JOURNAL OF KOREA MULTIMEDIA SOCIETY, VOL. 9, NO. 12, DECEMBER 2006(pp. 1617-1627)

Efficient Native Processing Modules for Interactive DTV Middleware Based on the Small Footprint Set-Top Box

Sang-Myeong Shin[†], Dong-Gi Im[††], Min-Soo Jung[†††]

ABSTRACT

The concept of middleware for digital TV receivers is not new one. Using middleware for digital TV development has a number of advantages. It makes it easier for manufacturers to hide differences in the underlying hardware. It also offers a standard platform for application developers. Digital TV middleware enables set-top boxes(STBs) to run video, audio, and applications. The main concern of digital TV middleware is now to reduce its memory usage because most STBs in the market are small footprint. In this paper, we propose several ideas about how to reduce the required memory size on the runtime area of DTV middleware using a new native process technology. Our proposed system has two components; the Efficient Native Process Module, and Enhanced Native Interface APIs for concurrent native modules. With our approach, the required memory reduced from 50% up to 75% compared with the traditional approach. It can be suitable for low end STBs of very low hardware limitation.

Keywords: Interactive DTV, DTV Middleware, Java Virtual Machine, Java Native Interface

1. INTRODUCTION

Millions of people watch DTV every day, and this number is growing fast as more network operators and governments see the benefits of digital broadcasting. In recent years, interactive digital television has become the next big thing for the broadcasting industry as broadcasters and network operators seek new ways of making money and keeping viewers watching. The development of efficient DTV middleware is essential for this reason. Using middleware for interactive TV has a number of advantages. It makes it easier for manufacturers to hide differences in the underlying hardware, which in turn makes it easier for network operators to buy receivers from more than one vender. It also offers a standard platform for application developers[1-4].

Java Virtual Machine is one of the layers in the system software of STBs. The environment of Ditital TV makes hard to perform Java virtual machine efficiently because of the limitation of STBs hardware resource. It is necessary that we have to improve performance and reduce memory requirement of middleware working on the low-end grade STBs[5-9].

The traditional Digital TV's set-top box needs a hardware requirement such as 166MHz CPU, 8MB flash memory, and 16MB RAM to execute the middleware system and applications based on DVB-MHP. However, the common preference of the STBs market is low hardware specification than traditional one. Even though we create the native modules that help to improve the speed of

※ Corresponding Author : Min-Su Jung, Address : (631-701) 449 Wolyoung-dong, Masan, Kyungnam, S.Korea, TEL : +82-55-242-7432, FAX : +82-55-248-2554, E-mail : msjung@kyungnam.ac.kr

Receipt date : Nov. 1, 2006, Approval date : Dec. 20, 2006

[†] Dept. of Computer Engineering, Kyungnam University (E-mail : aegisddd@kyungnam.ac.kr)
[††] Dept of Computer Engineering, Kyungnam University (E-mail : toodulli@kyungnam.ac.kr)
[†††] Dept of Computer Engineering, Kyungnam University

※ This work was supported by Kyungnam University research fund, 2006.

middleware system, the native module doesn't support concurrent execution so it will make all middleware system can be slow[10-14].

For the some reason that we mentioned before, we propose advanced technologies of the digital TV middleware that can perform efficiently on low-end grade STBs which have very low hardware limitation; the Efficient Native Process Module and enhanced java native module for the concurrent native process.

This paper is organized as follows. Section 2 describes about DVB-MHP, STB as related works in detail. Section 3 gives a design about how to reduce memory requirement of DTV middleware with the Efficient Native Process Module, the JNI process APIs of concurrent native modules. Section 4 gives the performance results. Finally, we present the conclusion and the future work in section 5.

2. A RELATED RESEARCH

2.1 DVB-MHP Systems

In the world of digital television, the most obvious example to anyone using a digital TV set or digital set-top box (STB) are the enhanced and interactive TV applications now becoming available. In the future, a major use of data broadcasting will be as an enabler for interactive TV (iTV) that will also include links to the internet[15].

