Compilation Techniques Specific for a Hardware Cryptography-Embedded Multimedia Mobile Processor

Masa-aki FUKASE
Graduate School of Science and Technology, Hirosaki University
Hirosaki, 036-8561, Japan

and

Tomoaki SATO
C&C Systems Center, Hirosaki University
Hirosaki, 036-8561, Japan

ABSTRACT

The development of single chip VLSI processors is the key technology of ever growing pervasive computing to answer overall demands for usability, mobility, speed, security, etc. We have so far developed a hardware cryptography-embedded multimedia mobile processor architecture, HCgorilla. Since HCgorilla integrates a wide range of techniques from architectures to applications and languages, one-sided design approach is not always useful. HCgorilla needs more complicated strategy, that is, hardware/software (H/S) co-design. Thus, we exploit the software support of HCgorilla composed of a Java interface and parallelizing compilers. They are assumed to be installed in servers in order to reduce the load and increase the performance of HCgorilla-embedded clients. Since compilers are the essence of software's responsibility, we focus in this article on our recent results about the design, specifications, and prototyping of parallelizing compilers for HCgorilla. The parallelizing compilers are composed of a multicore compiler and a LIW compiler. They are specified to abstract parallelism from executable serial codes or the Java interface output and output the codes executable in parallel by HCgorilla. The prototyping compilers are written in Java. The evaluation by using an arithmetic test program shows the reasonability of the prototyping compilers compared with hand compilers.

Keywords: Processor, H/S Co-Design, CMOS, Parallelizing Compiler, Java, Pervasive Computing, Hardware Cryptography.

1. INTRODUCTION

The emergence of pervasive computing is due to Internet expansion, multimedia computing, and mobile computing. Although this is inevitability, the expansion or diversity of pervasive platforms has also caused notorious security issues as is illustrated in Fig. 1. Facing with such circumstances in our daily life, we have felt alternative impressions, diversity or security [1]. Many ways to guarantee secureness have been taken, but they are time and power consuming in a word. This is because security techniques are mostly implemented in software and pervasive computing treats huge amount of multimedia data. The development of high performance application-specific and domain-specific systems is an innovative research of common interest [2]. In order to clear overall issues, the hardware integration of related techniques is indispensable for pervasive environment.

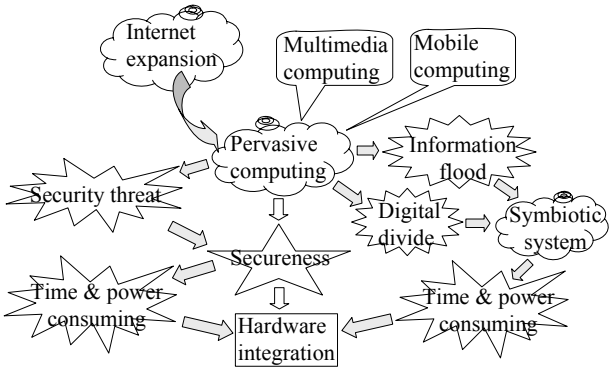


Fig. 1. The trend and issues of pervasive computing.

The development of single chip VLSI processors is the key technology of ever growing pervasive computing to answer overall demands for mobility, speed, and security in such a field. Thus, we have so far exploited following processors.

a. A multimedia mobile processor named gorilla [3, 4].
b. A hardware cryptography-embedded processor named RAP (random addressing-accelerated processor) [5, 6]. The CMOS implementation of RAP showed its possibility for stream cipher [7].
c. The integration of gorilla and RAP into HCgorilla [8]. HCgorilla was implemented by using CMOS standard cell technologies [9]. Also, a language processing support was studied to reduce the load and increase the performance of HCgorilla [10], [11].

For these processors, one-sided design approach is not always useful, because they integrate a wide range of techniques from architectures to applications and languages. Especially, HCgorilla needs more complicated strategy, that is, hardware/software (H/S) co-design to cover sophisticated features of Java compatibility, hardware security, low power, and high throughput.

Compilers are the essence of H/S co-design. Needless to say Proebsting's law: a compiler advances double computing power every 18 years, compilers, especially paralellizing compilers occupy absolute position in developing sophisticated processors. Since compilers are most crucial for practicing

HCgorilla's parallelism in pervasive environment, we focus in this article on our recent results about the design, specifications, and prototyping of parallelizing compilers for HCgorilla. The prototyping compilers are written in Java and evaluated in case of arithmetic media codes compared with a hand compiler. Although the evaluation is primitive, it shows the reasonability of the prototyping compilers.

2. OVERVIEW OF HCGORILLA

Table 1 summarizes processors we have so far developed for pervasive computing. The hardware cryptography-embedded multimedia mobile processor, HCgorilla has been developed considering the unification of RAP and gorilla has advantageous potential for downsizing, power reduction, and speedup.

Table 1. Architectures and processor derivatives related to HCgorilla.

Name	Architecture								Microarchitecture				
	ISA		Parallelism										
	No. of instructions	Format	No. of cores	ILP	Pipelining			Hardware crypto-graphy	Control	Chip	Process	Clock (MHz)	Current status
					Regular		Waved						
					Deg.	Number							
gorilla.1	16	JVM	2	2-degree	8	2 media pipes	2-wave EX	Not available	Wired logic	gorilla035	0.35μm	240	4.9-mm chip
gorilla.2					7		Not available			gorilla035v2		200	Synthesis
RAP	17	RISC	1	Not available	5	1 cipher pipe		SISD		Not available	FPGA	45	FPGA
HCgorilla.1	18	JVM	2	2-degree	8	2 cipher-embedded media pipes				HCgorilla035	0.35μm	150	4.9-mm chip
HCgorilla.2	63				7	3 (2 media pipes and a cipher pipe)	2-wave EX	SIMD		HCgorilla018	0.18μm	400	2.8-mm chip
										HCgorilla018 v2			Synthesis
										HCgorilla018 v3			5.9-mm chip

2.1 H/S co-design

H/S co-design scheme is indispensable for achieving PC-like performance as well as pervasive computing features. According to this, we have designed HCgorilla. Table 2 summarizes the strategy we have taken into account of in developing HCgorilla. The top priority in developing HCgorilla is to determine the target or application field. It should cover the sophisticated multimedia processing, user-friendly mobility, and comfortable pervasive computing environment. As is clear from Table 2, individual strategies and techniques to cover these demands overlap and are closely related each other. Also their relations are very complicated.

