Hyperscalar Processor Architecture *

Kazuaki Murakami Hiroshi Miyajima Yasuhiko Saitoh
Tetsuo Hironaka Satoru Shirakawa

Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyushu University
6–1 Kasuga-Koen
Kasuga, Fukuoka 816 JAPAN
{murakami, miyajima}@is.kyushu-u.ac.jp

Abstract

This paper describes a novel processor architecture, called *hyperscalar processor architecture*, which encompasses the advantages of superscalar, VLIW, and vector processor architectures and exculdes their disadvantages. In brief, hyperscalar is a processor; i) whose instruction size and instruction-fetch bandwidth are the same as those of superscalar, ii) whose datapath is as large as that of VLIW, iii) which provides every independent functional unit with one or more compiler-visible registers, called *instruction registers*, and iv) which allows the program itself to load the instruction registers with instructions fetched from the memory and to execute them as a subroutine. As compiler techniques for creating an object code placed in the instruction registers, this paper proposes *pseudo vector processing* and *software pipelining*, and further discusses several issues on applying software pipeling to hyperscalar processors. This paper evaluates the performance attainable in hyperscalar processors, and then concludes that hyperscalar processors can outperform conventional superscalar, VLIW, and vector processors in terms of cost/performance.

1 Introduction

Hyperscalar processor architecture [8] is a "post-superscalar" architecture, which encompasses the advantages of (RISC-type) superscalar, VLIW (Very Long In-

*This is an internal technical report describing research done at the Department of Information Systems of Kyushu University.

struction Word), and vector processor architectures and exculdes their disadvantages. The features of hyperscalar processors are summarized as follows.

- The control portion of hyperscalar processors, especially the instruction fetch and issue logic, is the same as that of superscalar processors in terms of the hardware size. The instruction size might be 32 or 64 bits as in today's superscalar processors. The number of instructions to be fetched and issued at a time should be two or three as in early superscalar processors, so that the instruction fetch and issue logic should not become more complex.

- On the other hand, the datapath of hyperscalar processors can be as large as that of VLIW processors in term of the number of independent functional units. It might be more than 10 if the transister budget allows.

- Obviously, there is a gap between the number of instructions to be issued at a time and the number of independent functional units; but, it does not become an issue. Hyperscalar processors provide every independent functional unit with one or more compiler-visible registers, called *instruction registers*. The instruction registers are placed immediately before the pipe stage where instructions are dispatched to the correponding functional units. Hyperscalar architecture allows program itself to load these instruction registers with instructions and to execute the loaded instructions.

- A sequence of instructions stored in instruction registers looks like a VLIW code and is executed in the same fasion as VLIW. By compiling loops into software-pipelined VLIW codes and loading each VLIW code before the execution of the correponding loop, hyperscalar processors can improve the loop performance as high as VLIW. At the same time, the hyperscalar processors do not increase the object-code size as large as VLIW.

Target applications of hyperscalar processors may range broadly;

- from non-scientific applications with low instruction-level parallelism, which traditional superscalar processors are good at,

- to scientific applications high instruction-level parallelism, which traditional VLIW processors are good at, and furthermore

- to scientific applications with both high instruction-level and parallelism and data parallelism, where traditional vector processors do their best.

This paper describes the principle of operations of hyperscalar processors and evaluates the performance attainable in them. The rest of the paper is organized as follows. Section 2 summarizes the advantages and disadvantages of conventional processor architectures which exploit instruction-level parallelism.

Section 3 describes the principle of operations of hyperscalar processors in detail. Section 4 presents how to utilize the feature of hyperscalar processors, and then discusses several compiler issues. Section 5 reports some preliminary results of performance evaluation. Section 6 offers a concluding remark.

2 Previous Work

So far, there are the following processor architectures which exploit instruction-level parallelism.

1. Pipelined (and also, superpipelined) processor
2. Superscalar processor
3. VLIW processor
4. Vector processor

We briefly discuss the advantages and disadvantages of these conventional processor architectures.

2.1 Pipelined Processor Architecture

Instruction pipelining is a basic technique for exploiting instruction-level parallelism. The parallelism exploited is called *temporal parallelsim*. Instruction pipelining is not only used in pipelined processors, but also used in the other processor architectures. If we try to improve the processor performance by means of instruction pipelining only, however, there is no way other than increasing the clock rate.

Superpipelined processor architecture is exactly the architecture which tries to enhance the perfomance by dividing pipe stages into smaller ones and by increasing the clock rate. A superpipelined processor with the clock rate m times higher than the original pipelined processor is said to provide the *superpipeline degree* of m [4].

A disadvantage of the superpipelined approach is that its performance improvement depends heavily on the growth of clock rate. The annual growth of clock rate is approximately 20% and lower than the annual growth of transister counts per die, which is motivating the superscalar approach.

2.2 Superscalar Processor Architecture

Superscalar processor architecture exploits *spatial parallelism* as well as temporal parallelism. The number of instructions issued per clock cycle is called *superscalar degree* [4]. The performance of a superpipelined processor with the

superpipeline degree of m is theoretically equal to that of a superscalar processor with the superscalar degree n, if $m = n$.

Most of superscalar processors detect and resolve pipeline hazards at run time, so as to ensure the compatibility and portability of object code. The superscalar degree of commercial superscalar processors is therefore 2–4 and lower than that of VLIW processors which resolve pipeline hazards at compile time. An advantage of superscalar processors against VLIW processors is that superscalar processors can ensure the compatibility and portability of object code. A disadvantage of superscalar processors against VLIW processors, on the other hand, is that the machine parallelism (i.e., superscalar degree) is poor for some scientific applications with high instruction-level parallelism.

2.3 VLIW Processor Architecture

VLIW processor is a derivative from superscalar processors in the sense that VLIW processors rely on compilers with respect to the pipeline-hazard resolution. The advantage of superscalar processors against VLIW processors now becomes a disadvantage of VLIW processors against superscalar processors, and vice versa.

Another disadvantage of VLIW processors is that the object code size might increase rapidly by inserting 'no-op's at unused operation fields in a very long instruction word.