The DVB Project was founded in 1993 to establish a framework for the MPEG-2 based digital television services. The DVB System provides an intermedium for delivering MPEG-2 Transport Streams via a variety of transmission media. The DVB Project released the MHP specification in February 2000. The architecture of the MHP is defined in terms of three layers: resources, system software and applications. The MHP system layers are presented in Fig. 1. MHP architecture defines three layers consisting of resources, system software and applications. MHP resources include MPEG processing, Input/Output devices, CPU, memory and graphics system. The system software utilizes the underlying resources to provide an abstract view of the platform to the applications. Java language has been chosen for the development of MHP applications, which are platform independent in nature. The system software mainly consists of operating systems, JVM and software package that provide necessary Application Programming Interface (APIs) for the Digital TV application development[16-19].

Although Fig. 1 only shows one possible way APIs can be built on top of one another, it gives us an idea of the dependencies among the various components. The APIs that make up MHP and OCAP can be split into two main parts. One part contains the components related to MPEG and MPEG streams. The other part provides services

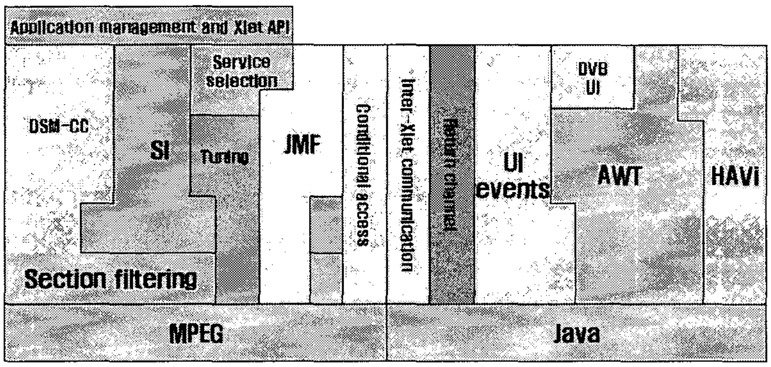


Fig. 1. Overview of the components in an MHP software stack.

built directly on top of the standard APIs that are a part of every Java platform. Important API for handling MPEG is the section-filtering component, used to filter packets from the MPEG stream. Almost all of the other MPEG-related APIs build on this in some way. The service information component uses it to filter and parse the MPEG sections that contain the SI tables required to build its SI database, which applications can then query using the two available service information APIs. The SI component could use a proprietary API for a accessing the section filters it needs, but in some designs it may be equally easy to use the standardized section-filtering API. So, we designed native module implementation to increase speed of this part[20,21].

MHP defines three different profiles and each one requires a certain minimum set of features. The simplest profile, Enhanced Broadcasting, enables downloading of Java applications with local restricted interaction. The next profile, Interactive Broad-casting, requires an interaction channel and presents the MHP-HTML applications, but as an optional feature. The most advanced profile, Internet Access, is intended for accessing Internet services. By dividing the MHP standard into these profiles the manufacturers can quickly provide clients with simple MHP receivers.

2.2 STB architecture

STB is the device used to receive the broadcasted signal in the digital form and convert into the television understandable form. The system software of the MHP complaint STB contains a layer of JVM on which Digital TV applications run.[16,17]

Table 1 shows the specification of set-top box types in the STBs market.

Set-top box is different from traditional desktop PCs. Current and upcoming set-top box hardware provides very limited storage capabilities and resources.

3. THE DESIGN AND IMPLEMENTATION

3.1 A typical native module between Java thread and Native thread in the DTV middleware

The architecture of a typical DTV middleware is composed of a thin native module and a thick middleware stack. The management of the program specific information table is important in the middleware. The efficient management of the information causes good system performance. The TS process module and SI DB are located in a middleware stack of the STB architecture. This middleware stack is implemented by Java language and native language. Although the middleware is simplistic and has portability, it has a problem in its processing memory requirement. A virtual machine commonly needs to call various native functions to perform I/O, access external resources, etc. In a VM that uses non-native threads, the rest of the VM is blocked when a native function is called. There is only one physical thread of control. If the native function does not return immediately, all the other threads in the VM are blocked indefinitely until the native function returns. In the worst case, the entire system may deadlock. We need the enhanced java native interface for fast middleware. Fig. 2 shows the algorithm of a traditional native processing module in the middleware.