Let us discuss some of key techniques shown in Table 2. The power conscious highly performance of HCgorilla is due to a novel architecture following symmetric multicore, superscalar, and LIW (long instruction word) processor techniques. LIW is not so broad parallelism like VLIW (very long instruction word), yet it is effective to practically enhance multimedia communication that deals with large quantity of data in pervasive environment.

The parallelism of multicore and LIW is very promising for power conscious high performance with the aid of parallelizing compilers. It is not always necessary to distinct multicore processors from multithreaded processors [12]. Although multithreading is not always only one software technique for parallelizing applications run on multicore chips [13], software approach is not our main concern. Thus, we take into account of TLP (thread level parallelism). TLP and ILP (instruction level parallelism) are on the back of instruction folding by API (application program interface) and a LIW compiler. In order to cover LIW in conjunction with Java native codes, microprogramming technique is useful. Then, the key technique to detect the length of each instruction and distribute it to appropriate pipes is superscalar-like IFU (instruction fetch unit). We make this by a wired logic. On the other hand, data level parallelism is covered by SIMD (single instruction stream multiple data stream) mode execution. This is indispensable for multimedia streaming.

Table 2. H/S co-design strategy for HCgorilla.

Application field	Demand	Strategy	Technique	
			Hardware	Software
Multimedia	Entertainment	Functionality	ISA: Java	API, compiler
			ISA: Cipher	SIMD
		Media streaming		
	High performance	Parallelism	IFU	LIW, compiler
Mobile	Wearable	Power consciousness	Wave-pipeline	Needless
Ubiquitous	Real time	High speed clock		
	Dynamic	Multithreading	Multicore	TLP
	Interactive			
	Learning	Object-oriented	Interpreter type Java CPU	API
	Quick response			
	Global engineering	Platform neutrality		
	Reliable diversity			
		Strong cryptography	RAP	Needless

Wave-pipelining is really effective for both PC-level high speed and mobile-level low power dissipation [14, 15]. Also, wave-pipelining reduces software loads, because it is completely a hardware approach. Conventionally, long delay times have been wasted in accessing web pages and drawing contents. This is due to iterative process within web servers, network switches, and local clients to safely treat large amount of packets.

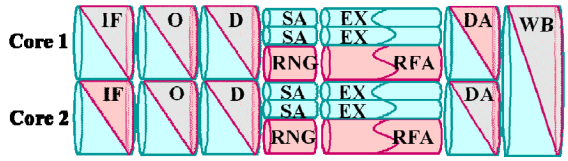
Java is one of the most promising solutions for the functionality of attractive multimedia entertainment. In order to prepare real time protection of huge amount of multimedia data, cipher proper instructions should be added to ISA (instruction set architecture). What we have devised about Java are as follows.

a. Compact ISA: Although complicated features demanded to ISA tend to increase it, compactness is also important in order to lighten the burden on hardware design. This is solved by the format of code and the number of codes.
b. Platform-friendly language processing: Due to an intermediate form or class file produced by Java compilers, Java is more awkward than regular languages. Since pervasive clients use small scale systems, we make large servers cover the preprocessing of complicated class files by API and compilers.
c. Direct execution of Java bytecode without JVM (Java virtual machine): Although preprocessed class files are further interpreted by JVM or JIT (just-in-time compiler) built-in runtime systems, they need more ROM space. This degrades response time, power consciousness, usability, cost, and performance of mobile devices. Considering hardware property as well as Java compatibility, we apply the interpreter type Java CPU.

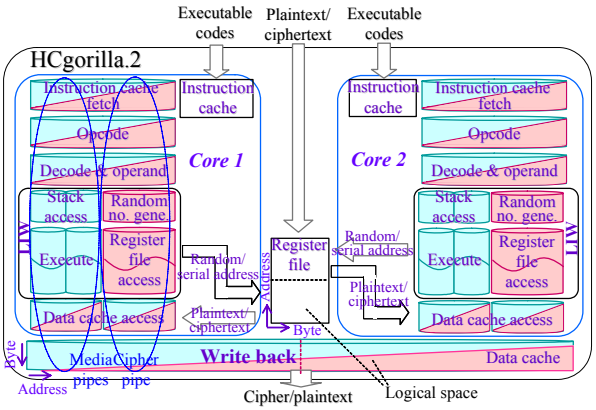
2.2 Hardware organization

Fig. 2 shows the architecture of HCgorilla. Fig. 2 (a) illustrates the hardware parallelism of double core and multiple pipelines. Each core is composed of Java compatible two media pipes and a cipher pipe. These pipes share the instruction fetch, opcode fetch, decode, data cache access, and write back stages. In spite of three pipes, instruction cache is made issue two executable codes in parallel. This is enough even for fully parallel execution. This is because SIMD mode cipher streaming does not need the instruction fetch stage in the steady state of streaming, and executes the streaming by repeating the later four stages. The wired logic instruction fetch stage detects the length of each instruction and distributes it to appropriate pipes.

Media pipes have the stack access and execute stages. Since the media pipe is a devised interpreter type Java CPU, operands are issued from stacks to the execute stages similar to JVM. The cipher pipe has RNG (random number generator) and register file access stages. The buffer of cipher streaming is provided with a register file. Each pipe is partly wave-pipelined for the power conscious high speed processing. Fig. 2 (b) shows the latest derivative of HCgorilla.2 more in detail [16].