2.4 Vector Processor Architecture

Vector processor architecture is the *de facto* most powerful processor architecture, especially suitable for computation-intensive scientific application with plenty of data parallelism. The vector performance depends mainly on the powerful vector instructions, the large, fast vector registers, and the vectorizing compilers. The limited accessibility to vector registers (i.e., only vector instructions can access vector register for computation, but scalar instructions cannot) , however, bounds the potential of vector registers. If scalar instructions are allowed to access vector registers for computation, a more flexible loop-transformation technique than vectorization, i.e., *software pipelining* can be used. This is a disadvantage of vector processors against scalar or VLIW processors with a large register file, such as WM computer [11], Cydra5 [10].

3 Hyperscalar Processor Architecture

Hyperscalar processor architecture tries to include the above-mentioned advantages and exclude the disadvantages of superscalar, VLIW, and vector processor architectures.

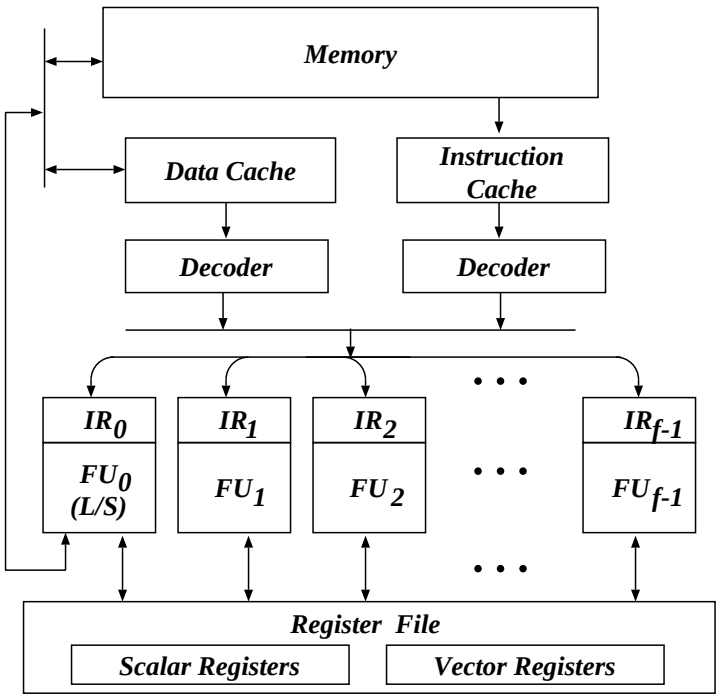

Figure 1: Basic Organization of Hyperscalar Processors

3.1 Instruction Registers

Figure 1 shows a basic oraganization of hyperscalar processors. The basic organization of hyperscalar processors is quite similar to that of superscalar processors. The instruction size and the instruction bandwidth are comparable to those of superscalar processors. For example, the instruction size might be 32 or 64 bits as in today's superscalar processors. The instruction-fetch bandwidth and the superscalar degree should be two or three as in early superscalar processors, so that the instruction fetch and issue logic should not become more complex. These features, or design constraints, relieves the disadvantages of VLIW described in Section 2.3, such as the code-size problem and the cache underutilization problem.

Fundamental differences between hyperscalar and superscalar are that hyperscalar processors might provide more functional units than superscalar processors do and that hyperscalar processors provide an additional set of compiler-visible registers, called *instruction registers* (IRs). The number of independent functional units of hyperscalar processors can be as large as that of VLIW processors. It might be more than 10 if the transister budget allows. For each of

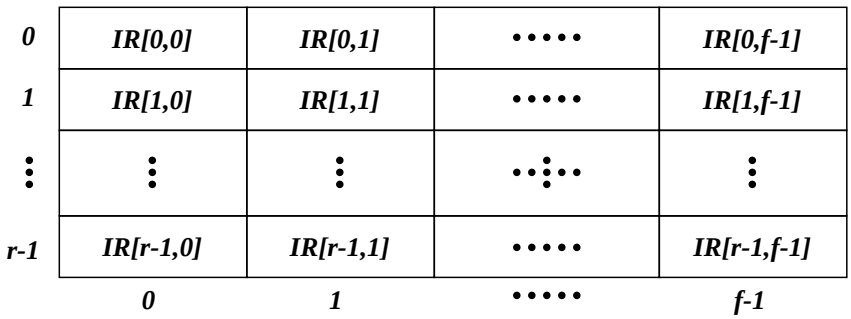

Figure 2: Address Space of Instruction Registers

these independent functional units, one or more instruction registers are provided. Assuming that the number of functional units is f and the number of instruction registers per functional unit is r, $f \times r$ instruction registers exist in a hyperscalar processor in total.

Figure 2 shows the two-dimensional address space of $f \times r$ instruction registers. Every instruction register is denoted as $IR[i,j](0 \leq i \leq r-1, 0 \leq j \leq f-1)$. Every row and column are denoted as $IR^i = \{IR[i,0], IR[i,1], \ldots, IR[r-1,j]\}$ and $IR_j = \{IR[0,j], IR[1,j], \ldots, IR[r-1,j]\}$, respectively. Each $IR^i(0 \leq i \leq r-1)$ corresponds to one VLIW instruction consisting of f operation fields, and then the whole rIR^is can store an VLIW program consisting of r VLIW instructions.

Figure 3 shows the structure of IR_j. Each IR_j stores at most r decoded instructions, each of which is executed at the corresponding function unit FU_j. The following two approaches are possible for loading instruction registers with decoded instructions.

- Loading IRs exclusively: A special instruction, called `load instruction register` (`LIR`), is defined and used for loading an instruction register with a decoded instruction. A `LIR` instruction fetches an instruction from the specified memory location, decodes it, and places it into the destination instruction register $IR[i,j]$.

- Loading IRs concurrently: An instruction is placed into the appropriate instruction register $IR[i,j]$ at the same time it is dispatched to the appropriate functional unit FU_j in a normal fashion. Instructions to be stored into IRs might be marked by a compiler.