Table 1. Comparison of Set-top box types in the market

Division	CPU	ROM	DRAM(Bit)	Graphic
High End	150Mhz ~	32M ~	16M ~	3D
Midrange	80~130Mhz	16M	8M	MPEG chip with a built-in Graphic
Low End	50Mhz	8M	4M	MPEGE chip with a built-in Graphic

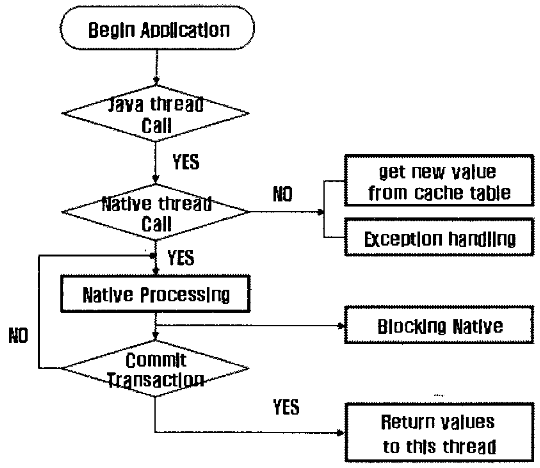


Fig. 2. The algorithm of a traditional native processing module in DTV middleware.

Because native thread was blocked over this mechanism, other Java threads can't enter into native thread area. The processing time of the middleware increases if errors or delays happen in the native thread. The solution of this problem is essential. To overcome the above problem, traditional middleware must create another virtual machine that will execute native thread stand-alone. The memory requirement will increase double. Finally, java virtual machine will be allocated too much memory on it. In addition, if the native thread will not return any value to the original virtual machine, it can be possible, otherwise the native thread has to know all information about the original virtual machine to return values. Traditional middleware however doesn't support this kind of method.

3.2 Our concurrent native processing technology between Java thread and Native thread

When an EPG, Xlet or applications are executed on JVM, they need the information such as video/audio, data and PSI. The system software of the MHP contains a layer of JVM on which Digital TV applications run.

Since Digital TV applications are developed using Java language, they are highly portable across many STBs. JVM layer provides the luxury of write once and run anywhere for the bytecode obtained from the compilation of Java applications. Portability and customization require the STB's applications to be developed using Java programming language. The major point of criticism with regard to Java for DTV middleware is its small footprint such as low memory and slow CPU. Programs written in Java are translated into Java Bytecode by a compiler. Even with a 32-bit processor, the execution speed of Java Bytecode executed by an interpreter is 10 to 20 times slower than program code written in C. Therefore, we have to use both the native method and Java Bytecode for efficient middleware.

Service information is such a central part of the middleware. It is important to many of the other components in an MHP or OCAP receiver. Because MHP and OCAP are both Java-based middleware stacks, that may be a reason to lean toward a Java implementation of the SI database. At the same time, if the SI database is getting updated or queried frequently it is better to use native code because the underlying section filtering and basic table parsing will also be carried out in the native code.

The main problem that has been observed is the inefficient native thread access of the Java applications. Fig. 3 shows architecture of our DTV middleware in the STB. It consists of thin middleware and a high-speed native module. It guarantees rather fast speed because the native module manages SI DB.

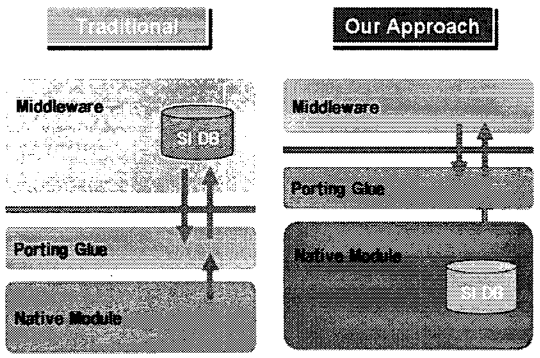


Fig. 3. Our STB Architecture compared with traditional ones.

Fig. 4 shows the algorithm of our native processing module in the middleware. This module manages java threads so that a process can call native threads at the same time, and provides necessary APIs. It stores current thread information and reads data, data length and so on. If a native process is over, this module refers to stored object information and transmits the process results to java thread. A concurrent native process is possible through this mechanism.

3.3 Design of Java native interface APIs for concurrent native modules

When an object is moved, only the object handle must be updated with the new location. All references to the object in the executing program will still refer to the updated handle, which did not move. While this approach simplifies the job of heap defragmentation, it adds a performance overhead to every object access.

We designed some enhanced java native interface APIs. These APIs manage the information of threads with memory management modules, pointing to which native method has been called.

