九州大学学術情報リポジトリ
Kyushu University Institutional Repository

Hyperscalar Processor Architecture and the Preliminary Performance Evaluation

Miyajima, Hiroshi
Department of Information Systems, Interdisciplinary Graduate School of Engineering Sciences, Kyushu University

Murakami, Kazuaki
Department of Information Systems, Interdisciplinary Graduate School of Engineering Sciences, Kyushu University

Saitoh, Yasuhiko
Department of Information Systems, Interdisciplinary Graduate School of Engineering Sciences, Kyushu University

Hironaka, Tetsuo
Department of Computer Engineering, Faculty of Information Sciences, Hiroshima City University

https://doi.org/10.15017/17366

出版情報：九州大学大学院総合理工学報告. 17 (1), pp.29-40, 1995-06-01. 九州大学大学院総合理工学研究科
バージョン：
権利関係：

九州大学大学院総合理工学研究科報告
第17巻 第1号 29-40頁 平成7年6月

Engineering Sciences Reports, Kyushu University
(KYUSHU DAIGAKU SOGORIKOGAKU KENKYUKA HOKOKU)
Vol. 17, No. 1, pp. 29 - 40, June. 1995

Hyperscalar Processor Architecture and the Preliminary Performance Evaluation

Hiroshi MIYAJIMA*, Kazuaki MURAKAMI**
Yasuhiko SAITOH* and Tetsuo HIRONAKA***

(Receive Febrauary 28, 1995)

This paper describes a novel processor architecture, called *hyperscalar processor architecture*, which encompasses the advantages of superscalar, VLIW, and vector processor architectures and excludes 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[6] is a "post-superscalar" architecture, which encompasses the advantages of (RISC-type) superscalar, VLIW (Very Long Instruction Word), and vector processor architectures and excludes 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 pocessors 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 transistor budget allows.

- Obviously, there is a gap between the number of instructions to be issued at a time and

*Department of Information Systems, Graduate Student
**Department of Information Systems
***Department of Computer Engineering, Faculty of Information Sciences, Hiroshima City University

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 corresponding 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 fashion as VLIW. By compiling loops into software-pipelined VLIW codes and loading each VLIW code before the execution of the corresponding 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 applicantions of hyperscalar processors may range broadly;

- from non-scientific applications with low instuction-level parallelism, which traditional superscalar processors are good at,
- to scientific applications high instruction-level parallelism, which traditional VLIW pocessors are good at, and furthermore
- to scientific applications with both high instruction-level 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 describes the principle of operations of hyperscalar processors in detail, and how to utilize the feature of hyperscalar processors. Section 3 reports some preliminary results of performance evaluation. Section 4 offers a concluding remark.

2 Hyperscalar Processor Architecture

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

2.1 Instruction Registers

Figure 1 shows a basic organization 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,

relieve the disadvantages of VLIW, 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 transistor budget allows. For each of 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.

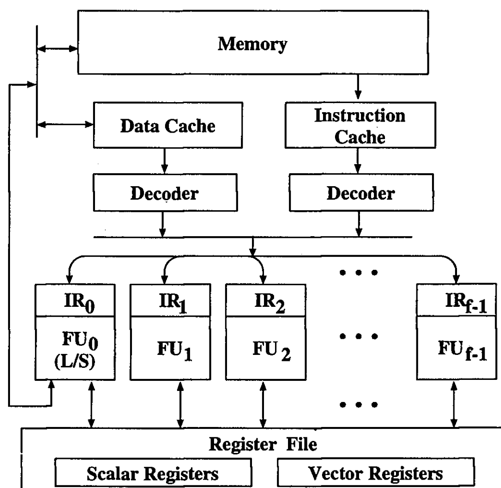

Fig. 1 Basic Organization of Hyperscalar Processors

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 $IR^i = \{IR[i, 0], IR[i, 1], \ldots, IR[i, f-1]\}$ 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 r IR^is can store an VLIW program consisting of r VLIW instructions.

0	IR[0,0]	IR[0,1]	•••••	IR[0,f-1]
1	IR[1,0]	IR[1,1]	•••••	IR[1,f-1]
⋮	⋮	⋮	••⋮••	⋮
r-1	IR[r-1,0]	IR[r-1,1]	•••••	IR[r-1,f-1]
	0	1	•••••	f-1

Fig. 2 Address Space of Instruction Registers

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.

