H/S Collaborative Development of a Ubiquitous Processor Free from Instruction Scheduling and Pipeline Disturbance

Masa-aki Fukase
Graduate School of Science and Technology
Hirosaki University
Hirosaki, Japan
slfuka@eit.hirosaki-u.ac.jp

Tomoaki Sato
C&C Systems Center
Hirosaki University
Hirosaki, Japan
tsato@cc.hirosaki-u.ac.jp

***Abstract*—Parallelism is one of fundamental concepts of recent years' trend in developing cutting edge VLSI processors in order to achieve power conscious high performance. HCgorilla is a ubiquitous processor that does not make much of high clock speed, but seeks high performance by applying the architecture of multicore and multiple pipeline. Each of two symmetric cores is composed of Java compatible media pipes and cipher pipes for cipher streaming. Similarly to other processors, HCgorilla is also accompanied with the awkward issue of instruction pipelining. Focusing on this, this paper shows how H/S collaborative parallelism can be used to accelerate the processing speed of the HCgorilla. The novelty of utilizing media pipes as fully as possible owes to a triple scheme for a waved MFU (multifunctional unit), multistack, and interleaved issue of related codes. Since this is useful for out-of-order arithmetic issue in conjunction with parallel stack operation, the triple scheme achieves a processor system free from not only instruction scheduling but also pipeline disturbance. The triple scheme is applied for the improved version of an HCgorilla chip and parallelizing compilers. According to H/S collaboration, these parallelizing steps are moved to web servers. This surely lightens the burden of mobile platforms.**

Keywords-parallelism; H/S collaboration; ubiquitous processor; instruction scheduling

I. INTRODUCTION

Parallelism has been dominating in the recent years' processor market ranging from embedded to high-performance VLSI systems, and the degree of parallelism continues increasing. This trend has been driven by the demand for power conscious high performance especially in ubiquitous fields. To take advantage of the increasing degree of parallelism, sophisticated H/S co-design scheme is indispensable. Actually, the hardware parallelism of multicore and multiple pipeline requires efficient software support for abstracting TLP (thread level parallelism) and ILP (instruction level parallelism). In addition, load balancing and instruction scheduling schemes are needed for cores and pipelines, respectively.

HCgorilla is a ubiquitous processor that does not make much of high clock speed, but seeks high performance by applying the architecture of multicore and multiple pipeline [1, 2]. Each of two symmetric cores is composed of Java compatible media pipes and cipher pipes for cipher streaming. HC is the abbreviation of hardware cryptography. In order to achieve effective media processing over internet, strong conscious has been paid for utilizing floating point arithmetic and Java IP (intellectual property).

Floating point arithmetic is indispensable for basic algorithms of image processing, etc. Since the floating point arithmetic is used in conjunction with integer arithmetic, this requires complicated instruction scheduling to map codes with different latencies on arithmetic execution units. Although the previous version of HCgorilla needed instruction scheduling together with parallelizing executable codes [1], an improved version is free from such process [2]. This is due to the wave-pipelining of a multifunctional unit.

On the other hand, Java IP is required to receive benefit for media processing so far developed. Thus, HCgorilla's media pipe is able to execute Java bytecodes and do stack operation following JVM (Java virtual machine) style. However, stack machines have been generally recognized to be opposed to ILP due to in-order serial process [3]. That is, another kind of pipeline disturbance due to the stack machine's serial processing has still remained even if the problem of instruction scheduling is solved.

As a solution for the awkward issue of the instruction pipelining due to different latencies and the stack machine's serial processing, H/S collaborative parallelism is really reasonable. Actually, the VLSI trend of parallelism owes both hardware and software techniques. However, software requires rather huge resource, power, cost, etc. This surely restricts the scale of systems, and is inconvenient for ubiquitous computing. Thus, hardware parallelism is rather practical in collaboration with software support in order to achieve a processor system free from not only instruction scheduling but also pipeline disturbance.

According to this policy, a triple scheme for a waved MFU (multifunctional unit), multistack, and interleaved issue of related codes is exploited in this study. The waved MFU is free from the scheduling of arithmetic issue. Then, the combination of multistack and interleaved issue is useful for parallel stack operation. The triple scheme is applied for the improved version of an HCgorilla chip and parallelizing compilers.

