An OBSM Method for Real Time Embedded System

Liu Fang, Ming Cai, Jinxiang Dong, Meirong Xu
Institute of Artificial Intelligence, Zhejiang University, Hangzhou, China, 310027
{fsx, cm, djx, xmr } @cs.zju.edu.cn

Abstract

The traditional OBSM method in spacecraft and other real time embedded system is to re-compile the whole system after source code modification, and then reboot the target machine with the new version software. This hurts the availability of the system. Most of the new methods are based software or hardware redundancy. To reduce the cost and make the OBSM more conveniently, we classify OBSM, and patch the target system with different method according to different OBSM type, so the new solution can reduce the cost.

Keywords: OBSM, real time, embedded, availability-critical

1 Introduction

With recent generations of spacecraft, complex onboard software has emerged as a key component between the ground control facilities and the spacecraft platform and payload being operated, by offering a high level of spacecraft autonomy and flexibility. This software can be modified or even redefined during the mission, whereas most other on-board subsystems are confined within the limits of pre-defined configurations. [1] Although instrument on-board software is designed, developed and tested following strict quality assurance procedures, experience of past and current missions show that the capability of reprogramming instrument on-board software from the ground is an essential requirement throughout the instrument lifetime. [2]

To ensure safe operations and service continuity, the On-Board Software Maintenance (OBSM) concept that will be employed allows in-flight modification of the running software, through an incremental model based on patches. New programs are compared to the current software on board and the differences are converted into memory-load commands. Four independent versions of the on-board software will be maintained - one per spacecraft - with multi- and inter-spacecraft handling for the common functionalities. [3]

Exclude aerospace, the demand for continuous service in mission- and availability-critical software applications, such as Internet infrastructure, telecommunication, military defense and medical applications, is also expanding. The evolutionary change of software is unavoidable due to changes in the environment or in the application requirements that cannot be completely predicted during the design phase, or due to bug-correction or enhancement of functionality. For these availability-critical applications, it is unacceptable to shutdown and restarts the system during software upgrade, but in many real time embedded system, it needs to re-compile the whole system and then reboot the target machine with it. The objective of on-board software maintenance is to be able to add, remove or replace any relevant components without significantly affecting other parts of the system.

The remainder of the paper is organized as follows. Section 2 discusses some related works of other researchers. Section 3 describes the framework of the OBSM system. Section 4 describes the implementation of the system, followed by section 5 presents a prototype of the system. The final section concludes the effort of our work.

2. Related work

This section discusses selected approaches to the problem of OBSM systems:

Upgrading need system reboot• •

Most of the embedded system applications are compiled with the embedded operating system, so the system will reboot after maintenance in the traditional OBSM solution.

An JunShe and his partners proposed an approach to reduce the size of the patch and save the time of writing the patch to flash [4]. In their approach, they allocate more memory for each module at compiling time. When maintaining software, the module which need modified is replaced by the new one, and then system is rebooted. But it is hard to estimate the extra size that is needed to allocate for each module, and the memory is wasted.

C. Steiger R. Furnell and J. Morales describe an approach of using On-board Control Procedures (OBCPs) to do On-board Software Maintenance (OBSM) activities in a safer and more efficient manner [5]. An OBCP is a stand-alone program executed on-board and capable of

1-4244-0165-8/06/$20.00 ©2006 IEEE.

Authorized licensed use limited to: University of Florida. Downloaded on February 12, 2010 at 18:30 from IEEE Xplore. Restrictions apply.

interacting with other on-board subsystems, with similar capabilities to what a ground operator would have, so problems of the on-board subsystems would not affect the OBCP. But with their design of OBCP, the system needs to reboot likewise.

Upgrading needs no system reboot• •

Ann. Tai et al, supported by Jet Propulsion Laboratory, do a series of researches on OBSM [6-9]. In their solution, with hardware redundancy, the new version software runs concurrently with the old version software. The old version guards the upgrading, and if the new version works as expected for some time, the system switch to the new version from old version, else if errors occur when upgrading, the system rollback to a checkpoint. Their work avoids or minimizes the unavailability and performance loss of spacecraft/science functions due to software upgrading activities and due to system failure caused by residual faults in an upgraded version, but needs hardware redundancy.