At least, the former method is sufficient to be implemented.

The instruction decoder used for loading IRs by means of `LIR` instructions is either:

- a special decoder provided for `LIR` instructions, or

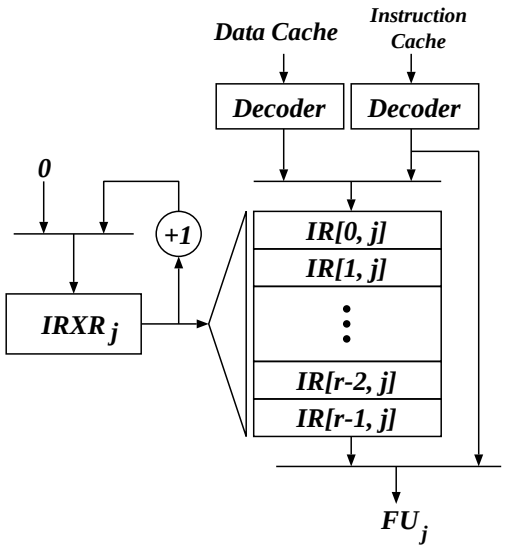

Figure 3: Structure of IR_j

- a normal decoder used in a normal execution fashion.

Figures 1 and 3 show an example of the implementation which provides a special instruction decoder for loading IRs.

3.2 Principle of Operations

Hyperscalar processors provide the following two operation modes.

- *Normal mode*: As normal superscalar processors do, a hyperscalar processor fetches two or three instructions from the instruction cache, decodes them, and then dispatches them to functional units. The pipeline flow of instruction execution is shown in Figure 4. Instruction registers are not used for issuing instructions to functional units.

- *Turbo mode*: Instructions are fetched from not the instruction cache but the instruction registers IRs. Since the instructions in IRs are already decoded, they can be dispatched to functional units immediately after they are fetched from IRs. The instructions fetched from the instruction register IR_j are issued to the functional unit FU_j. The pipeline flow of instruction execution is shown in Figure 5.

As the instruction-set architecture, in addition to the `LIR` instruction, at least two instructions for transferring the processor between two operation modes should be defined.

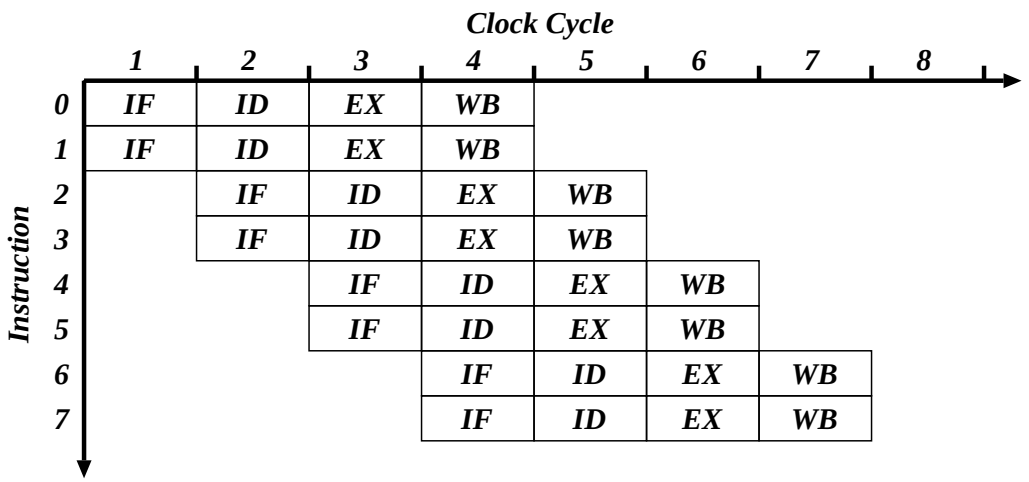

Figure 4: Instruction Pipeline in Normal Mode

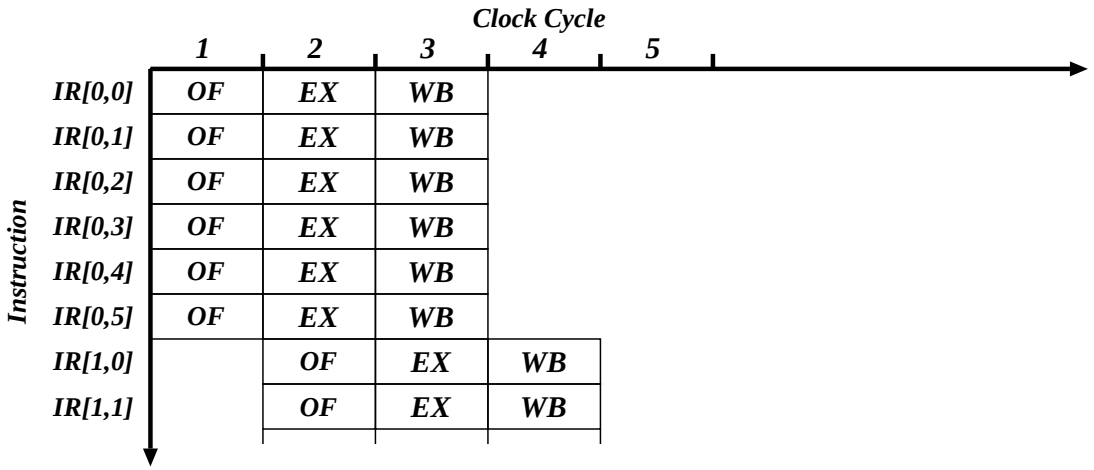

Figure 5: Instruction Pipeline in Turbo Mode

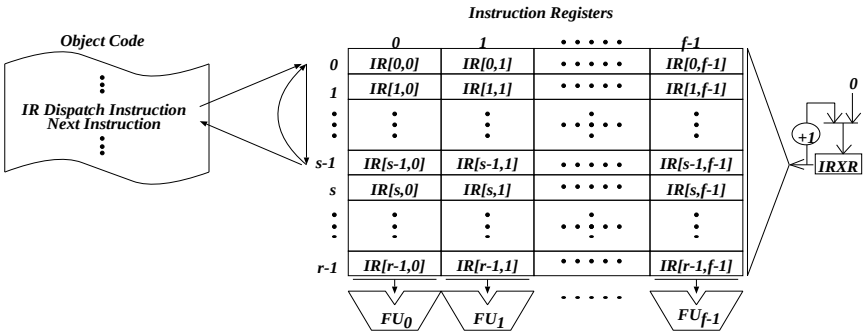

Figure 6: Principle of Operations of Hyperscalar Processor

A method for utilizing the instruction registers in hyperscalar processors is as follows.