(a)



(b)

Fig. 2. The architecture of HCgorilla. (a) Parallelism. (b) Detail of the latest derivative.

2.3 Cipher process

Since the cipher pipe is originally dedicated to HCgorilla, let us describe it more in detail. The cipher pipe enables non-traditional cryptographic streaming due to the extremely long cycle of random numbers. These are easily produced by making RNG with LFSR (linear feedback shift register). LFSR falling into the category of M-sequence requires trivial additional chip area and power dissipation. A tiny n-bit LFSR produces the huge n-th power of 2 random numbers. The built-in RNG is directly connected to the address line of data cache. Due to the direct connection, the content of a specified register file location is transferred to data cache addressed by the RNG output. The transfer rate is several bytes per clock depending on the width of the register file and data cache. The register file plays the role of a pseudo-streaming buffer.

In order to save hardware resources, the register file for buffering and data cache for destination operands are physically shared by the cores. Although they are physically shared, it is logically divided by using quasi 2-port I/O for the single memory space. The physical share means that the cores do not have their own memories but use a single memory space in common. In order to mutually use the single space, their addressing is not monotonous or logically divided. This is made by automatically or hardwarily biasing addresses to be distinguished. This is equivalent to have quasi 2-port I/O.

The SIMD mode execution by the cipher pipe is due to the two executable codes, that is, a random store code *rsw* and a random load code *rlw*. These are dedicated to simple and fast cryptographic streaming. Fig. 3 illustrates an internal behavior in executing *rsw* for the encryption of a byte string like text, image, etc. Cryptographic streaming is the continuous encryption or decryption of such data. Here, $d_1d_2d_3d_4d_5$ exemplifies a plaintext block. 30241 is corresponding key or RNG outputs. $d_2d_5d_3d_1d_4$ is a resultant encrypted block.

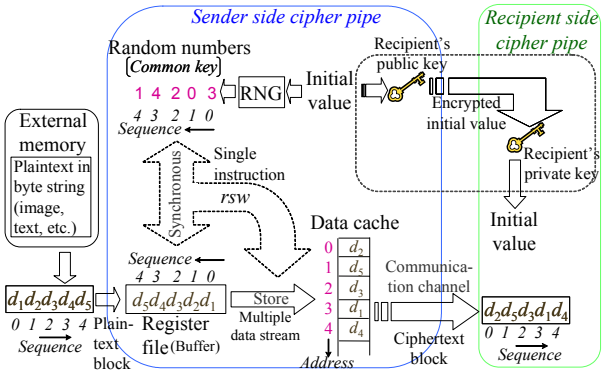


Fig. 3. Execution of the SIMD mode cipher code *rsw*.

In the execution of *rsw*, the block and RNG's output are synchronized according their sequence. For example, the first byte data "d_1" and the first random number "3" are synchronized and stored to the 3rd location of the data cache. The sequence of random addressing store like this results in the formation of an encrypted block in the data cache. Then, the encrypted block is forwarded to the recipient and decrypted similarly by *rlw*.

The common key process by using *rsw* and *rlw* is called RAC (random number-addressing cryptography) [17]. Note that random numbers themselves are not exchanged between the sender and recipient. The initial-value of RNG is crucial for RAC and it should be treated by hyper protection. Practically, public key is promising for communication between a sender and a recipient, though it is no account in this study.

The block length and the size of the register file and data cache are specified as follows.

Block length ≤ n-bit LFSR's output length (2^n)
= register file's logical space size
= data cache's logical space size.

Here, the size is the product of length and width (byte). The logical space is the half size of a physical space as shown in Fig. 2 (b). The block length, register file size, and data cache

size crucially influence the performance and security, assuming the processing of one block per clock.

3. SOFTWARE ASPECTS

Various aspects related to the software system of HCgorilla, except compiler techniques, are concentrated in this chapter.

3.1 Software Support System

Usual Java language systems have needed rather complicated language processing and hardware organization in order to achieve the platform neutrality. This is due to an intermediate form or class file produced by using the Java compiler. Although the class file contains many kinds of information, the essence is Java byte codes, and the remaining is additional information. This is really convenient for JVM, but secondary for a processor itself. Yet, the complicated language processing is imposed on platforms in general.

Since HCgorilla is a devised interpreter type Java CPU without JVM, HCgorilla also needs the software support to adjust class files and executable codes. The software support includes Java interface and parallelizing compilers as shown in Fig. 4. The software support best maps software threads onto instruction cache.

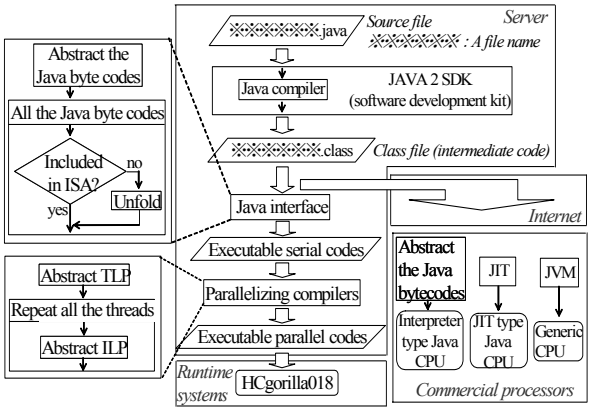


Fig. 4. Language process flow.

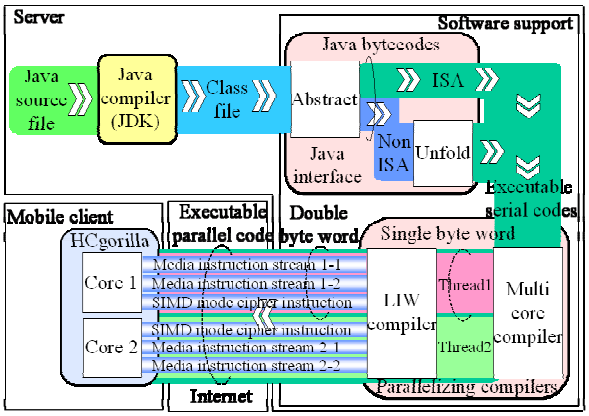


Fig. 5. Software support for HCgorilla.