Core modules are as follow:

- ***JNI_getCurrentThread*** : It can get the threads ID which called by native function, and can be called by only native function and wakeup function.
- ***JNI_resume_Thread*** : Due to resume threads which specified by unique thread ID, This function registers java thread ID to JVM Ready queue. When the thread is executed by scheduler, and when wakeup function has been registered, the thread is executed after wakeup function run.
- ***JNI_suspendThread*** : It can suspend the java thread using thread ID that you want to stop it.

Fig. 5 shows Concurrent Native Process Protocol of Digital Television System. First, java application thread calls a native module and stops the thread. Second, it calls an event handler and adds events to different java threads. Third, native module sends a return value to the event handler. It takes out event and resumes previous thread. Last, it calls wakeup function to resume threads.

3.4 Our efficient middleware technology

Digital TV not only provides high-definition and

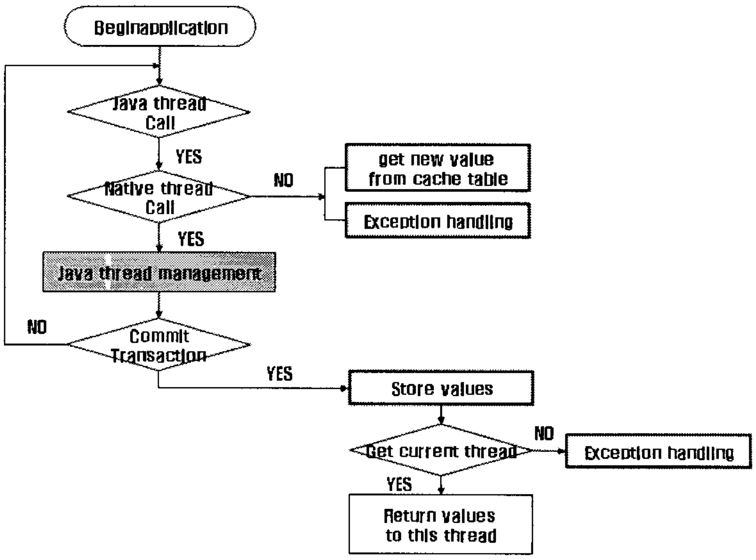


Fig. 4. The algorithm of our native processing module in DTV middleware.

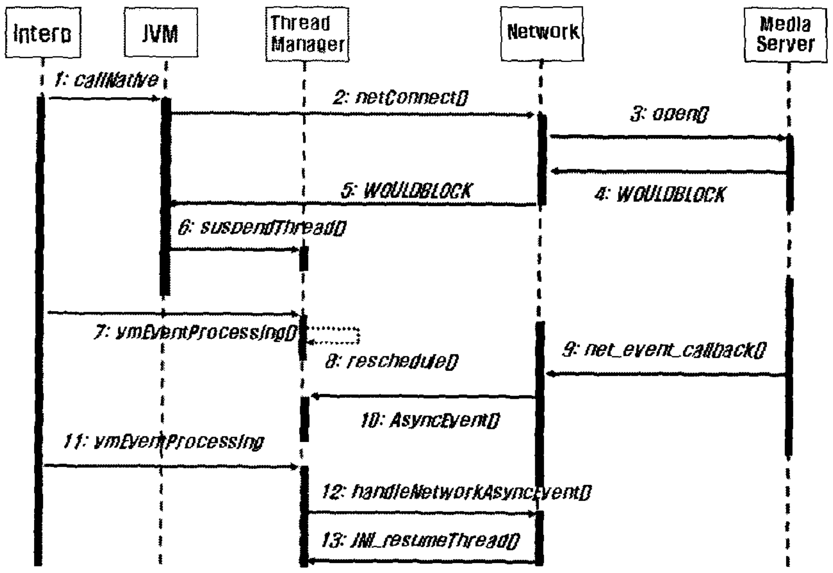


Fig. 5. Concurrent native process protocol of digital television system.

multi-channels, but also provides application programs. This kind of additional service can be provided by broadcast networks including video and audio which can be downloaded within set-top box. Depending on the application program and digital TV set-top box, it also can provide interactive services using return channels.

However, information that transported by MPEG-2 TS is not suitable for all digital TV set-top box. There being so many broadcasting protocols, it is slow and difficult to work. To solve those problems, we designed four modules to reduce the required memory size on the runtime area of DTV middleware using a new native process technology; Data Cache Manager, Packaging Module, Monitoring Module, Retrieving Module. All modules require a native processing module to improve a processing speed and to reduce the required memory size. Transport Streams consist of Media data, SI data, and Xlet data. Each type of data has a PID which is already defined. And it is inter-leaved together into a stream. Section Filtering and SI DB must be executed quickly and efficiently. This is the reason why we put the native module in between H/W and java middleware.