1. For program segments where the instruction-level parallelism is low, the hyperscalar processor operates in the normal mode; i.e., as normal superscalar processors do.

2. Before entering a program segment, such as loops, where the instruction-level parallelism is high, the program itself loads the instruction registers IRs with a sequence of instructions corresponding to the program segment. `LIR` instructions might be used for loading IRs.

3. After completing the loading of IRs, the program transfers the processors into the turbo mode, and then the processor begins to fetch instructions from IRs. The IR index register $IRXR$ is reset to some value (e.g., 0).

4. In the turbo mode, the hyperscalar processor seems to execute a VLIW code of s VLIW instructions stored in $IR^i = \{IR[i,0], IR[i,1], \ldots, IR[i,f-1]\} (0 \leq i \leq s-1 \leq r-1)$. The IR index register $IRXR$ works as the program counter. Every clock cycle, the processor fetch a VLIW instruction from IR^i specified by the $IRXR$ (i.e., from IR^{IRXR}), dispatches each instruction fetched from $IR[IRXR, j]$ to the corresponding functional unit FU_j, and then increments the $IRXR$.

5. Once the processor enters the turbo mode, it continues until an exit condition is satisfied. Any VLIW instruction may include an instruction for quiting the turbo mode. This instruction checks whether or not the exit condition is satisfied, and then takes one of the following actions.

   - If the exit condition is satisfied, the processor transfers to the normal mode. The program resumes its execution from some specified

instruction or the instruction succeeding the instruction which transferred the processor into the turbo mode.

- Otherwise, the processor continues the turbo mode. The next VLIW instruction to be fetched is either some specified instruction of the succeeding instruction.

The above-mentioned method for utilizing the instruction registers corresponds to a sequence of executing a procedure or subroutine, as follows:

- An instruction transferring the processor into the turbo mode corresponds to a "procedure-call" instruction.
- A sequence of VLIW instructions stored in IRs corresponds to a "procedure".
- An instruction transferring the processor into the normal mode corresponds to a "procedure-return" instruction.

The numbers of instructions issued in the normal and turbo modes are referred to as *superscalar degree* and *hyperscalar degree*, respectively.

3.3 Vector Registers

If the transister budget allows, a hyperscalar processor might provide vector registers. Althogh the hyperscalar architecture itself does not require vector registers, vector registers (or, a large register file) are effective for increasing the effect of instruction registers (for the reason, see Section 4.3) . Providing vector registers does not necessarily mean providing vector instructions as the instruction-set architecture. As pointed out in [1], even though vector instructions are not provided, a sequence of scalar instructions (or, VLIW instructions) can operate on vector data stored in vector registers in a pseudo-vector-processing or software-pipelined manner (see Section 4).

4 Hyperscalar Compiler

Compilers for hyperscalar processors should create object codes stored in IRs. Candidates for such codes are loop portions where the instruction-level parallelism is high. The following two methods are possible for transforming loops into object codes stored in IRs:

- **Pseudo vector processing**
- **Software pipelining**

4.1 Pseudo Vector Processing

We refer to simulating a sequence of vector instructions by means of scalar or VLIW instructions as *pseudo vector processing* [9]. Let's use the following simple example for explaining the concept of pseudo vector processing in hyperscalar processors.

```
do I=0, 31
A(I) = B(I) + C(I)
do end
```

Conventional vector processors execute this loop by means of the following sequence of vector instructions.

```
VLD B(I) ⟶ VR1
VLD C(I) ⟶ VR2
VADD VR1 + VR2 ⟶ VR3
VST VR3 ⟶ A(I)
```

Now, assume a hyperscalar processor which provides vector registers and the following five functional units:

- FU_0, FU_1: load units
- FU_2: add unit
- FU_3: store unit
- FU_4: branch unit

This hyperscalar processor can simulate the above vector code by means of the following VLIW code stored in IRs (e.g., IR^0), as shown in Figure 7.

```
IR[0,0] : LD B(SR1) ⟶ VR1
IR[0,1] : LD C(SR2) ⟶ VR2
IR[0,2] : ADD VR1 + VR2 ⟶ VR3
IR[0,3] : ST VR3 ⟶ A(SR3)
IR[0,4] : if VRXR(VR1)=31 then return to normal mode
          else branch to 0
```

Here, we assumed that load/store instructions (`LD/ST`) provide some scale-indexed addressing mode with post-increment for index registers (SR1, SR2, and SR3). And, VRXR(VR1) stands for the vector-register index-register for VR1. Note that, as shown in Figure 7, some array elements are processed in the normal mode; i.e.,