In order to release smaller platforms from heavy duties for Java language processing, we are planning to install the software support in larger servers. This is also effective to increase the performance of client platforms. Fig. 5 exemplifies a load-balancing network system. The network is composed of servers, Internet, and HCgorilla systems. Then, an HCgorilla system is formed by a mobile client embedded with an HCgorilla chip and software support.

3.2 Executable Code Set

Table 3 summarizes the code set of HCgorilla.2 composed of the two cipher codes and 61 Java compatible media codes. These are carefully selected from the 202 Java bytecodes for ever growing usage of Java in mobile fields [18].

Table 3. Code set of HCgorilla.2.

	Opcode		Operand									
	Mnemonic	Binary	(byte)									
Cipher codes	rsw	0xF0	0	iconst_1	0x04	0	astore_2	0x4D	0
	rlw	0xF1	0	iconst_2	0x05	0	astore_3	0x4E	0
	nop	0x00	0	iconst_3	0x06	0	pop	0x57	0
	bipush	0x10	1	iconst_4	0x07	0	pop2	0x58	0
Media codes	iload	0x15	1	iconst_5	0x08	0	dup	0x59	0
	istore	0x36	1	sipush	0x11	2	dup_x1	0x5A	0
	iadd	0x60	0	aload	0x19	1	dup_x2	0x5B	0
	isub	0x64	0	iload_0	0x1A	0	dup2	0x5C	0
	ineg	0x74	0	iload_1	0x1B	0	dup2_x1	0x5D	0
	ishl	0x78	0	iload_2	0x1C	0	dup_x2	0x5E	0
	ishr	0x7A	0	iload_3	0x1D	0	imul	0x68	0
	iand	0x7E	0	aload_0	0x2A	0	iushr	0x7C	0
	ior	0x80	0	aload_1	0x2B	0	iflt	0x9B	2
	ixor	0x82	0	aload_2	0x2C	0	ifge	0x9C	2
	ifeq	0x99	2	aload_3	0x2D	0	ifgt	0x9D	2
	ifne	0x9A	2	astore	0x3A	1	ifle	0x9E	2
	if_icmplt	0xA1	2	istore_0	0x3B	0	if_icmpeq	0x9F	2
	goto	0xA7	2	istore_1	0x3C	0	if_icmpne	0xA0	2
	iconst_m1	0x02	0	istore_2	0x3D	0	if_icmpge	0xA2	2
	iconst_0	0x03	0	istore_3	0x3E	0	if_icmpgt	0xA3	2
				astore_0	0x4B	0	if_icmple	0xA4	2
				astore_1	0x4C	0			

Fig. 6 shows the assembler/executable code format of HCgorilla. The code format of JVM is added for comparison. While a so-called instruction is formed by an opcode and an operand, they are grammatically independent in case of HCgorilla and JVM. Because the concept of the machine instruction is vague as shown in Fig. 6, we do not say instruction set but code set in this study.

Architecture	Assembler code		Executable code
HCgorilla	Opcode	Media code (Java byte code)	1 byte
		Cipher code	
	Operand		1 byte
JVM	Java byte code		1 byte
	Operand		0-8 bytes

Fig. 6. Code format of HCgorilla.

The opcode of HCgorilla is a media or cipher code. The media codes are the subset of JVM. They operate for operand, stack, and data cache. A media code refers operands stored just after the opcode in instruction cache. The number of 1-byte operands related to a media code is variable between 0 and 2. Cipher codes function between register file and data cache. Cipher codes are SIMD mode to do streaming for multimedia data stored in register file. Cipher codes do not attach operands, but get cipher data from register file.

The policy of length variant reduced codes achieves the compactness of HCgorilla's code set. The length of an opcode and its related operands is made variable within instruction cache, which is effective to optimize cache size. The scheme of dense codes at instruction cache is also effective to reduce critical path delay and power dissipation. The compact scheme of HCgorilla is not similar to ARM's optimization scheme for encoding density. While the encoding density of ARM' ISA is simply concerned with the bit density of executable codes, dense codes at instruction cache referred to HCgorilla aim to pack executable codes as many as possible without increasing cache size. The compact code scheme is more preferable in view of mobility.

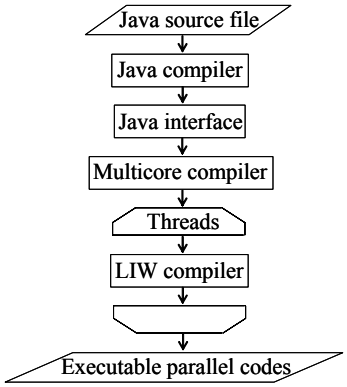


Fig. 7. Language process flow specific for HCgorilla.

Table 4. The target of API and the LIW compiler.