Data Cache Manager controls the information of Java Middleware. It maps the raw data of TS Reader module to Object types, which DVB/Java-TV API requests. It also stores the created objects for an amount of time, collect data request of other components, and transmits data fitted for the request. This process designed native module to improve an execution speed as well.

Fig. 6 shows architecture of Section Data Cache Manager. This module is made up of two parts: Section Data Parser and Section Data Cache. Java

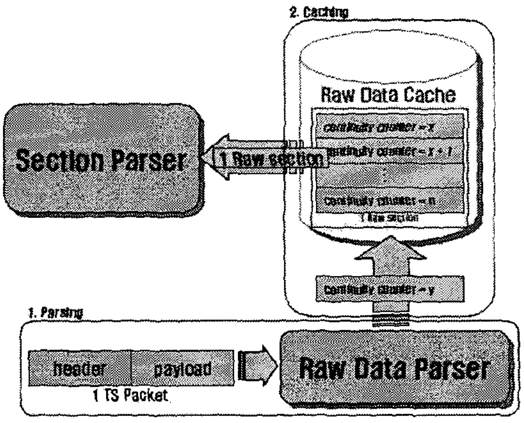


Fig. 6. The procedure of Raw data cache manager process.

thread uses all of the table information in the Section cache. All of the tables in the Section cache want to call native thread concurrently. When any native thread runs, java virtual machine however has a stop state. The concurrent process module does apply to Data Queue and Raw Section module.

In our algorithm, a native module always must be used-typically for the efficient execution. The situation can be better in case of the much more expensive write operation. The performance in this situation is better than that of independent of platform in the DTV middleware. As mentioned earlier, Java thread cost is more expensive than concurrent process of native module. Fig. 7 and 8 describe more detail about prcedure of TS reader module and retrive process.

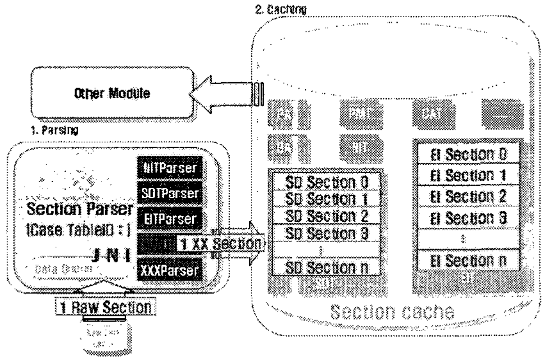


Fig. 7. The procedure of Transport Stream Reader module.

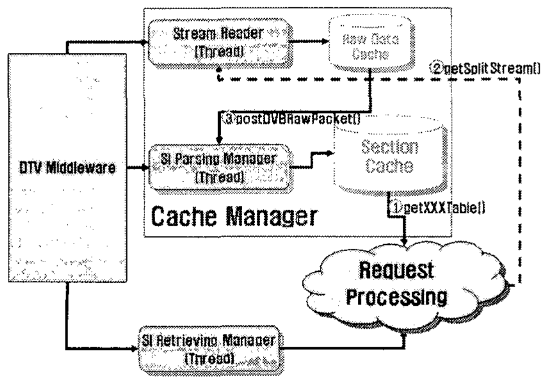


Fig. 8. The procedure of retrieve process.

4. EVALUATION OF OUR APPROACH

Fig. 9 shows a evaluation board to test of our new approaches. The CPU on the board is the product running on Linux operating system developed out of Intel XScale IXP 425 400MHz and TI DSP DM642 600MHz.

We examined our approaches with concurrent native program and non-blocking program. We tested our algorithms in terms of enhanced the total memory requirement and the used memory. Fig. 10 shows the result of execution of concurrent native module.