Component-based dynamic architecture: Darwin, proposed by Jeff Kramer and Jeff Magee, is a configuration language for describing dynamic architecture [10]. It is a declarative language, which is intended to specify the structure of distributed systems composed from diverse components using diverse interaction mechanisms. It separates the description of structure from that of computation and interaction. C2-style architecture is another component-based architecture, which highlights the role of connectors in supporting runtime change [11][12]. Connectors are explicit entities that bind components together and act as mediators among them. Components communicate by passing asynchronous messages through connectors. Connectors provide a mechanism for adding and modifying component bindings in order to support reconfiguration.

Distributed object-based approach: In CORBA [13] and COM+ [14], client IDL stubs and server IDL skeletons are generated at the compilation of IDL interface so that a client object can transparently invoke a method on a server object across the network. The method invocation will be handled by a group of objects, so that if in a distributed application one replica object fails or is being upgraded, another object is able to operate normally. This approach requires basic CORBA architecture, reliable group communication such as totally ordered protocol, and frequent checkpoint mechanism in order to maintain the state consistency in the object replicas during the running of CORBA applications.

As indicated above, if system reboot is not needed after upgrading, the primary standby method relies on redundant hardware and software. In centralized and real time system, it is a high cost to have hardware or software redundancy; because the communication and synchronization between a software component and it's redundant part is time consuming. Although hardware redundancy is well employed in spacecraft and other applications, America, England et al, to save cost, have developed spacecraft Computer System based on 80386 specially designed for small explorer satellites and built with minimal hardware redundancy; software system will reboot after OBSM [15]. Many commercial applications don't have hardware redundancy for the sake of reducing cost if safety is not so critical. We have developed an OBSM tool to make the OBSM more convenience and enhance the system availability while software upgrading for small satellites without system reboot, and it is also useful for commercial applications, especially for real time embedded system.

3. OBSM system overview

3.1 Frameworks

An OBSM workbench integrated into the operations environment has to include a compiler and software development environment (SDE), a software validation facility (SVF), a generator for patch telecommands, and a mission database of "memory images". The latter are files representing, the contents of the target computer memory (programs and data) at any time during the mission.

SVF for OBSM may be integrated with simulator already exists, many real time embedded system, such as VxWorks, QNS, have simulator with SDE. Reference [2, 16] presents a method to simulator the OBSM by software when design the spacecraft OBSM system. Patch maker receives source file modification information from SDE, creates the patch and then sends the patch to SVF, if errors occur during testing, error information is sent to SDE. Until SVF says ok, the new version software will be sent to the target machine, where patch handle task is waken up and the system is patched.

The framework of the OBSM system is as follows:

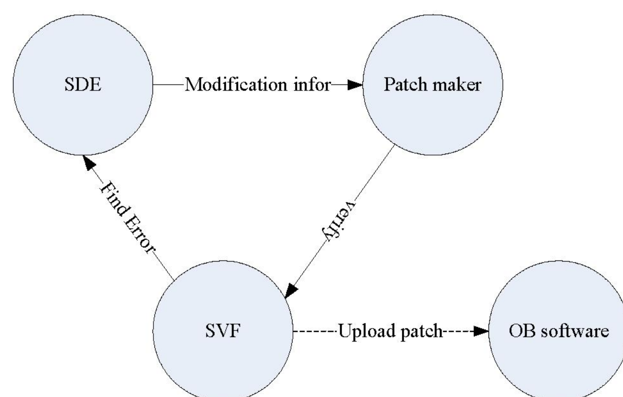

Figure 1: Framework

3.2 Classification of OBSM

OBSM can be classified into 3 catalogs, kernel modification, user application modification, and parameter modification.

Authorized licensed use limited to: University of Florida. Downloaded on February 12, 2010 at 18:30 from IEEE Xplore. Restrictions apply.

Kernel modification: the operating system may also have bugs since the complexity of the software. In this situation, if the kernel is preemptible, PC register of each task may pointer to an instruction that will be modified, so all the tasks must be stopped except the patch handle task; if the kernel is non-preemptible, any task that is not in the ready state needs be restarted.

User application modification: comparing with kernel modification, it is more common to modify user application. Different application is designed to different tasks. When an application needs to be upgraded, its according task is killed, and according code is updated, then the task is restarted. The traditional OBSM is to update the whole software in the Flash or other erasable storage and then reboot the system.

Parameter medication: it is also common to modify the parameter. Parameter is referred to global variable or static variable or a data area in memory. Local variable is in stack and can't be modified. In some situation, variables and data can be modified just rewrite the memory where the data locates; but in other situation, in the sake of keeping consistency of the data, for part of data may be in the registers, the tasks which use these data must be restarted.