	Category	HCgorilla's code set		API	
		Serial	LIW compiler	Available	Next step
Java bytecodes	No operation		nop		
	Memory, stack access	iconst_ml, iconst_<i>, bipush, sipush, pop,pop2, dup,dup_x1, dup_x2, dup2, dup2_x1, dup2_x2	iload, iload_<n>; aload,aload_<n>; istore, astore, istore_<n>; astore_<n>	swap	aconst_null, lconst_<l>; fconst_<f>; dconst_<d>; ldc, ldc_w, ldc2_w, lload, fload, dload, lload_<n>, fload_<n>, dload_<n>; lstore, fstore, dstore, lstore_<n>; fstore_<n>, dstore_<n>; wide
	Type conversion				i2l, i2f, i2d, l2i, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f, i2b, i2c, i2s
	Arithmetic, logic, shift operation	iadd, isub, imul, ineg, ishl, ishr, iushr, iand, ior, ixor		iinc	ladd,fadd,dadd,lsub,fsub,dsub,lmul,fmul,dmul,idiv,ldiv,fdiv,ddiv,irem,lrem,frem,drem,lneg,fneg,dneg,ishl,lshl,lshr,iushr,lor,land,lxor
	Compare and branch		ifeq,iflt,ifle,ifne,ifgt,ifge,if_imeq,if_imne,if_imlt,if_imgt,if_imle,if_imge,goto		ifnull, ifnonnull, lcmp, fcmpl, fcmpg, dcmpl, dcmpg, goto_w, jsr, jsr_w, ret
	Method call, return			invokevirtual,invokespecial,invokestatic,invokeinterface,ireturn,lreturn,freturn,dreturn,areturn, return	
	Instance generation				new, newarray, anewarray, multianewarray
	Array access				baload,caload,saload,laload,faload,daload,aaload,bastore, castore,sastore,iastore,lastore,fastore,dastore,aastore,arraylength
	Field access				putfield,getfield,putfield,getstatic
	Table jump				lookupswitch, tableswitch
	Object				checkcast, instanceof, athrow, monitorenter, monitorexit
	Cipher	rsw, rlw			

3.3 Language processing

Fig. 7 abstracts language process flow specific for HCgorilla. Table 4 summarizes the target codes of API and the LIW compiler. Fig. 8 summarizes the Java interface flow specific for HCgorilla. The Java interface plays as API, and is composed of two components. The one abstracts Java bytecodes from a class file. The other unfolds Java bytecodes not defined by HCgorilla's code set. Since the output is still serial codes, it needs parallelizing compilers.

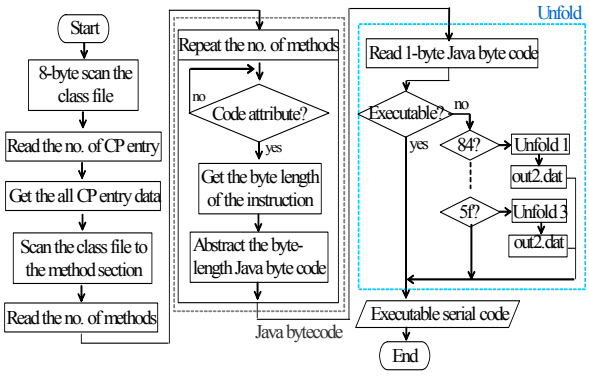


Fig. 8. Java interface.

4. COMPILATION TECHNIQUES

As shown in Figs. 4 and 5, HCgorilla's compilers are specified as follows.

a. To abstract parallelism from the Java interface output or executable serial codes.
b. To readdress codes newly produced by parallelization to avoid the conflict of data cache access.
c. To output the codes executable in parallel by HCgorilla.
d. To map parallel executable codes on instruction cache within a core. In mapping, jump codes are renamed by modifying their destination addresses.

In order to respond these steps, HCgorilla's compilers are composed of the multicore compiler and LIW compiler.

4.1 Multicore compiler

Table 5 summarizes threads of Java applications. They appear at various levels. Threads at source code level are abstracted by API. The target of this study is the abstraction of threads at executable level. This is covered by the multicore compiler. Fig. 9 shows the current status of the multicore compiler.

Table 5. Various threads of Java applications.

Thread	Language level	Granularity	Abstract means
MyThread	Source code	Large	API
Function			
Method	Executable code	↓	Multicore compiler
Library function			
Loop			
Others		Small	

TLP at executable level is judged by looking for return process that expresses the end of instructions sequence or thread. The multicore compiler does not readdress the second thread needed to avoid the conflict of data cache access with the first thread. The readdressing is covered by the hardwarily

logical share of data cache by the two cores. The readdressing of additional threads complements the logical share of data cache by the cores.

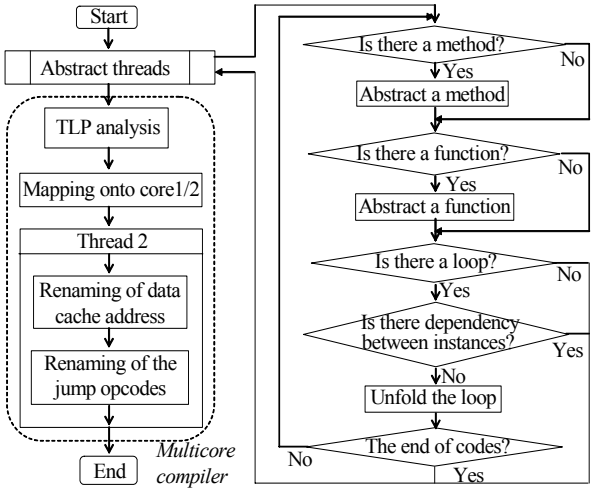


Fig. 9. Multicore compiler vs. threads.

4.2 LIW compiler

The LIW compiler abstracts ILP from a thread and does reorder, renaming, etc. ILP is judged by examining the conflict of data cache access. The conflict can be detected by checking the dependent operand of store and load instructions. Jump codes are excluded from ILP abstraction, because they directly affect program running. Fig. 10 shows the flowchart of the LIW compiler. LIW1 and LIW2 are working areas for a current code in examining code dependency.