The test-bed is divided to four tasks to reduce the required memory size on the runtime area of DTV middleware using a new native process technology; Data Cache Manager, Packaging Module, Monitoring Module, and Retrieving Module. Media Server transmits data to set-top box. HAL and Native module deal with this data. And middleware transmits data to java applications. It is composed

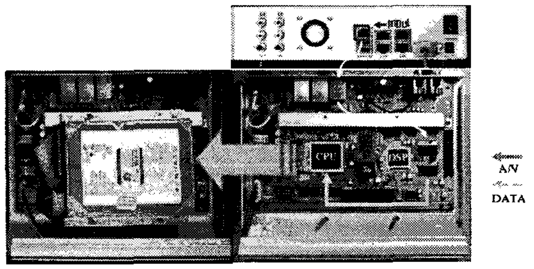
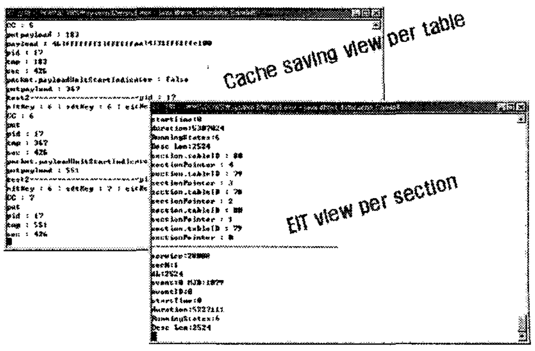


Fig. 9. The inside of NSTB-LX2000 set-top box.



```
CC : 6
getpayload : 183
payload : [illegible]
pid : 17
tmp : 183
sec : 426
packet.payloadUnitStartIndicator : false
getpayload : 367
test2---------------------------pid : 17
nitKey : 6 : sdtKey : 6 : eitKe
CC : 6
pat
pid : 17
tmp : 367
sec : 426
packet.payloadUnitStartIndicato
getpayload : 551
test2---------------------------pi
nitKey : 6 : sdtKey : 7 : eitKe
CC : 7
pat
pid : 17
tmp : 551
sec : 426
```

```
startTime:0
duration:5307024
RunningStatus:6
Desc Len:2524
section.tableID : 80
sectionPointer : 4
section.tableID : 79
sectionPointer : 3
section.tableID : 78
sectionPointer : 2
section.tableID : 80
sectionPointer : 1
section.tableID : 79
sectionPointer : 0
------------------------------------
service:20000
secN:1
dL:2524
event:0 MJD:1077
eventID:0
startTime:0
duration:5327111
RunningStatus:6
Desc Len:2524
```

Fig. 10. The window contains the execution of concurrent native module.

to several threads. Java applications request data to a lot of threads of middleware.

Table 2 shows the comparison both the traditional native process and our new native process in regard to memory usages.

We used the part of our digital television middleware for this evaluation. There are three threads. They use transport stream information, and a native thread to get transport stream information. Native module must solve the error or delay efficiently. We define three cases of transport stream.

Table 3 shows the results that compare to the total memory requirement. In this evaluation, we compare a traditional approach and our new one, checking the requirement of total memory on the java virtual machine, when the number of tasks is increasing.

As you can see the results in Table 3, the amount of total memory requirement is started on the same value when the number of tasks is one. During the number of tasks is increased 2 to 4, you can find differences clearly. As we mentioned before, the total memory requirement of the traditional approach increases in a fan shape because java virtual machine is created by each task, whereas our approach is to create threads with one java virtual machine for program execution. Therefore, it doesn't increase total memory requirement and it maintains the amount of memory allocation which allocated at first time. In the result as follow, we can see the reduction of our new approach; the maximum memory reduction is up to 75% compared with two approaches.

Table 2. The evaluation of an execution in a traditional native process and our one, the part of our digital television middleware for its evaluation

```
readerRun = new Thread(reader);
readerRun.start();
parserRun=new Thread(SIParsingManager.getInstance());
parserRun.start();
requestRun = new
Thread(SIRetrievalManager.getInstance());
requestRun.start()
------------------------------------------
JNIEXPORT void JNICALL
Java_native_readPMT(JNIEnv *env, jobject nt){
   syncByte = readByte();
   if(ret = WOULDBLOCK){
      dt[idx].readThread = JNI_getCurrentThread();
      JNI_suspendThread();
   }
}
readByte( ){
   ret = transport_stream_read();
}
```

Table 3. The comparison of total memory requirement

	Traditional	Our approach	Reduced Rate
1 Task	4,194 KB	4,194 KB	0 %
2 Task	8,388 KB	4,194 KB	50 %
3 Task	12,582 KB	4,194 KB	67 %
4 Task	16,777 KB	4,194 KB	75 %

Fig. 11 shows the comparison of total memory requirement in a traditional native process and a newly changed native module with enhance concurrent native process technology.