This work is supported by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc.

978-0-7695-4147-1/10 $26.00 © 2010 IEEE
DOI 10.1109/ICIS.2010.45

II. H/S COLLABORATIVE PARALLELISM

H/S co-design does not mean simply relying on software components for sophisticated multimedia processing, but implicitly supporting hardware activity by the potential of software. Table 1 summarizes the H/S collaborative scheme established in designing HCgorilla. This surveys hardware and software techniques, the object of collaboration, the effect of collaboration, and contribution for ubiquitous applications. Multicore architecture is basically promising for mobile processors as well as PC processors to achieve high performance with less power.

TABLE I. OVERVIEW OF THE H/S COLLABORATION SCHEME

<table>
<tr><th colspan="3">Technique</th><th rowspan="2">Collaboration</th><th rowspan="2">Effect</th><th rowspan="2">Contribution</th></tr>
<tr><th colspan="2">Hardware</th><th>Software</th></tr>
<tr><td colspan="2" rowspan="2">Multicore</td><td rowspan="2">TLP compiler</td><td colspan="2">Dynamic</td><td rowspan="2">Internet</td></tr>
<tr><td colspan="2">Interactive</td></tr>
<tr><td rowspan="5">Multiple pipeline</td><td>LIW fetch</td><td>ILP compiler</td><td rowspan="2">Parallelism</td><td rowspan="3">High performance</td><td rowspan="4">Multimedia</td></tr>
<tr><td rowspan="3">Multistack machine</td><td>Multistack instruction</td></tr>
<tr><td>Interleaved operation</td><td rowspan="2">Disturbance free pipelining</td></tr>
<tr><td colspan="2">Java</td><td colspan="2">Global engineering</td><td>Internet</td></tr>
<tr><td rowspan="3">Waved -MFU</td><td rowspan="3">Needless</td><td>Out-of-order</td><td>High performance</td><td>Multimedia</td></tr>
<tr><td>Power consciousness</td><td>Wearable</td><td>Mobile</td></tr>
<tr><td>High speed clock</td><td>Real time</td><td>Internet</td></tr>
</table>

Although software parallelism has put much emphasis on TLP, ILP is still an important subject even for multithreaded processors. Actually, multithreaded and multicore processors include scalar units that execute arithmetic instructions in parallel. With respect to this viewpoint, the hardware parallelism of multiple pipeline is more important to fully utilize software parallelism. Multiple pipeline is inevitably encumbered by instruction scheduling. This is one of the most important issues to fully utilize multiple pipeline like a regular scalar unit.

The essence of H/S collaborative parallelism for the awkward instruction pipelining lies in how to merge scalar units and issue executable codes in parallel. Incorporating MFU in EX stage frees anxious instruction scheduling, because it takes the same latency to execute any function. However, it surely reduces clock speed. This is caused by the scale up of a multifunctional circuit. Such degradation can be recovered by wave-pipelining, because it has potential to achieve higher speed with less occupied area. Thus, this study finds usefulness in the wave-pipelining of the resultant MFU.

In order to fully utilize the waved MFU, further improvement is required. The waved MFU access stacks to cover Java, which is well suited to follow the global engineering of platform neutrality, multithreading for dynamic interaction over Internet, etc. Thus, multistack is crucial for both fully utilizing the waved MFU and sustaining stack machine's IPC. The multistack is achieved by providing stacks equal to the wave degree of the waved MFU. Each stack is provided with a dedicated arithmetic code. Then, those arithmetic codes are interleaved in order to fully utilize the waved MFU. Both the instruction fetch according to an LIW (Long Instruction Word) mode and the operation of multistack are interleaved to achieve disturbance free pipelined arithmetic processing.

Fig. 1 exactly illustrates multiple pipeline structure in conjunction with multistack and waved-MFU. The parallel degree of the multistack is made correspond to the wave degree of the execution stage. In this case the degree is three for convenience sake. The executable codes are output from an LIW compiler that extract ILP, and is stored in an instruction cache. The issue of stack-related codes like load, store, arithmetics are interleaved. Also, the stack operations of pop up and push down are interleaved. The twofold interleaving makes the stack related codes occupy a single stack continuously by one clock.