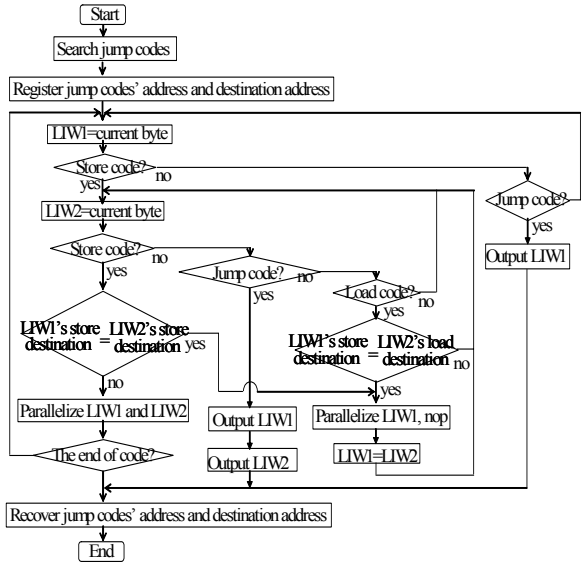


Fig. 10. LIW compiler flow.

4.3 Evaluation

The prototyping compilers have been written in Java as shown in Table 6. The total codes of multicore and LIW compilers are 300, respectively. By using a Java application shown in Fig. 11, the behavior of the parallelizing compilers is exemplified. The application is composed of two methods, each of which summarizes 0 to 1023. Each method is made a thread by the multicore compiler as shown in Fig. 12 (a). Fig. 12 (b) is derived by a hand compiler. Then, Fig. 13 (a) shows the decomposition of the first thread by the LIW compiler. Fig. 13 (b) shows the resultant form of the two threads. Although the comparison with the hand compiler is primitive, it shows the reasonability of the parallelizing compilers.

Table 6. Specifications of the parallelizing compilers.

	Descriptive language	No. of codes	Input	Output
Multicore compiler	Java	300	Java interface	Thread1, Thread2
LIW compiler	Java	300	Thread	Executable code

```
public class abcde
{
    void sum()
    {
        int i1,sum1,a1;
        sum1=0;

    for(i1=0;i1<1024;i1++){
            sum1=sum1+i1;
            a1=sum1+i1;
        }
    }
```

```
public static void
main(String args[])
{
    int i2,sum2,a2;
    sum2=0;

for(i2=0;i2<1024;i2++){
        sum2=sum2+i2;
        a2=sum2+i2;
    }
}
```

Fig. 11. A test program for parallelizing compilers.

5. DISCUSSION

In order to distinguish the potential aspects of the H/S system related to HCgorilla, the fundamental aspects of pervasive media, current status of information security and hardware security techniques are briefly discussed.

5.1 Pervasive Media

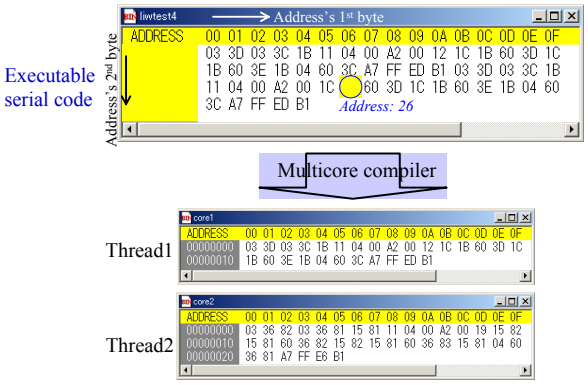
Table 7 surveys various aspects of pervasive media. They are classified in discrete and streaming media. Both types are expressed by byte structure. Discrete media is still useful in pervasive environment. Interactive games use many algorithmic processes for discrete data. Streaming media is more important because most pervasive computing applications owe to streaming media. This is further divided into two types in view of complexity. Text data is one of streaming data, because it is useful as refrain information in case of disaster. Considering endless data is hard to treat by mobile devices, the target of the HCgorilla system is discrete media and stream data. Yet, they need sophisticated and complicated process. Since streaming media is massive, it is reasonable to protect their security by common key, which is preferable to protect large quantity of byte-structured information.

5.2 Security

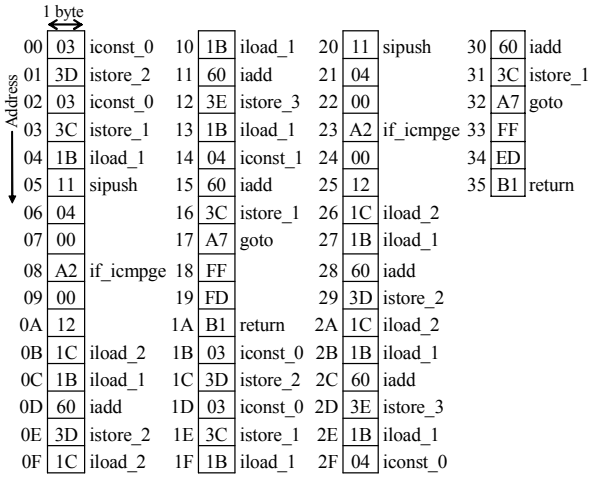
Table 8 surveys the current status of security techniques related to pervasive network. The one of hardware techniques at platforms is the public key applied to security chip, secure

coprocessor, cryptographic core, elliptic curve processor, etc. They ensure the front-end security of pervasive computing devices. These cutting-edge techniques are strong and effective for digital signing. However, it is not always reasonable in developing cipher streaming to take into account of only such public key module-embedded processors. Since one of most important aspect of cipher streaming is throughput, it is necessary to develop common key module-embedded processors. Actually, these are built in cryptography processors for IC cards and portable electronic devices. They implement common key ciphers as well as public key ciphers.

Compared with usual common key module-embedded processors, the hardware cryptography in this work has potential features. The prominent feature owes the multicore architecture to enhance throughput with less power. The technical feature is a byte-structured plaintext block. Since pervasive media is also byte-structured, and the block's length is set wider than usual ciphers, HCgorilla provides higher throughput. The academically new feature is the cipher transformation by random number addressing. This simplifies computational complexity and thus saves running time.