The classification of OBSM can minimize maintenance-caused system unavailability.

4. System Implementation

4.1 Patching information fetching

To code modification, when the user has done the edit, SDE will tell OBMS to start to fetch the patch information. OBMS find the source files that were updated by the modification time of the files. Then compare the two and remove the functions that have not changed. The comparison uses the files that have been precompiled as its input, so the modification of the micro and the header file will be reflect in the functions. For global and static variable, if it has already in the file of the old version, then add an "extern" before the declarations. Then compile the file to the object file. The relocation will be illustrated in the section 4.3.

To parameter modification, users can modify the value of the global variables or static variables. Users input the variable name and OBSM system would find the address of the variable by looking up in the symbol table. If there are more than one variable called that name, then a warning occurs and the users are asked to specify the location of the variable. Then users input the value of the variables. Because there will be byte order (big-endian and little-endian) problem, we build a temporary source file for these variables and structures. After users having done all the inputs, in the temporary source file, we assign the values to the variables and members of structures, and output the values byte by byte. For structure, users input the structure type, and OBSM will find the definition of the structure type, and then show the members of the structure. Users may choose the member and input its value. To get the address of a structure member in target machine's memory, first we find the address of the structure in symbol table. Second compute the offset of the member in the structure, and then add the two to get the real address. But the offset of the member in the structure is affected by structure compact that is depending on the compilers. To make the compilers transparent to users, we add two macros in the temporary source file to compute the offset:

```
/* byte offset of member in structure*/
#define OFFSET (structure, member)          \
((int) & (((structure *) 0) -> member))
/* size of a member of a structure */
#define MEMBER_SIZE (structure, member)          \
(sizeof (((structure *) 0) -> member))
```

Using the temporary source file, we can easily avoid the byte order problem and compact problem which user must compute carefully according the architecture and compiler.

4.2 Task dependency analyzing

If a task calls a function (procedure), we say that the task is depending on the function. Software upgrading can be divided into functions upgrading and parameters upgrading. Functions upgrading may result in "Invalid OPCode" error et al if the task which depending on them does not be restarted. This can be described more clearly as follows:

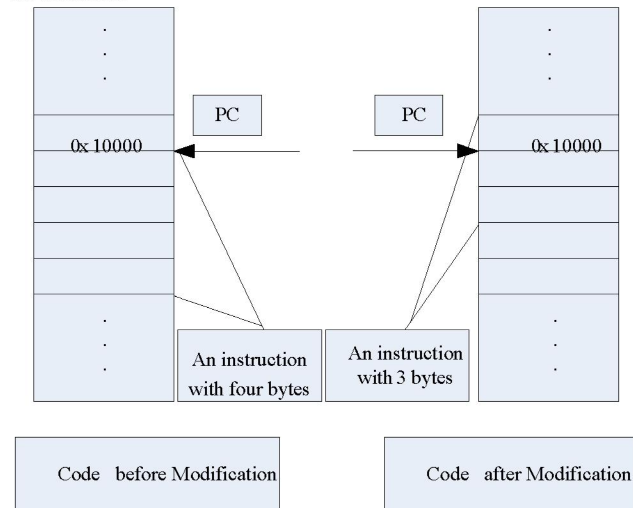

Figure 2• Invalid OP Code

Parameter modification needs task to be restarted also.

To resolve this problem, we create a caller graph of functions for each source file. In the following graph, a node denotes a function, and a directed edge (such as F1->F2) denotes that function F1 is called by F2. A dashed

Authorized licensed use limited to: University of Florida. Downloaded on February 12, 2010 at 18:30 from IEEE Xplore. Restrictions apply.

edge denotes that the caller function is in another file. We store the graphs in the database one relation per file.