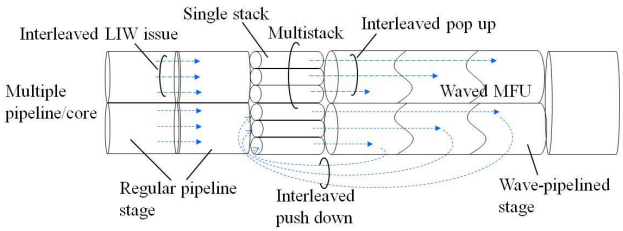

Figure 1. Multiple pipeline structure in conjunction with multistack and waved-MFU.

Fig. 2 illustrates the interleaved LIW issue in the case of simple summation. Although more practical summation is carried out for floating point numbers, integer summation is shown to focus on interleaving. Anyway, the simple summation is frequently used, for example, as a part of a normalized correlation factor. This factor is used in stereo matching, which is a basic obstacle detection algorithm for the image processing of ASV (advanced safety vehicle) and ITS (intelligent transport system). Since the parallel degree is three corresponding to Fig. 1, the source code is unfolded into three stack codes, and these are interleaved. Here, "1iadd" is an executable code that adds the stack's first top and the second top, and pushes down the stack's first top.

III. HARDWARE DESIGN

HCgorilla is a processor architecture designed for the ubiquitous computing of media processing and cipher streaming as well. Basically, HCgorilla is composed of two symmetric cores to cover bidirectional communication. Fig. 3 shows the hardware organization of HCgorilla,5 that is the fifth version of HCgorilla designed in this study. Each core has two arithmetic media pipes and a cipher pipe. The arithmetic pipe has a double stack and a two-waved MFU. Since the parallel degree of the media pipe is two, each core is able to execute four stack-related codes in parallel

following JVM style. The cipher pipe with a random number generator executes a SIMD mode cipher codes.

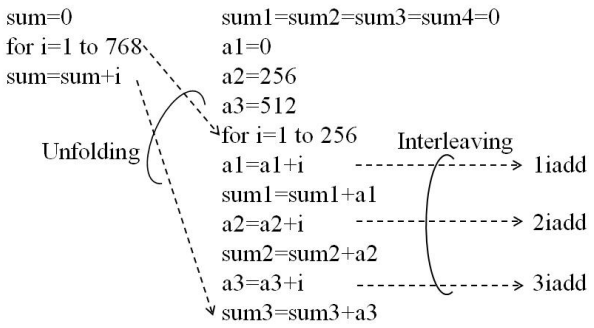

Figure 2. Interleaved issue.

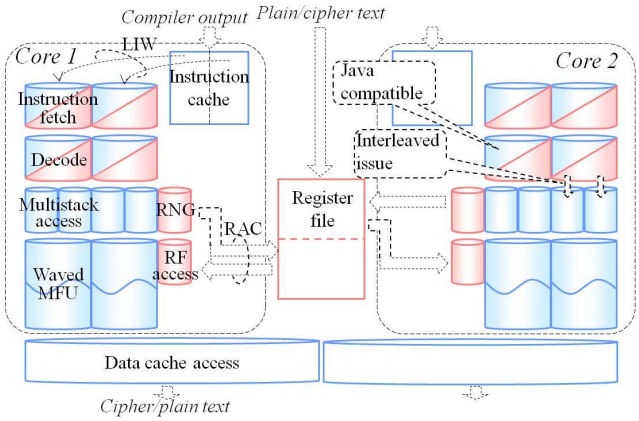

Figure 3. Hardware organization of HCgorilla.

HCgorilla.5 has been implemented by using a 0.18-µm CMOS chip [2]. Table 2 summarizes the architectural aspects of the HCgorilla.5 chip compared with those of dominant versions so far developed. Although HCgorilla.4 improves the HCgorilla.3 chip [1] that lacks floating point units, it requires instruction scheduling between floating and integer units. HCgorilla.5 is free from this issue due to the use of the two-waved MFU and the double stack. Interleaved issue to the double stack is useful for sustaining IPC or full utilization of the waved MFU. The HCgorilla.5's instruction set is composed of 2 cipher codes and 102 media codes. 58 media codes are Java compatible. The parallel execution of media codes and cipher codes is indispensable for the basic ability of HCgorilla that unifies Java features, strong security, low power, and high throughput.