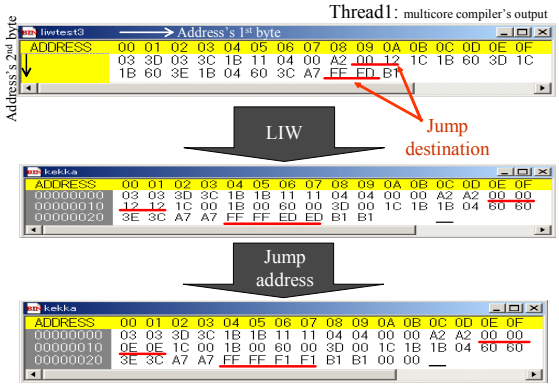


(a)

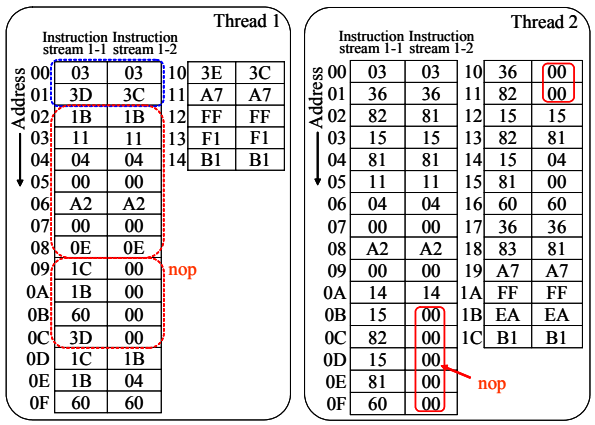


(b)

Fig. 12. Behavior of the multicore compiler. (a) The prototyping. (b) The hand compiler.



(a)



(b)

Fig. 13. Behavior of the LIW compiler. (a) The prototyping. (b) The hand compiler.

Table 7. Pervasive media.

		Complexity →		
Item	Discrete media	Streaming media		
		Stream data	Data stream	
Definition	Individual data	A sequence of similar elements	A sequence of data, which may be different each other	
Characteristic	Discrete	Stream of continuous media		
Size or quantity	Short	Long	Endless	
Example	Game, intelligent process	Text, audio, video	Seismography, tsunami, traffic	
Basic structure	Byte string			
Buffer storage	Respectable reregister file			
Data handling: Media	Algorithmic process	SIMD mode applications like signal processing, graphic rendering, data compression, etc.		
Data handling: Security	Public key	Common key cryptography		

Illegal attack such as tapping, intrusion and pretension are another issues of network security. Since they need complicated algorithms to detect and recognize individual phenomena, software techniques have been mainly used. However, they are not always sufficient from practical

viewpoint. The hardware implementation of IDS and IPS is our recent result exploited independently on this work [19].

Table 8. Network and hardware security techniques.

Device		Technique		Target	Running time	Secure-ness	Remarks
Mobile, ubiquitous	Platform	Hardware	Common key	Full text	Short	Practical	This work
			Public key	Password	Out of account	Large	Useful for digital signing
Internet			IDS, IPS	Full traffic	Short	Practical	Our another work
	Server	Software		Sampling	Large	Medium	Inflexible for individual demands

5.3 RAC

The random number-addressing cryptography, RAC has prospective features compared with regular block ciphers like AES (Advanced Encryption Standard) and stream ciphers like Vernam in view of running time, cipher strength, etc. RAC is applicable for any byte-structured multimedia data like text, audio, pixel, etc.

Table 9 summarizes the qualitative discussion on the characteristics of RAC vs. regular common key cryptography. Since quantitative measurement by using real chips and actual processors is hard at this point, we evaluate RAC's potential by examining algorithmic complexity, block structure, and cipher means.

Table 9. RAC vs. regular common key cryptography.

Cipher			Block		Cipher means		Running time	Cipher strength	Resource (cost)
			String unit	Length	Key (random numbers) length	Trans-formation			
RAC			Byte	As long as a buffer (register file's logical space) length		Needless (random number addressing only)	Medium	Practically (temporary or ad hoc) strong	Small
Regular	Vernam		Bit	Full length		Bitwise XOR	Short	Ideally strong	Large
	Stream	LFSR		A few bits or a character		Bitwise XOR	Medium	Medium	Small
		A5							
		AES-CTR		128 bits	1, 1.5, 2 times greater than the block length in case of AES	Bitwise XOR, scramble, shift, etc.	Long	Strong	Large
	Block	AES							
		DES		64 bits					

In view of algorithmic complexity, the dominant factor to determine running time is the number of iterative loops. RAC has only one iteration loop for blocks and has no iteration in transforming each block. To be accurate, the running time of RAC run on HCgorilla is the product of the total number of blocks and the sum of following factors.

a. t_1: the latency taken to transfer a block to the register file.
b. t_2: the time of a SIMD mode cipher operation.
c. t_3: latency taken to transfer a block from data cache.

On the other hand, AES has triple nesting loops. Except the 2nd loop for rounds, the 1st loop for matrix operation and the 3rd loop for blocks can be parallelized. The parallelism effectively reduces running time, but inevitably causes some tradeoff. Besides, such a discussion covering overall factors of software and hardware is very hard. Thus, we evaluate the running time in the case of normal condition of serial processing, and exclude the effect of double core.

Considering the ideal strength of Vernam cipher, the cipher strength is closely related to the key length. If an ideal buffer to immediately store a full text was available for HCgorilla, RAC could have the same strength as Vernam cipher. However, the practical buffer built in HCgorilla is a register file whose space and speed are actually limited. Yet, an adequate length register file makes the key length long, and thus the strength of RAC is expected to be practically strong. This exactly matches our goal that is not to achieve perfect strength but to provide ad-hoc encryption for pervasive devices. The strong strength of AES is mainly due to the iteration of a series of transformations.

Fig. 14 shows forwarding the cipher text to a recipient in cooperation with public key to safely transfer the initial key of RNG. It is treated by hyper protection. Encrypted initial key is a digital envelope. Decryption is similarly done by *rlw*.