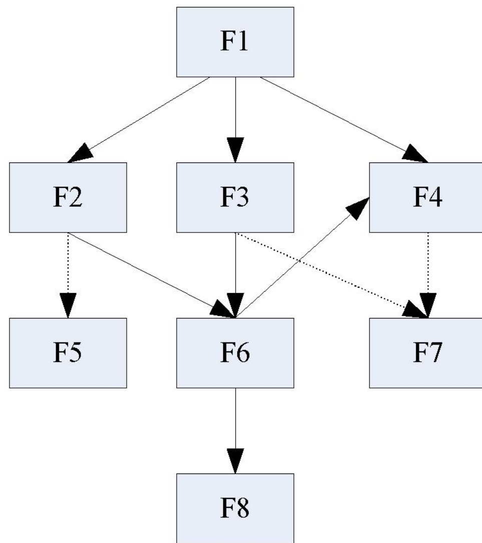

Figure 3: Caller Graph

After a source files have been modified, we find each function has been upgraded, and use the Breadth-first Traversal to traverse the graph. When arriving a node Fn, we add it to a set E, if the node is already in E, it indicates that the sub-graph started at Fn has already been traversed, so we go back to traverse the other part of the graph. Finally, we get the all functions, which are depending on the starting function, and find the tasks whose entry point is in these functions.

Function pointer makes the problem a little more complex. A function pointer may pointer to different function at different time in different conditions. Each function pointed by the pointer is added to graph; this would affect the efficiency little for function pointers are not always used.

Modification of variables and other data can also lead to task reboot, and its dependency can be computed in the same way.

4.3 relocation and re-relocation

In order to upgrade the software, target machine may need to keep a symbol table to do the relocation. But it is memory cost to maintain the symbol table; so we do the relocation in the host machine instead. If a new version function is larger than the old one, the new function will put into new memory address, so it needs to do re-relocation after the patch is uploaded to the target machine.

First, we assume the new version function will be put to address 0, and at X, it calls the function Y, after the patch is uploaded to the target machine, OS allocates memory for the new function at Z, so it calls Y at Z+X. This is described as follow figure.

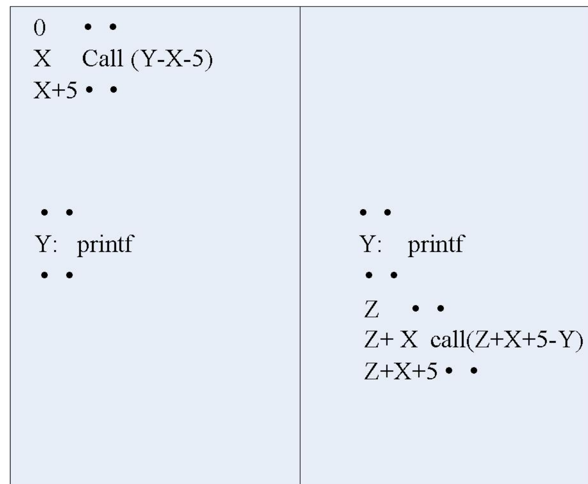

Figure 4: re-relocation

The caller instruction at X is call (Y-X-5) when doing the first relocation, and it should be call (Z+X+5 – Y) when the function is putted to Z. The value of target function Y can be found in the symbol table and value of X can be found in the object file. If a call instruction is call A after the first relocation and the new function is putted to address B, then the call instruction after re-relocation is changed to A-B.

Use the method described as above, users don't need to know the details of the relocation and object format, and can save memory for the symbol tables.

Relocation and re-relocation should be done according to different object format.

4.4 patching

When the target machine has received the patch command, it starts the OBSM task to receive a patch; a patch can be a code patch or a parameter patch. The format of the patch is described as follow,

{{affected task, entry point} {parameter address parameter size (in bytes) parameter value}}

The format of the code patch is described as follow:

{{affected task, entry point} {old code address, code size, code, new code address, code, and code size}}

After receiving the patch and doing the sum check, the maintenance task suspend the affected tasks, then copy the parameter value to the parameter address or copy new code to the address according its size. Finally restart the tasks.

To restart a task, the task priority, entry point, base of stack and arguments are always needed. Tasks priority and base of stack can be fetched in the TCB (task control block). In most of embedded system, the max number of arguments is determinable and the type is integer, else the

Authorized licensed use limited to: University of Florida. Downloaded on February 12, 2010 at 18:30 from IEEE Xplore. Restrictions apply.

patch must include the arguments number for a task; when initializing the task and its registers, we set the esp register as: esp register = (int) (pStackBase - (MAX_TASK_ARGS * sizeof (int)) or esp register = (int) (pStackBase – arguments length). Then we begin the task.

4.5 Resource Protection

When we kill a task, if the resources belonged to the task are not been properly released, there will be serious problems. For example, many applications in real time or embedded environment, don't support or use MMU, so if a task is killed, the memory its allocated but not freed, will lost; if a semaphore is not handle properly, other tasks who are waiting the semaphore, will wait it permanently.