The total memory is a whole memory area allocated on the java virtual machine. In Fig. 11, the traditional approach is in totally different memory allocation compared with our new approach because the traditional approach allots the same number of tasks with the number of java virtual machine

Another important data in this evaluation with the total memory requirement is the used memory. The used memory means the area of memory for operation; a part of the total memory when the java virtual machine executes tasks. The used memory also can be increased or decreased at different tasks. In this evaluation, we measured memory usage to go along with task increasing both the traditional approach and our one.

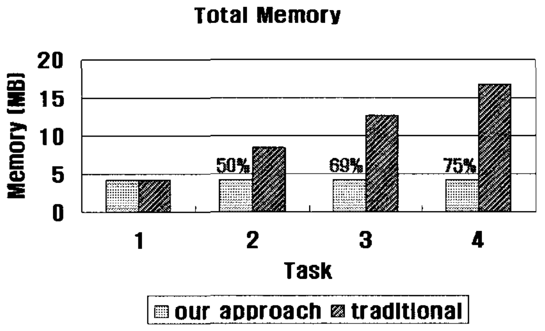


Fig. 11. The chart shows the advantages of our system about memory requirement.

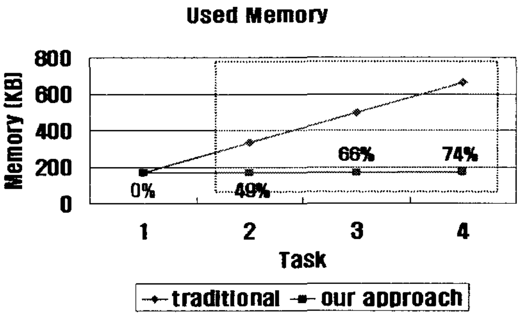


Fig. 12. The chart shows the advantages of our system about memory requirement.

Table 4 shows the real amount of memory use when tasks are executed by java virtual machine. In case of the traditional approach, the used memory is increased up to 394% during the number of task is growing. It explains that the individual memory area is allocated for each task when a task is added. Unlike the traditional approach, the used memory of our new approach executing task on the java virtual machine is increased up to 4Kbyte to go along with task increasing. Even during all four tasks are running, the used memory is increased 2% compared with one task. This is very small size of memory use compared with the traditional approach executing four tasks, 4 times bigger than this.

Fig. 12 expresses Table 4 with a graph. If you Fig out the inside of the box, you can easily understand what the difference of between the traditional approach and our new one to go along with task increasing.

As you see, in Table 4 and Fig 12, the gap of reduced rate between the traditional approach and our approach is of from 49% up to 74%. This some figures explain that the traditional approach has been on the increase in the memory use because each task has its own memory management system including the system memory of java virtual machine and all kinds of basic loaded classes. On the other hand, our approach hasn't been on the big increase in the memory use. When the number of tasks is increased, java virtual machine creates thread instead of another virtual machine, so it doesn't have to allocate system memory for another virtual machine, but it just needs to allocate memory size of another thread.

Table 4. Comparison of used memory usage

	Traditional	Our approach	Reduced Rate
1 Task	168,404 Byte	167,232 Byte	0 %
2 Tasks	332,668 Byte	168,320 Byte	49 %
3 Tasks	497,332 Byte	169,440 Byte	66 %
4 Tasks	662,116 Byte	170,752 Byte	74 %

According to the prior statements, it is very clear to know that our approach reduces memory use a lot. As we explain before, tasks can be executed concurrently on only one java virtual machine process up to four tasks. According to the result, we can reduce much smaller amount of the used memory than traditional approach that each task has its own virtual machine. In order to use the new approach, we have to set up the concurrent environment, the traditional approach doesn't support it.

In addition, a part of the real memory use by the total memory requirement, the traditional approach reported rates of 39.5% in memory use, but our analysis from the result data reported rates of 4% in memory use by total memory requirement that executes the same task as traditional one.

5. CONCLUSION

Java virtual machine (J2ME) technology is already a standard for digital TV on embedded devices. A Java language is basically slower than other languages. The set-top box platforms also have a heavy hardware limitation. In spite of Java's slow speed, the reasons why Java virtual machine technology is selected as a standard is platform independence. Therefore, DTV middleware uses native module together for fast process speeds. At this time, an enhanced native module is necessary.

