in the SCALT.

The figure9 and the figure10 show that SCALT can get a high performance with the few number of processors. The SCALT processing saturation was based on the memory system performance limit.(figure10) SCALT performance using buffer check instruction was lower than SCALT performance UN-using it, in case of few processors. In case of few processors, the rate of memory utilization is low. This show that instruction overhead of buffer check instruction causes performance disadvantage in case of little non-arrived data number.

From the figure7 to the figure12, we evaluated the performance running the kernel 1. When a buffer check instruction is used, it can be expected about 20% in the performance improvement. This shown that the effect of buffer check instruction was higher than buffer check instruction overhead.

The figure13(evaluation running the kernel 4) shows the almost same result than the figure11. That is, when there is stride, the effect of a buffer check instruction can be expected.

The figure14 shows that about 30 buffer entries are enough to this kernel 4 simulation.

7 About implementation

The implementation of SCALT buffer is almost the same as that of a cache. As the simulation shows it is effective even with a few kilo bytes. That is, the buffer can be smaller than a cache. Because we do not add a heavy load to a processor critical pass, we consider that SCALT will not hurt the processor's clock frequency.

8 Conclusion

We described our proposal architecture with a software controllable buffer. And we made a performance evaluation for this architecture and shows the effectiveness of our architecture.

Our proposed SCALT got higher performance than the general RISC processor to deviation of the memory latency, and improved the performance almost proportional to the number of the memory banks, and it can get a higher performance with the fewer processors.

However, the effect of buffer check instruction depend on applications or the memory performance as shown in the simulation result, therefore we needs further evaluation.

HDL version SCALT is now in the logic simulation phase. From now on, we will desire to do a simulation on various objects, develop a compiler and library, and make a FPGA version of SCALT running.

References

[1] Harris,Toham,The Scalability of Decoupled Multiprocessors,SHPCC'94,1994

[2] J. L. Baer etc,An effective on-chip preloading scheme to reduce data access penalty,In Proceedings of Supercomputing'91 ,1991

[3] Nakazawa,Nakamura,Pseudo Vector Processor based on Register-Windowed Supersalr Pipline,Supercomputing'92,1992

[4] Todd C.Mowry,Tolerating Latency Through Software-Controlled Data Prefetching

[5] Shimizu,Scalable Latency Tolerant Architecture, IPSJ Sig Notes,Vol.97,No.21,1997

[6] Shimizu,A Study on Performance of a SMP SCALT(SCAlable Latency Tolerant architecture)Instructions,IPSJ Sig Notes,Vol.97,No.76,1997

[7] Mitake,Shimizu,A Hardware feature to overcome the unpredictable memory latency fluctuations by software, Proceeding of the 1998 IEICE General conference

[8] Mitake,Shimizu,Performance evaluation of the processor corresponding to the latency deviation with buffer check instruction,57th IPSJ General conference 1998

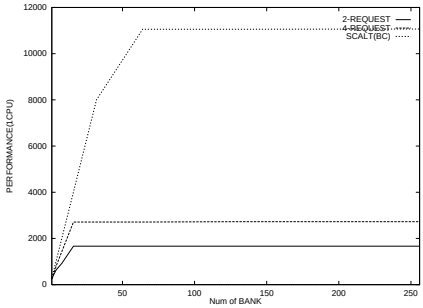

Figure 7: Performance to change of the memory banks number(1cpu)

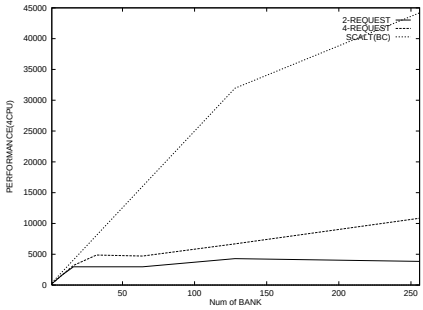

Figure 8: Performance to change of the memory banks number(4cpu)

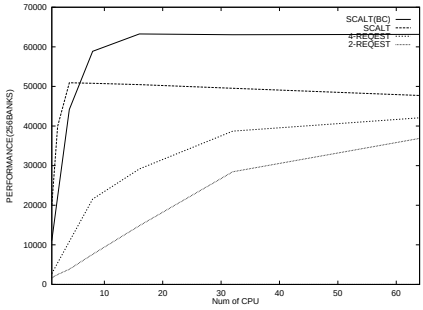

Figure 9: Performance to change of the number of CPU(256BANKS)

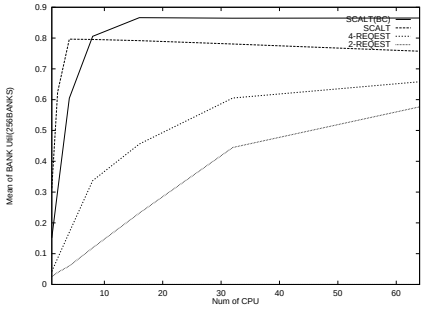

Figure 10: The average of the rate of bank utilization to change of the number of CPU(256BANKS)

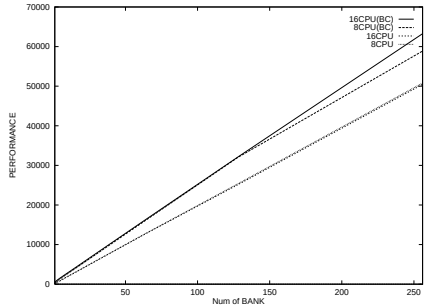

Figure 11: Performance to change of the memory banks number (Kernel1-8,16cpu)

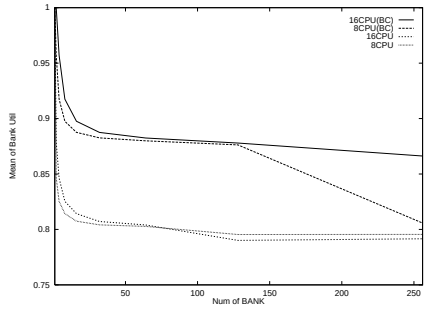

Figure 12: The average of the rate of bank utilization to change of the memory banks number(kernel1-8,16cpu)

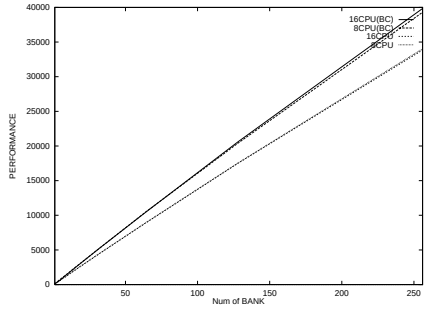

Figure 13: Performance to change of the number of banks(Kernel4 -8,16cpu)

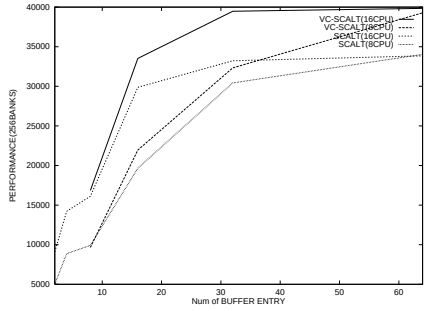

Figure 14: Performance to change of the number of Buffer entry(Kernel4-8,16cpu)