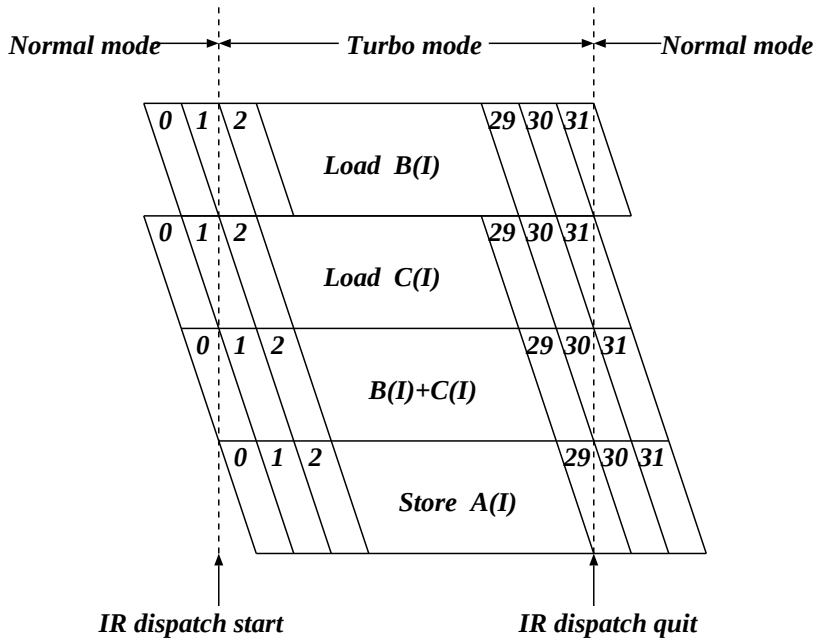

Figure 7: Pseudo Vector Processing

- loading of B(0), C(0), B(1), and C(1),
- addition of B(0)+C(0) before entering the turbo mode,
- addition of B(31)+C(31) after exiting the turbo mode, and
- storing of A(30) and A(31).

4.2 Software Pipelining

Software pipelining is a loop transformation technique which exploits instruction-level parallelism across consecutive iterations of a loop [5]. Pseudo vector processing described in Section 4.1 is also a kind of software pipelining. Pseudo vectorization can apply to vectorizable loops only, but software pipelining can also apply to non-vectorizable loops.

Let's use the following simple example for explaining the concept of software pipelining in hyperscalar processors.

```
do I=0, 30
A(I+1) = A(I)+B(I)
do end
```

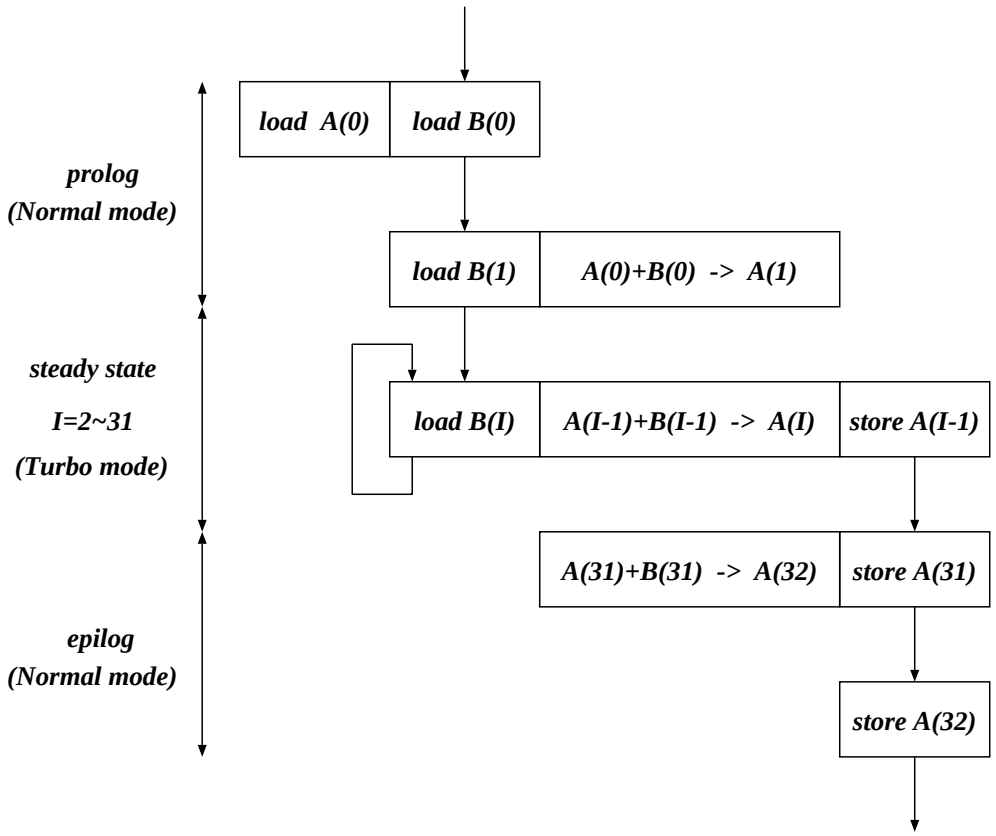

Figure 8: Software Pipelining

Since the above loop includes the first-order recurrence, it cannot be vectorized for vector processors without a special vector instruction. An object code before software pipelining is performed is as follows.

```
        LD A(0) ⟶ SR13
L1 :    LD B(SR1) ⟶ SR11
        ADD SR13 + SR11 ⟶ SR13
        ST SR13 ⟶ A(SR3)
        if SR1<30 then goto L1
```

Figure 8 shows the object code after software pipelining is applied to the above code. Here, we made the same assumption on the functional units and the load/store instructions as in Section 4.1. The software-pipelined loop consists of the following three parts.

1. **Prolog**: After loading IRs with the VLIW code corresponding to the steady-state part of the software-pipelined loop (see below), the processor executes the following prolog code in the normal mode.

```
LD    A(0) ⟶ SR13
LD    B(SR1) ⟶ VR1
ADD   SR13 + VR1 ⟶ SR13
LD    B(SR1) ⟶ VR1
```

2. **Steady state**: The processor enters the turbo mode, and then executes the following VLIW code in IRs until the exit condition is satisified.

```
IR[0,0] : LD B(SR1) ⟶ VR1
IR[0,1] : no-op
IR[0,2] : ADD SR13 + VR1 ⟶ SR13
IR[0,3] : ST SR13 ⟶ A(SR3)
IR[0,4] : if SR1=30 then return to normal mode
          else branch to 0
```

3. **Epilog**: The processor exits the turbo mode, and then executes the following epilog code in the normal mode.

```
ST    SR13 ⟶ A(SR3)
ADD   SR13 + VR1 ⟶ SR13
ST    SR13 ⟶ A(SR3)
```

4.3 Hyperscalar Constraints on Software Pipelining

In general, the following two constraints must be satisfied when software pipelining is performed.