TABLE II. SPECIFICATIONS OF HCGORILLA FAMILY

Chip version					HCgorilla.3	HCgorilla.4	HCgorilla.5
Design Rule					ROHM 0.18 µm CMOS		
Per processor	Chip area				5.0 mm × 7.5 mm		
	Power Supply				1.8 V (I/O 3.3 V)		
	Power consumption				241 mW		274 mW
	Clock frequency				330 MHz	400 MHz	200 MHz
	Instruction cache				2 byte × 32 word × 2		2 byte × 64 word × 2
	Data cache				2 byte × 128 word		2 byte × 128 word × 2
	Stack memory				2 byte × 8 word × 4		2 byte × 16word × 8
	Register file				2 byte × 64 word		2 byte × 128 word
	No. of cores				2		
Per core	Pipeline		Number		2		
		Media	EX	IU	2-wave × 2		2-wave MFU × 2
				FPU	NA	5-clock × 2	
			No. of stacks		1		2
			ILP degree		2		4
			Throughput			0.07-0.09 GIPS	0.13 GIPS
			Java bytecode		61	77	58 (102)
		Cipher	Number		1		
			RNG		4-bit LFSR		6-bit LFSR
			Throughput		0.1-0.2 GOPS		
			SIMD code		2		
Current status					Chip	Synthesis	Chip

IV. SOFTWARE DESIGN

The design of software for HCgorilla includes Java as is described in Table 1. Fig. 4 illustrates the flow of Java language processing for HCgorilla, which is compared with that for commercial processors. The turning point of the both flows is the processing of class files. While commercial processors directly receive them via Internet to take the benefit of platform neutrality, HCgorilla receives the product of class files produced in web servers. The processing of class files is supported by the software composed of a Java interface and parallelizing compilers.

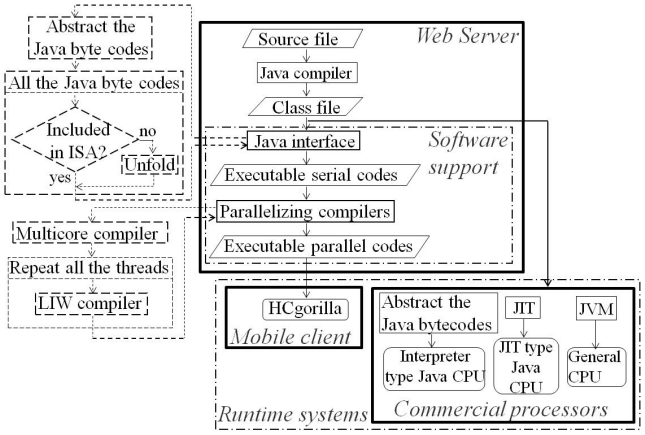

Figure 4. Java language processing flow for HCgorilla vs. commercial processors.

A main idea shown in Fig. 4 is moving the necessary parallelizing steps from the processor to a web server. These are installed in web servers, and finally outputs executable parallel codes to Internet. They are directly run on HCgorilla. Since the small scale is imposed on ubiquitous clients, installing the software support in external large servers is

very convenient for HCgorilla-embedded small platforms. Although any processor is allowed to run Java by installing JVM and JIT (just-in-time compiler), this needs more software load and memory space. Such imposition surely degrades response time, power consciousness, usability, cost, and the performance of small mobile devices.

Fig. 5 shows more clearly the physical arrangement of ubiquitous clients embedded with HCgorilla chip, web server, software support, and parallelizing compilers. Such architecture can be easily established by simply installing the software support in web servers. For example, the software support may run on proxy servers.

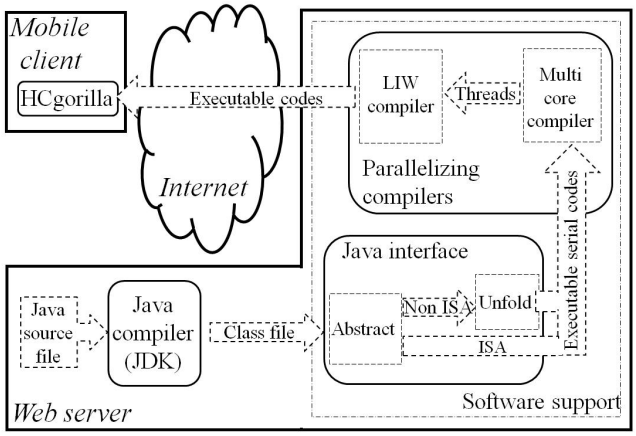

Figure 5. Global architecture of HCgorilla, web server, software support, and parallelizing compiler.