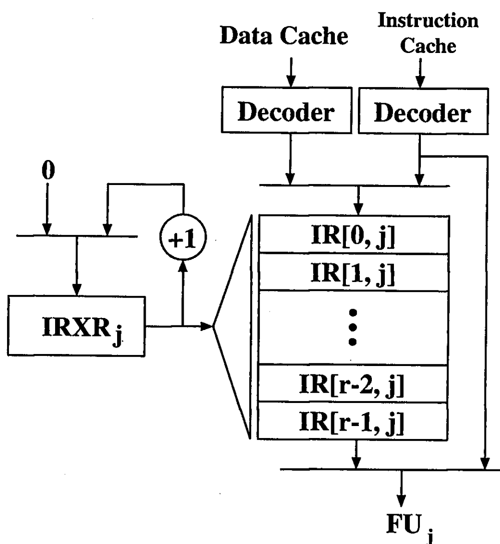

Fig. 3 Structure of IR_j

- 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 *IR*s 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 *IR*s might be marked by a compiler.

At least, the former method is sufficient to be implemented. The instruction decoder used for loading *IR*s by means of LIR instructions is either.

- a special decoder provided for LIR instructions, or
- 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 *IR*s.

2.2 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 *IR*s. Since the instructions in *IR*s are already decoded, they can be dispatched to functional units immediately after they are fetched from *IR*s. 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**.

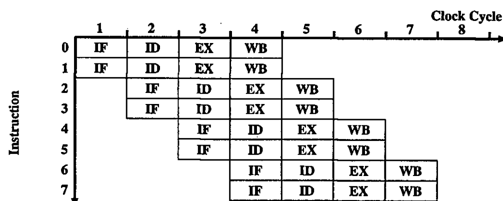

Fig. 4 Instruction Pipeline in Normal Mode

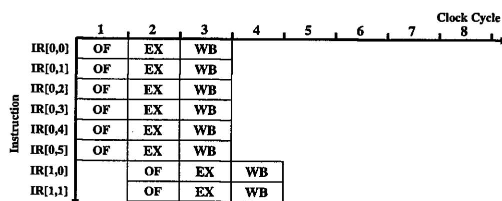

Fig. 5 Instruction Pipeline in Turbo Mode

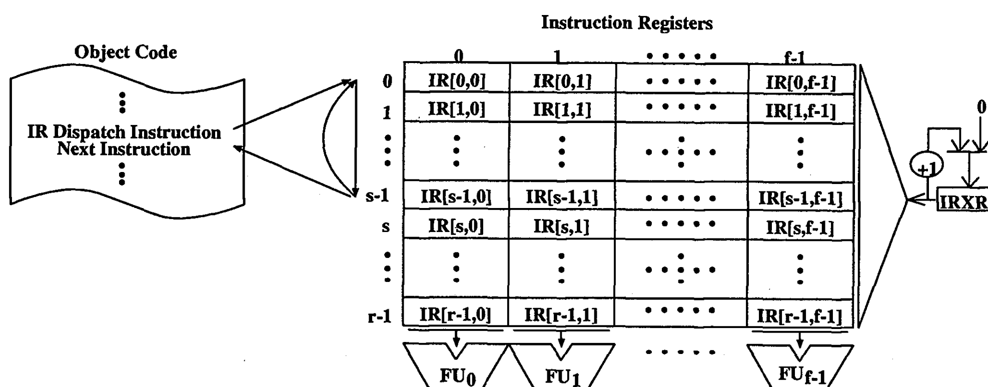

Fig. 6　Principle of Operations of Hyperscalar Processor

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.

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 *IR*s with a sequence of instructions corresponding to the program segment. LIR instructions might be used for loading *IR*s.
3. After completing the loading of *IR*s, the program transfers the processors into the turbo mode, and then the processor begins to fetch instructions from *IR*s. 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 fetches 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 *IR*s 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.

2.3 Software Pipelining

Compilers for hyperscalar processors should create object codes stored in *IR*s. 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 *IR*s:

- Pseudo vector processing
- Software pipelining

Software pipelining is a loop transformation technique which exploits instruction-level parallelism across consecutive iterations of a loop[4]. Pseudo vector processing 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
```

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 7 shows the object code after software pipelining is applied to the above code. 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

And we assumed that load/store instructions (LD/ST) provide some scale-indexed addressing mode with post-increment for index registers (SR1, SR3, SR11 , and SR13).

The software-pipelined loop consists of the following three parts.

1. **Prolog**: After loading *IR*s 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 *IR*s until the exit condition is satisfied.