- Data and control dependences between instructions should be preserved even after software pipelining is performed.
- A fixed amount of processor resources only can be used. The processor resources include functional units and registers.

For hyperscalar processors, when software pipelining is applied to the VLIW code stored in IRs, the resource constraints include the instruction registers.

Modulo scheduling [5], one of software pipelining algorithms, aims at minimizing the iteration initiation interval (III) of a target loop while satisfying the above constraints. If the obtained III is not smaller than the original III, then software pipelining for the loop failed. For hyperscalar processors, even when the obtained III is smaller than the original III, software pipelining for the loop failed if the obtained III is greater than the number of instruction registers IR^i (i.e., r). In this case, the following two approaches are possible.

- Decompose the original loop into several loops, and then apply software pipelining to each loop so that the software-pipelined loop could be stored in IRs.
- Abandon storing the original software-pipelined loop in IR, and then execute it in the normal mode.

Software-pipelined loops consist of three parts: prolog, steady state, and epilog. For superscalar and VLIW processors, the machine parallelism available across these parts is constant and equal to the superscalar degree, and therefore we can say that the software-pipelined loop execution time should be smaller than the original loop execution time if the obtained III is smaller than the original III.

For hyperscalar processors, on the other hand, the machine parallelism available across the three parts of a software-pipelined loop is not constant, as follows:

- prolog: it is equal to the superscalar degree,
- steady state: it is equal to the hyperscalar degree, and
- epilog: it is equal to the superscalar degree.

Usually, the superscalar degree is not larger than the hyperscalar degree. For hyperscalar processors, even when software pipelining for a loop succeeded, we cannot immediately say that the software-pipelined loop execution time should be smaller than the original loop execution time. Furthermore, hyperscalar processors require an additional time for loading IRs with VLIW code corresponding the steady state of the software-pipelined loop. Hyperscalar compilers should estimate the loop execution time for the following three cases and choose one out of them.

- Execute the original (not software-pipelined) loop in the normal mode.
- Apply software pipelining to the loop, and execute the entire software-pipelined loop in the normal mode.
- Apply software pipelining to the loop, and execute its prolog and epilog parts in the normal mode and its steady-state part in the turbo mode, respectively.

Note that the loop execution time is determied as a function of the number of iterations, and therefore the program itself should estimate the execution time for the above three cases and choose one out of them if the number of iterations is not known at compile time.

As described earlier, the objective of software pipelining is to minimize the iteration initiation interval (III). Impediments to the objective include inter-instruction dependences (i.e., data and control dependences) and resource conflicts. Among them, anti-dependences which are caused by registers allocated

to data with long lifetime produce a detrimental effect on the III. There are the following conventional methods for relieving the effect of anti-dependences.

- **Modulo variable expansion** [5]: Anti-dependences at issue occur when an iteration writes into a register before the preceeding iteration does not read the same register. If we allocate different registers, rather than a single register, to different iterations, we can resolve these anti-dependences. For this to be performed, compilers usually unroll the target loop several times, allocate different registers to different iterations, and then apply software pipelining to the unrolled loop. This method might cause another problem of increasing the code size.

- **Vector registers**: The modulo variable expansion described above allocates different "scalar" registers to different iterations. Assuming the existence of vector registers, it corresponds to allocating different "entries" of a vector register to different iterations. If we replace the scalar register causing anti-dependences with a vector register, we can resolve the anti-dependences. This method might require an additional hardware cost for vector registers.

The modulo variable expansion is not always effective for hyperscalar processors, since the amount of instruction registers is limited and the modulo variable expansion might increase the code size. On the other hand, introducing vector registers into hyperscalar processors is not necessarily cost-effective. We therefore have proposed another method, called *stage balancing* [2], which is suitable for hyperscalar processors.

Figure 9 shows the concept of stage balancing. In brief, stage balancing is to keep all the register lifetimes shorter than a given III by spliting every longer lifetime into several shorter lifetime. This is performed by inserting several `move register` instructions at appropriate points during a long register lifetime for transmitting data from one register to another register continuously.

5 Performance Evaluation

We evaluate the performance attainable in hyperscalar processors.

5.1 Evaluation Models

We compare the performance of a hyperscalar processor prototype under development [7] with that of the following counterparts:

- pipelined processor,

- superscalar processor,

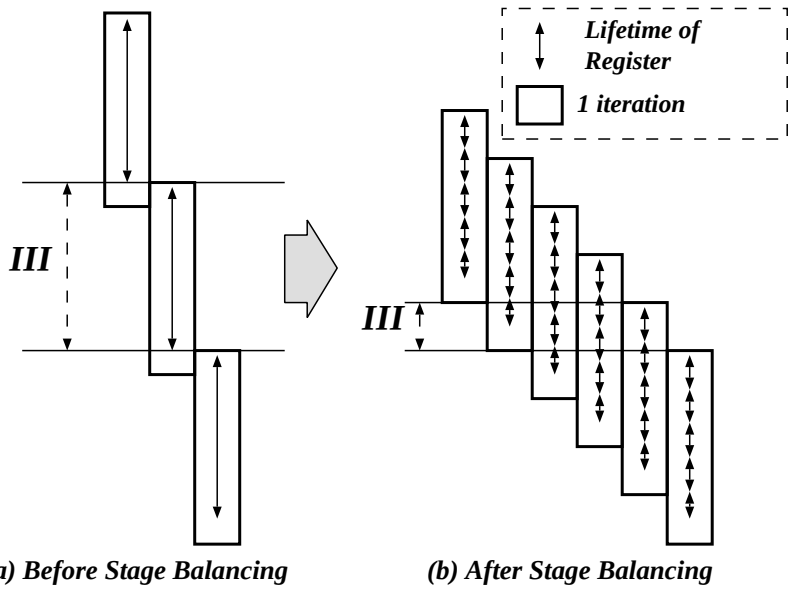

Figure 9: Stage Balancing

- VLIW processor, and
- vector processor.