The one of anticipative drawbacks of this approach is web delays. Obviously, it will cost some time to transfer the executable code over the internet. However, the transfer of class files to commercial processors also takes some time. In addition, the transfer time is not so important for the evaluation of web delays [4]. The main factor of web delays is the response time of web servers. Another apprehension about this approach is to keep security during the transfer. However, getting the executable codes over the internet does not yield a problem of trust, because Java is basically seeks for global standard in Internet.

Table 3 summarizes the granularity of parallelism of the software support. According to HCgorilla's parallelism, the parallelizing compilers for HCgorilla are a multicore compiler and LIW compiler. Their role is to abstract media codes that are executable in parallel by the double core and multiple pipelines. In order to fully utilize the parallelism of multicore, TLP is taken into account of. Although multithreading is not always only one software technique for parallelizing applications run on multicore chips, it is not the main concern of this study.

TABLE III. GRANULARITY OF PARALLELISM

Software	Granularity	Abstracted module or codes	Language level
Java interface	TLP	MyThread	Source code
		Function	
Multicore compiler		Method	Executable code
		Library function	
		Loop	
		Others	
LIW compiler	ILP	Integer codes	
		Floating point codes	
		Cipher codes	

Fig. 6 shows the basic algorithm of the HCgorilla's multicore compiler designed to abstract TLP from executable serial codes or the Java interface output. The multicore compiler judges threading by looking for return process that expresses the end of instructions sequence or thread. Other abstraction units like functions and loops should be incorporated in the next step of this study. Fig. 7 shows the basic algorithm of the LIW compiler that abstracts ILP from each thread and does reorder and renaming as shown in. The LIW compiler outputs the codes executable in parallel by the media pipes. The output is mapped on an instruction cache within each core.

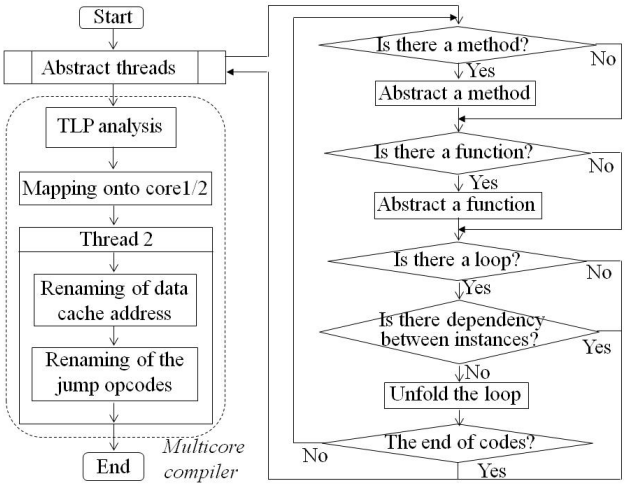

Figure 6. The algorithm of HCgorilla's multicore compiler.

Although main techniques illustrated in Figs. 6 and 7 are not so specialized, there exist some novelties. They are as follows. Firstly, the LIW compiler basically reflects the parallel structure of the execution stage and abstracts codes that use one of multiple execution units. Other codes that do not use execution units are not the target of the LIW compiler, because they are basically processed in serial. Secondly, HCgorilla's jump codes do not need renaming.

Generally, renaming or the readdressing of parallelized code sequences is required for both TLP and ILP abstraction in order to avoid the conflict of data cache access. This complements the logical share of data cache by the cores. Some of jump codes direct absolute addresses, and such codes need renaming by modifying their destination addresses in parallelization. However, renaming is not necessary for HCgorilla's jump codes, because all of the HCgorilla's jump codes point destination indicated by the sum of the address of itself and the operand.