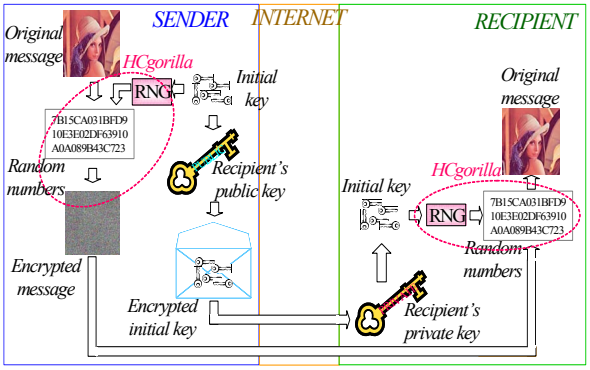


Fig. 14. Cooperation of RAC with a public key system.

6. CONCLUSION

Parallelizing compilers for HCgorilla composed of the multicore compiler and LIW compiler are specified to abstract parallelism from executable serial codes or the Java interface output and output the codes executable in parallel by HCgorilla. The prototyping compilers are written in Java. They are evaluated in case of an arithmetic test program. Although the evaluation is primitive, it shows the reasonability of the prototyping compilers compared with hand compilers.

The next step of our study will be as follows.

a. Improvement of the multicore compiler algorithm for TLP abstraction and core allocation.
b. Improvement of the LIW compiler algorithm for the enhancement of ILP abstraction, yield of smaller executables by register renaming, interprocedural optimization, procedural abstraction of repeated code fragments.
c. Detailed evaluation of the parallelizing compilers by using more test programs and Java benchmarks.
d. Matching the parallelizing compilers with the Java interface.
e. Installing the software support system composed of the Java interface and the parallelizing compilers in real web servers.

7. ACKNOWLEDGEMENT

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

8. REFERENCES

[1] D. Saha and A. Mukherjee, "Pervasive Computing: A Paradigm for the 21st Century," **Computer Magazine**, Vol. 36, No. 3, 2003, pp. 25-31.

[2] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," **IEEE micro**, Vol. 23, No. 6. 2003, pp. 46-51.

[3] M. Fukase, K. Shioji, N. Imai, D. Murakami, and K. Mikuni, "An Experiment in the Design and Development of a Multimedia Processor for Mobile Computing," **Technical Report of IEICE**, Vol. 102, No. 400 (DSP2002-130~137), 2002, pp. 13-18.

[4] M. Fukase, Y. Nakamura, R. Akaoka, and T. Sato, "Development of a Multimedia Mobile Processor," **Proc. of ISCIT2004**, 2004, pp. 672-677.

[5] M. Fukase, T. Oyama, and Z. Liu, "Endeavor in the Field of Random Sampling-Designing and Prototyping a Processor Suited for its Acceleration-," **Technical Report of IEICE**, Vol. 102, No. 272 (SDM2002-154, ICD2002-65), 2002, pp.7-12.

[6] M. Fukase and T. Sato, "Power Conscious Endeavor in Processors to Speed Up Random Sampling", **Proc. of SCI2003**, Vol. V, 2003, pp. 111-116.

[7] M. Fukase, H. Takeda, R. Tenma, K. Noda, Y. Sato, R. Sato, and T. Sato, "Development of a Multimedia Stream Cipher Engine," **Proc. of ISPACS 2006**, 2006, pp. 562-565.

[8] M. Fukase, A. Fukase, Y. Sato, and T. Sato, "Cryptographic System by a Random Addressing-Accelerated Multimedia Mobile Processor," **Proc. of SCI2004**, 2004, pp. 174-179.

[9] M. Fukase and T. Sato, "Low Energy Digital Electronics for Multimedia Ubiquitous Environments," **Proc. of EIC'05**, 2005, pp. 409-414.

[10] M. Fukase, R. Akaoka, and T. Sato., "Hardware Cryptography-Embedded Multimedia Mobile System," **Proc. of WMSCI2006**, 2006, pp. 225-230.

[11] M. Fukase, H. Takeda, and T. Sato, "Hardware/Software Co-Design of a Secure Ubiquitous System," **Computer Intelligence and Security**, Springer Berlin/Heidelberg, LNCS Vol. 4456/2007, 2007, pp. 385-395.

[12] D. Geer, "Chip Makers Turn to Multicore Processors," **Computer Magazine**, Vol. 38, No. 5, 2005, pp. 11-13.

[13] Y. Lin, C. Kozyrakis, Ali-Reza Adl-Tabatabai, "Multicore Programming: From Threads to Transactional Memory," **HOT Chips 18**, 2006.

[14] M. Fukase, R. Egawa, T. Sato, S. Itoh, and T. Nakamura, "Performance Evaluation of Wave-Pipelines and Conventional Pipelines," **Technical Report of IEICE**, Vol. 101, No. 386 (DSP2001-110, ICD2001-115, IE2001-94), 2001, pp. 1-8.

[15] M. Fukase, T. Sato, R. Egawa, and T. Nakamura, "Scaling up of Wave-Pipelines," **Proc. of the Fourteenth International Conference on VLSI Design**, 2001, pp. 439-445.

[16] M. Fukase and T. Sato, "Design of a Hardware-Cryptography-Embedded Processor for Pervasive Computing," **Proc. of UCAS-3**, 2007, pp. 15-22.

[17] M. Fukase and T. Sato, "Innovative Ubiquitous Cryptography and Sophisticated Implementation," **Proc. of ISCIT2006**, 2006.

[18] D. S. Kochnev and A. A. Terekhov, "Surviving Java for Mobiles," **IEEE pervasive COMP.**, Vol. 2, No. 2, 2003, pp. 90-95.

[19] T. Sato, D. Miyamori, R. Sakuma, and M. Fukase, "Power-Consumption Aware Intrusion Detection Logic for WLAN," **Proc. of WMSCI2005**, Vol. III, 2005, pp. 409-414.