```
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)
```

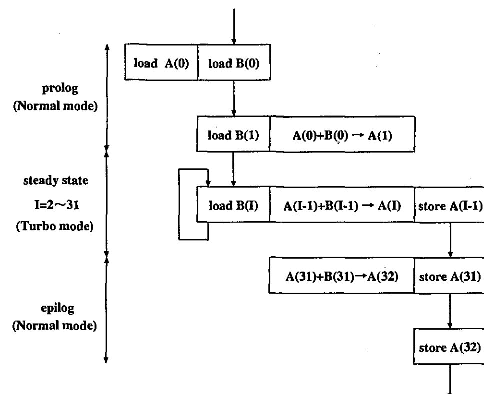

Fig. 7 Software Pipelining

3 Performance Evaluation

We evaluate the performance attainable in hyperscalar processors.

3.1 Evaluation Models

We compare the performance of a hyperscalar processor prototype under development[5] with that of the following counterparts:

- pipelined processor,
- superscalar processor,
- 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[2], 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 prototype 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

Table 1 Specifications of Evaluation Models

<table>
<tr><th colspan="3">Model</th><th>SP</th><th>SSP</th><th>VLIW</th><th>HSP+SB</th><th>HSP+VR</th><th>VP</th><th>VP+M+F</th></tr>
<tr><td rowspan="2">Machine Parallelism</td><td colspan="2">Superscalar Degree</td><td>1</td><td>2</td><td>5</td><td colspan="2">2</td><td colspan="2" rowspan="2">2</td></tr>
<tr><td colspan="2">Hyperscalar Degree</td><td>—</td><td>—</td><td>—</td><td colspan="2">5</td></tr>
<tr><td colspan="3">Peak FLOPC</td><td>1</td><td colspan="6">2</td></tr>
<tr><td rowspan="16">Functional Units</td><td rowspan="4">Integer ALU (IALU)</td><td>Number of Units</td><td colspan="5">1</td><td colspan="2" rowspan="4">—</td></tr>
<tr><td>Issue Latency</td><td colspan="5">1</td></tr>
<tr><td>Result Latency</td><td colspan="5">2</td></tr>
<tr><td>Throughput</td><td colspan="5">1 op/cycle</td></tr>
<tr><td rowspan="4">Floating-Point ALU (FALU)</td><td>Number of Units</td><td colspan="7">1</td></tr>
<tr><td>Issue Latency</td><td colspan="7">1</td></tr>
<tr><td>Result Latency</td><td colspan="7">4</td></tr>
<tr><td>Throughput</td><td colspan="7">1 FLOPC</td></tr>
<tr><td rowspan="4">Floating-Point Multiplier (FMUL)</td><td>Number of Units</td><td colspan="7">1</td></tr>
<tr><td>Issue Latency</td><td colspan="7">1</td></tr>
<tr><td>Result Latency</td><td colspan="7">4</td></tr>
<tr><td>Throughput</td><td colspan="7">1 FLOPC</td></tr>
<tr><td rowspan="4">Load/ Store Unit (L/S)</td><td>Number of Units</td><td colspan="7">2</td></tr>
<tr><td>Issue Latency</td><td colspan="7">1</td></tr>
<tr><td>Result Latency</td><td colspan="7">4</td></tr>
<tr><td>Bandwidth/Unit</td><td colspan="7">8 byte/cycle</td></tr>
<tr><td rowspan="4">Registers</td><td>Integer Register</td><td>Number of Registers</td><td colspan="7">32</td></tr>