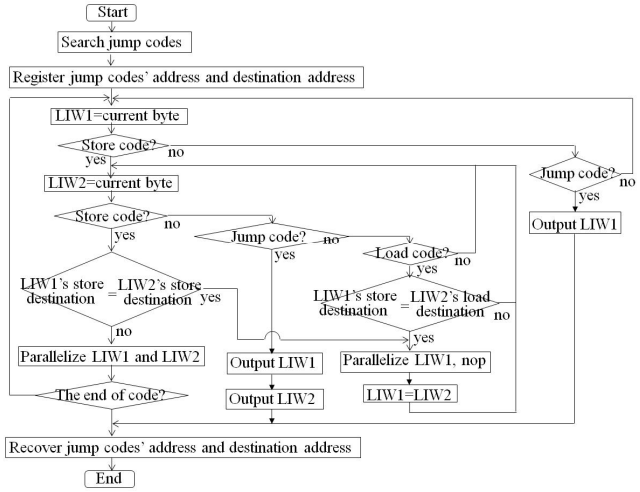

Figure 7. The algorithm of HCgorilla's LIW compiler.

Table 4 summarizes the specifications of the implementation example of the software support. While the Java interface is written in *C*, the parallelizing compilers have been written in Java. These are combined within web servers as is shown in Fig. 5.

TABLE IV. SPECIFICATIONS OF THE SOFTWARE SUPPORT

	Description language	No. of statements	Input	Output
Java interface	*C*	300	Class file (Java compiler output)	Java bytecodes
Multicore compiler	Java	90	Java bytecodes	Thread1, Thread2
LIW compiler	Java	290	Thread1/2	Executable codes

V. EVALUATION

The effectiveness of the idea described in this paper is shown in view of hardware and software aspects. Fig. 8 shows the effect of the triple scheme for a waved MFU, multistack, and interleaved issue by using an arithmetic test program that sums integers. The running time is derived from the analysis on the space-time chart drawing the pipeline behavior of the media pipes. While HCgorilla.4 executes four threads at 400 MHz, HCgorilla.5 does eight threads at 200 MHz. Since the effect of arithmetic parallelism is mostly dominant, HGgorilla.5 is faster excepting upper limit x is small. Small x reflects the dominancy of clock speed. In running practical application programs that repeat similar multimedia process, HCgorilla.5 is more superior in view of parallelism.

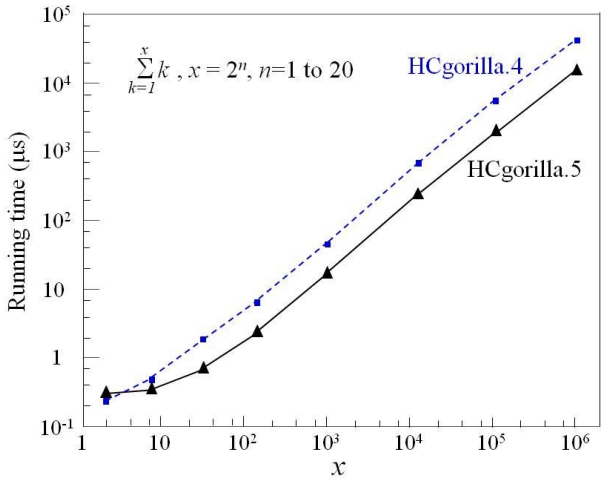

Figure 8. Running time of an arithmetic test program.

The prototyping compilers shown in Table 4 have been evaluated by using arithmetic media codes for simple summation and combination, Tower of Hanoi, and a Java benchmark of SPEC JVM98. The metrics are memory space, running time, and the effect on performance. Fig. 9 shows IPC before and after parallelization of the test programs of simple summation and combination. The derivation of IPC is similar to that of running time shown in Fig. 8.

Fig. 9 shows that IPC increases after parallelization. Thus, the feasibility of the parallelizing compilers in combination with multiple pipelines is obvious. Then, 60% improvement is achieved for Summation and 20% for Combination. The reason why the improvement of Summation is larger than Combination is because Summation's ILP is more explicit. Since the target processor has only two cores, the data presented in Fig. 9 show not so large improvement. This implies that marginal improvement will surely increases as the number of cores in the case of simply structured programs easily divided into parallel codes. This is worth of effective feedback from the evaluation of parallelizing compilers to hardware design.