For the hyperscalar processor prototype, we compare the performance of the following two methods for relieving the detrimental effect of anti-dependences on software pipelining:

- stage balancing, and
- vector registers.

We have developed the following seven evaluation models. Table 1 summarizes the specifications of these evaluation models.

1. *SP* : A conventional pipelined RISC-type scalar processor with the superscalar degree of one.
2. *SSP* : A conventional pipelined RISC-type superscalar processor with the superscalar degree of two.
3. *HSP* : The hyperscalar processor ptototype with the superscalar degree of two and the hyperscalar degree of five.
   (a) *HSP+SB* : Stage balancing is employed.

(b) *HSP+VR* : A set of vector registers is provided.

4. *VLIW* : A conventional pipelined VLIW processor with the superscalar degree of five.

5. *VP* : A conventional vector processor with the computational throughput of 2FLOPC (floating-point operations per clock cycle).

6. *VP+M+F* : An extended model of the model *VP* , which introduces a multithreading facility at vector-instruction level and allows vector registers to work as ring FIFO buffers [1].

The datapaths of the evaluation models *SP* , *SSP* , *HSP* , and *VLIW* are identical and based on that of the hyperscalar processor prototype, except that the model *HSP+VR* provides vector registers. The datapaths of the evaluation models *VP* and *VP+M+F* are identical and based on that of the model $Base_2^2$ in [1]. For the evaluation models *HSP* , we assumed that the number of instruction registers IR^i is infinite.

We also made the following assumptions on the memory systems for all the evaluation models.

- The instruction cache is a perfect cache; i.e., the cache hit rate is 100%.
- Whether or not a data cache is provided, all load/store accesses are performed on the main memory. The main memory is conflict-free, and every memory accesses completes in a constant time.
- The memory access latency is three clock cycles, and the load/store latency including another clock cycle for address generation is four clock cycle.

5.2 Evaluation Results

As the benchmark programs, we used the first 14 kernels of 24 Livermore Fortran Kernels (LFK). For the evaluation models *SP* , *SSP* , *HSP* , and *VLIW* , we tried to apply software pipelining to all the LFKs; for the *VP* and *VP+M+F* models, we also tried to vectorize all the LFKs.

Figure 10 shows the performance of all the evaluation models in terms of FLOPC. Figure 11 shows the speedup of every evaluation model over the *HSP+SB* model. From Figure 10, we recognize that every model ranks as follows, in descending order of the harmonic means of thier FLOPC rates:

1. *VP+M+F* : 0.4888FLOPC
2. *HSP+VR* : 0.4578FLOPC
3. *VLIW* : 0.4384FLOPC

Table 1: Specifications of Evaluation Models

Model			*SP*	*SSP*	*VLIW*	*HSP+SB*	*HSP+VR*	*VP*	*VP+M*
Machine Parallelism	Superscalar Degree		1	2	5	2		2	
	Hyperscalar Degree		—	—	—	5			
Peak FLOPC			1	2					
Functional Units	Integer ALU (IALU)	Number of Units	1					—	
		Issue Latency	1						
		Result Latency	2						
		Throughput	1 op/cycle						
	Floating-Point ALU (FALU)	Number of Units	1						
		Issue Latency	1						
		Result Latency	4						
		Throughput	1 FLOPC						
	Floating-Point Multiplier (FMUL)	Number of Units	1						
		Issue Latency	1						
		Result Latency	4						
		Throughput	1 FLOPC						
	Load/ Store Unit (L/S)	Number of Units	2						
		Issue Latency	1						
		Result Latency	4						
		Bandwidth/Unit	8 byte/cycle						
Registers	Integer Register	Number of Registers	32						
	FP Register	Number of Registers	32						
	Vector Register (Data FIFO)	Number of Registers	—–				16		
		Number of Elements	—–				32		

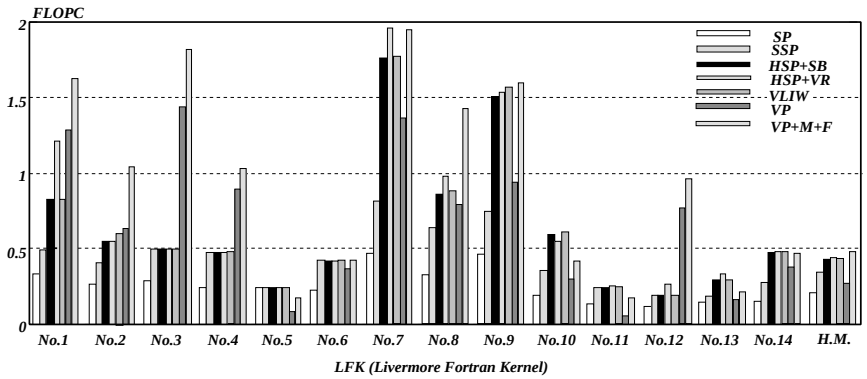

Figure 10: FLOPC Rating

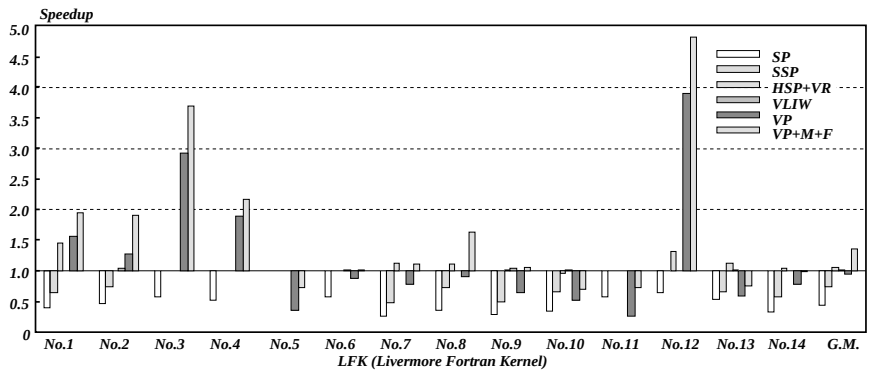

Figure 11: Speedups over *HSP+SB*

Table 2: Fraction of Turbo Mode Time in Total Execution Time

LFK	(%)	LFK	(%)
1	99.00	8	96.46
2	42.31	9	83.91
3	98.88	10	88.41
4	94.32	11	99.10
5	99.63	12	99.32
6	79.47	13	84.88
7	98.51	14	98.94

4. *HSP+SB* : 0.4292FLOPC
5. *SSP* : 0.3527FLOPC
6. *VP* : 0.2776FLOPC
7. *SP* : 0.2221FLOPC

We obtain the following observations.