<tr><td>FP Register</td><td>Number of Registers</td><td colspan="7">32</td></tr>
<tr><td rowspan="2">Vector Register (Data FIFO)</td><td>Number of Registers</td><td colspan="4">——</td><td colspan="3">16</td></tr>
<tr><td>Number of Elements</td><td colspan="4">——</td><td colspan="3">32</td></tr>
</table>

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 access 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 cycles.

3.2 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

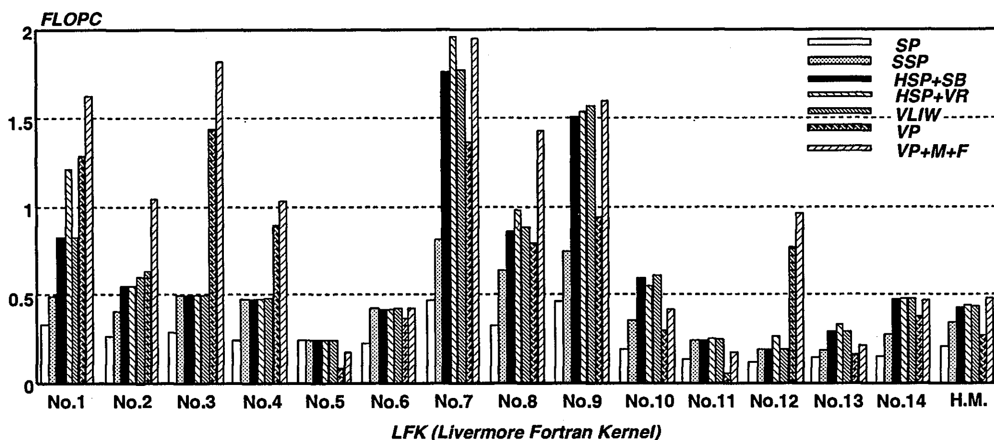

Fig. 8 FLOPC Rating

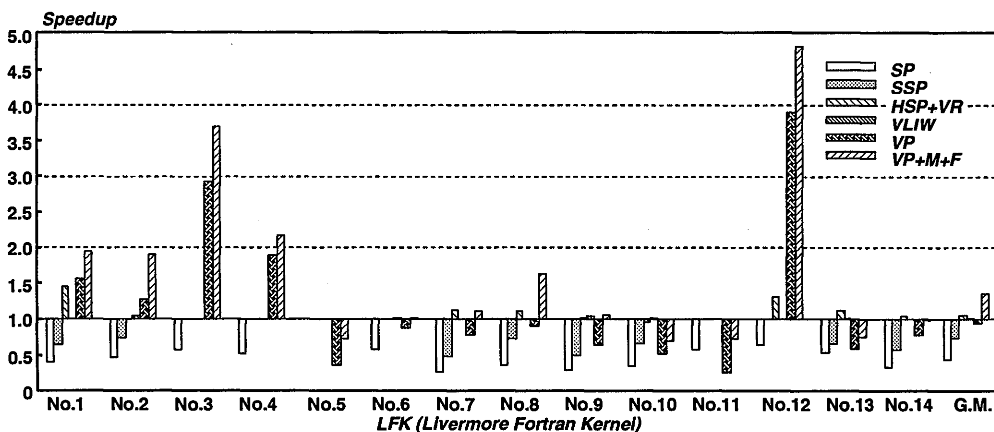

Fig. 9 Speedups over *HSP+SB*

vectorize all the LFKs.

Figure 8 shows the performance of all the evaluation models in terms of FLOPC. **Figure 9** shows the speedup of every evaluation model over the *HSP+SB* model. From **Figure 8**, we recognize that every model ranks as follows, in descending order of the harmonic means of their FLOPC rates:

1) *VP+M+F* : 0.4888FLOPC
2) *HSP+VR* : 0.4578FLOPC
3) *VLIW* : 0.4384FLOPC
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-pocessor 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 ture, 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.

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

- 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.

4 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 Micro-vectorprocessor 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 Architecure—Performance of Software Pipelining— (in Japanese)," *IEICE Technical Report*, VLD93-89, Dec. 1993.

3) 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.

4) Lam, M. S., *A Systolic Array Optimizing Compiler*, Kluwer Academic Publishers, pp. 83-124, 1989.

5) Miyajima, H. and Murakami, K., "The Architecture of the Hyperscalar Processor: NAKASU-1 (in Japanese)," *IPSJ Technical Report*, ARC-107-4, Jul. 1994.

6) 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.

7) Nakamura, H. et al., "A Scalar Architecture for Pseudo Vector Processing based on Slide-Windowed Registers," *Proc. 1993 Int'l. Conf. on Supercomputing*, pp. 298-307, July 1993.

8) 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.

9) Wulf, W. A., "The WM Computer Architecture," *ACM SIGARCH Comput. Architect. News*, vol. 16, no. 1, pp. 70-84, Mar. 1988.