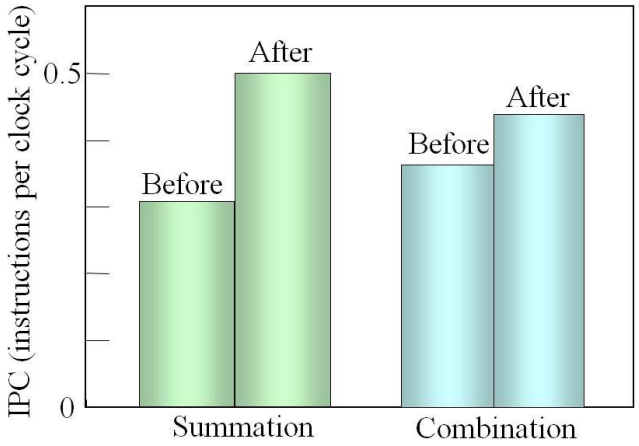

Figure 9. IPC before and after parallelization.

Table 5 summarizes the effect of parallelization. Although the evaluation is primitive, it shows the reasonability of the parallelizing compilers. Tower of Hanoi cannot be handled by the prototyping compilers. Occupied words are the total number of instruction cache addresses. The length of thread 1 or thread 2 is shorter than the length of serial codes before compilation due to the effect of parallelization. The occupied words of instruction cache have been determined by the great of thread1 and thread2. The occupied area of HCgorilla's instruction cache has been measured to optimize the hardware chip performance like pipeline degree, clock speed, and power dissipation. The total number of clock cycles has been counted to measure running time. The number of clocks and running time have been derived by analyzing pipelined behavior on a time and space coordinates.

TABLE V. EFFECT OF PARALLELIZATION

Program			Summation (Integer)	Combination	Tower of Hanoi	Peptest
Before multicore & LIW compile	No. of executable codes		61	74	73	5,693
	Occupied word × byte		61 × 1	74 × 1	73 × 1	5,693 × 1
	No. of clocks		23,554	181		
After multicore & LIW compile	No. of executable codes	Thread1	26	5	Parallelization impossible	2,520
		Thread2	21	67		2,849
	Occupied words of instruction cache		26	67		2,849
	Running time		59msec	440nsec		

In view of the ubiquitous network shown in Fig. 5 that is a long term goal of this study, the overhead analysis about the cost for running the parallelizing compiler outside the mobile client should be performed. In addition, practical gain of using the parallelizing compiler must be clarified in more detail in a multimedia streaming environment. How much performance gain and power reduction can be achieved is desirable for the H/S collaborative development.

VI. SUMMARY

This article has described the triple scheme for the development of the ubiquitous processor HCgorilla free from not only instruction scheduling free but also pipeline disturbance. The triple scheme has been applied for the HCgorilla.5 chip and the development of parallelizing compilers according to H/S collaborative development strategy. Although more implementation examples and experiments are needed to prove the proposed approach, the effect of the triple scheme on performance and the feasibility of software support have been shown.

The next step of this study will be a more in depth analysis on (1) HCgorilla.5's advantage against that of the version 4 by using more practical test programs with floating point arithmetic operations, (2) practical gain of using the parallelizing compilers in a multimedia streaming environment, (3) quantitative feedback for the H/S collaborative development through the overall evaluation of the HCgorilla chip and the parallelizing compilers, (4) overhead about the cost for running the parallelizing compiler outside the mobile client.

REFERENCES

[1] M. Fukase, K. Noda, A. Yokoyama, and T. Sato, "Design and Chip Implementation of the Ubiquitous Processor HCgorilla," Proc. of ASP-DAC 2009, pp. 129-130, Jan. 2009.

[2] M. Fukase, R. Murakami, and T. Sato, "Design and Chip Implementation of an Instruction Scheduling Free Ubiquitous Processor," ASP-DAC 2010 (In press).

[3] S. Nakagawa and H. Yanagi, "Development of Realtime JavaTM Processor Execution Core," OMRON TECHNICS, Vol. 40, No. 1, pp. 38-42, 2000.

[4] M. Zari, H. Saiedian, and M. Naeem, "Understanding and Reducing Web Delays," Computer Magazine, Vol. 34, No. 12, pp. 30-37, Dec. 2001.