In this paper, we have proposed the method to reduce the storage and resource of native thread module. It used Java native interface with a concurrent native process module and an efficient Java native interface module. It also makes the used memory of native thread module in the DTV middleware. Our approach to resource-constrained, the required memory reduced from 50% up to 75% compared with the traditional approach. It can be suitable for low end STBs, very low hardware limitation. In the future, this mechanism will contribute greatly to the performance of DTV middleware.

6. REFERENCES

[1] Edward M, Schwalb, *Technologies and Standards, ITV Handbook*, Prentice Hall PTR, July 2003.

[2] ETSI EN 301 192 v1.4.1, Digital Video Broadcasting (DVB); DVB specification for Data Broadcasting, ETSI, Nov. 2004.

[3] ETSI TS 102 812 v1.1.1, Digital Video Broadcasting (DVB); Multimedia Home Platform (MHP) Specification 1.1, ETSI, Nov. 2001.

[4] Steven Morris, *Anthony Smith-Chaigneau. Interactive TV Standards*, Focal Press, 2005.

[5] *Connected Device configuration and Foundation Profile*, Version 1.0.1 Java 2 Platform, Micro Edition. Porting Guide, 2002.

[6] ETSI EN 300 468 V1.6.1: Digital Video Broadcasting (DVB) - Specification for Service Information (SI) in DVB systems, 2004.

[7] Iain D. Craig, *Virtual Machines*, Springer, 2005.

[8] Java 2 Platform, Micro Edition (J2ME technology) at http://java.sun.com/products/j2me

[9] Multimedia Home Platform (MHP) at http://www.mhp.org

[10] Grzegorz Czajkowski, "Application isolation in the Java Virtual Machine," *ACM Press*, pp 354-366, 2000.

[11] Grzegorz Czajkowski, Laurent Daynes, and Mario Wolczko, "Automated and Portable Native Code Isolation," *Sun*, 2001.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[13] Lizy Kurian John, Lieven Eeckhout, *Performance Evaluation and Benchmarking*. CRC, 2005.

[14] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (2nd Edition)*, The Java Series, Addison-Wesley Pub Co, 1999.

[15] G.Sivaraman, P.Cesar, and P.Vuorimaa. "System software for digital television applications," *In Proc. The IEEE Int. Conf. on Multimedia and Expo*, ICME2001, pp. 784-787, Aug. 22-25 2001.

[16] *ATSC data broadcast standard*, ATSC Standard A/90, 2000.

[17] Bill Venner, *Inside the Java Virtual Machine. The Java Masters Series*, Computing McGraw-Hill, 1998.

[18] ETSI TR 101 211 V1.6.1 : Digital Video Broadcasting (DVB) - Guidelines on implementation and usage of Service Information (SI), 2004

[19] *Program and system information protocol for terrestrial broadcast and cable (Revision B)*, ATSC Standard A/65B, 2003.

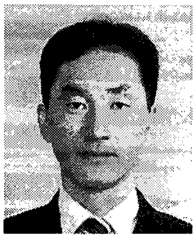
[20] R.Radhakrishnan, N. Vijaykrishnan, L.K.John,

and A. Sivasubramaniam, "Architectural issues in java runtime systems," *In Sixth International Symposium on High-Performance Computer Architecture*, pp 387-398, Jan. 08-12 2000.

[21] Transport stream file system, ATSC Standard A/95, Feb. 2003.

Sang-Myeong Shin

He received B.S. degrees both in Electronic Engineering and in Computer Engineering from Kyungnam University, S. Korea, in 2005. He is currently a M.S. candidate in the Dept. of Computer Engineering in 2007. His research interests include home network, embedded system management, java virtual machine and RFID.



Dong-Gi Im

He received the B.S. and M.S. degree in Computer Engineering from Kyungnam University in 2001 and 2003, respectively. and the Ph.D. degree in Computer Engineering from Kyungnam University in 2007. He has been working at Digital Homenet since 2001. He is currently a chief researcher in the Digital Homenet. His research interests include java virtual machine, radio frequency identification and embedded System.

Min-Soo Jung

He received the B.S. degree in computer engineering from Seoul National University in 1986, and the M.S. degree in computer since from KAIST in 1988, and the Ph.D. degree in computer science from KIDST in 1994. He has been working at Kyungnam University since 2000. He is currently a professor in the division of computer engineering. His research interests include java technology, home networking, mobile programming, and USIM technology.