- The *VP+M+F* model is the fastest among all the models, and it is 0.71 (min: LFK10) – 1.36 (geometric mean) – 4.82 (max: LFK12) times faster than the *HSP+SB* model. Among 14 LFKs, the *VP+M+F* model outperforms the *HSP+SB* model in only 9 LFKs (LFK1, 2, 3, 4, 6, 7, 8, 9, and 12), and the speedups are less than two, except for LFK3, 4, and 12. On the other hand, the conventional vector-processor model *VP* is generally slower than the *HSP+SB* model, and it is 0.26 (min: LFK11) – 0.92 (geometric mean) – 3.88 (max: LFK12) times faster than the *HSP+SB* model.

- The *HSP+SB* model is as fast as the *VLIW* model, since the *VLIW* model outperforms the *HSP+SB* model by only 1.00 (min: LFK5 and 12) – 1.01 (geometric mean) – 1.10 (max: LFK2) times. Table 2 summarizes how much the hyperscalar processor ran in the turbo mode. In the turbo mode, the machine parallelism of the *HSP+SB* model (i.e., the hyperscalar degree) is equal to that of the *VLIW* model, but in the normal mode this is not the case. The difference of machine parallelism between the *HSP+SB* model in the normal mode and the *VLIW* model causes the performance difference between two models. In this evaluation, we assumed the perfect instruction cache for both models. In the real situation, however, this is not true, and the hit rate of the VLIW instruction cache should be lower since the VLIW code size is usually larger.

- The *HSP+SB* model improves the performance of the *SSP* model, just by adding instruction registers and providing additional ports in the register file. The *HSP+SB* model is 0.94 (min: LFK6) – 1.32 (geometric mean) – 1.77 (max: LFK14) times faster than the *SSP* model. Among 14 LFKs, the *HSP+SB* is slightly slower than the *SSP* model in 6 LFKs (LFK3, 4, 5, 6, 11, and 12) because of the overhead for loading instruction registers. If the *HSP+SB* model did not execute these LFKs in the turbo mode, it would achieve the same performance as the *SSP* model.

- The *HSP+SB* model is as fast as the *HSP+VR* model which provides vector registers, since the *HSP+VR* model outperforms the *HSP+SB* model by only 0.98 (min: LFK10) – 1.08 (geometric mean) – 1.48 (max: LFK1) times. The effect of stage balancing is comparable to that of vector registers which incurs a large hardware cost.

6 Conclusions

We have described the hyperscalar processor architecture and the hyperscalar compilers. We have also evaluated the performance attainable in a hyperscalar processor prototype under development.

As a result of the preliminary performance evaluation, the hyperscalar processor prototype is found to be:

- 1.32 times faster in geometric mean than a conventional RISC-type superscalar processor with almost the same datapath,
- 1.09 times faster in geometric mean than a conventional vector processor, and
- 0.99 times faster in geometric mean than a conventional VLIW processor which requires more instruction-fetch bandwidth.

From the above, we can conclude that hyperscalar processors could outperform conventional superscalar, VLIW, and vector processors in terms of cost/performance.

References

[1] Hashimoto, T., Murakami, K., Hironaka, T., and Yasuura, H., "A Microvectorprocessor Architecture — Performance Modeling and Benchmarking —," *Proc. 1993 Int'l. Conf. on Supercomputing*, pp.308–317, July 1993.

[2] Hironaka, T., Saitoh, Y., and Murakami, K., "Hyperscalar Processor Architecture — Performance of Software Pipelining — (in Japanese)," *IEICE Technical Report*, VLD93–89, Dec. 1993.

[3] Hironaka, T., Saitoh, Y., Miyajima, H. and Murakami, K., "Hyperscalar Processor Architecture — Design and Performance Evaluation of the Prototype Processor —(in Japanese)," *Proc. 1994 Joint Symp. on Parallel Processing*, pp.9–16., May 1994.

[4] Jouppi, N. P., "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance," *IEEE Trans. Comput.*, vol.37, no.12, pp.1645–1658, Dec.1989.

[5] Lam, M. S., *A Systolic Array Optimizing Compiler*, Kluwer Academic Publishers, pp.83–124, 1989.

[6] Miyajima, H., Saitoh, Y., Hironaka, T. and Murakami, K., "Hyperscalar Processor Architecture — Design and Performance of High-Performance-Prototype Processor — (in Japanese)," *IPSJ Technical Report*, ARC–105–7, Mar. 1994.

[7] Miyajima, H. and Murakami, K., "Hyperscalar Processor Architecture (in Japanese)," *IPSJ Technical Report*, ARC–107–4, Jul. 1994.

[8] Murakami, K., "Hyperscalar Processor Architecture — The Fifth Approach to Instruction-Level Parallel Processing — (in Japanese)," *Proc. 1991 Joint Symp. on Parallel Processing*, pp.133–140, May 1991.

[9] Nakamura, H. et al., "A Scalar Architecture for Pseudo Vector Processing based on Slide-Windowd Registers," *Proc. 1993 Int'l. Conf. on Supercomputing*, pp.298–307, July 1993.

[10] Rau, B. R. et al., "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs," *Computer*, vol.22, no.1, pp.12–35, Jan.1989.

[11] Wulf, W. A., "The WM Computer Architecture," *ACM SIGARCH Comput. Architect. News*, vol.16, no.1, pp.70–84, Mar.1988.