In the OBSM system, we use two methods to protect the resource. One is the register mechanism, and the other is reboot-delayed mechanism.

The register method can be described as follows, when a task allocates resource, such as memory, system will register its task ID and the resource. Once the task needs to be restarted, the resource belonged to it will be released. Many real time systems, even like VxWorks, tasks won't reclaim its resource if they are killed.

We add a property called safe-count to task control block to protect the resources which cannot be reclaimed when the tasks need to be restarted. When a task allocates a resource, increase the safe-count, and vice versa. If a task need to be upgraded (but not crashed), then OBSM system will check the safe-count to see whether the task can be killed safely, if not, the task will be allowed to run for more time, and its tcb is add to a queue. OBSM system will check the queue to see whether there are tasks can be restarted. If a task overruns the limited time and has not been restarted, it will be forced to restart, and tasks that share resources with this task will also be restarted, else the task is restarted without any impact on other tasks. This is called reboot-delayed mechanism.

5 System Prototype

We have conducted some experiments to determine the overhead incurred due to OBSM. Our experiments based on Pentium-s, 100 MHz, 3 source files modification (20 functions), and the overhead during OBSM is about 15 microseconds. If the conventional method is used, the process will take 2 seconds and 110 microseconds. The big overhead of the conventional method is because the reboot of the system. The overhead of our method can be divided into two parts, one is the overhead for suspending and restarting the tasks, this normally takes about 5 seconds. The other is to copy the new version function and data to its proper address and do the re-relocation, and this may take 10 seconds and may increase with the code size.

An operation interface is as below, the upper part of the interface is the changed source files, and the lower part is the depending tasks that need to be restarted in the target machine.

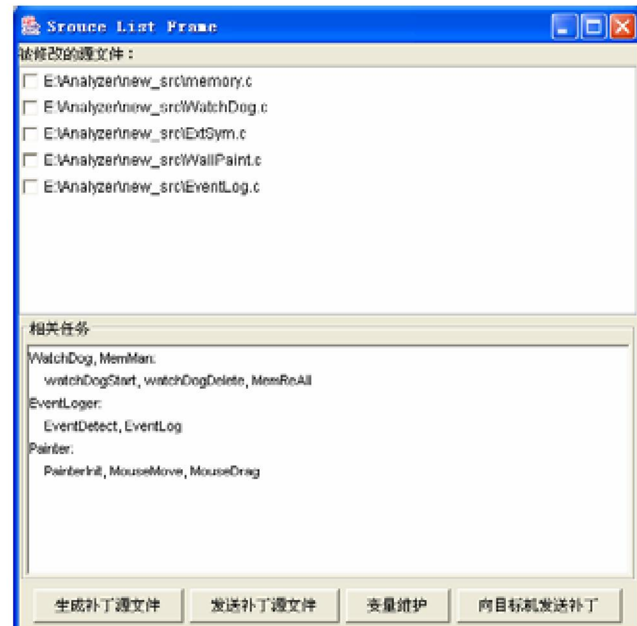

Figure 5: example

6. Summary

There are two traditional OBSM methods, one needs reboot after OBSM, and the other relies on redundant hardware and software. But in centralized and real time embedded system, it is a high cost to use hardware and software redundancy. So we develop a method to update the software by writing the patch to memory directly.

The classification of OBSM can reduce the overhead during the OBSM. The task dependency analysis and resource protection ensure the correctness of upgrading. And Automation of patch information fetching, relocation and re-relocation, and patching help the user upgrade the software conveniently and correctly.

This OBSM method has a little overhead comparing to other OBSM methods, even to the live software upgrading. It will be useful for many other applications that are subject to frequent software upgrading and require high availability, such as Internet services, transportation systems, airline reservation systems, telephone systems and medical systems, especially for real time embedded system.

Acknowledgement

This work is supported by the National Defense Advanced Research Program of China.

Authorized licensed use limited to: University of Florida. Downloaded on February 12, 2010 at 18:30 from IEEE Xplore. Restrictions apply.

Reference

[1] Faren Qi, Renzhang Zhu, Yili Li, "Manned Spacecraft Technology", National Defense Industrial Press, Beijing, China, 1999, pp. 401-402.
[2] Bartolome Real Planells, "XMM Instrument On-board SW. Maintenance: An Approach Via Processor Simulation", SpaceOps 98, Tokyo, Japan, 1998.
[3] Denis M, "The cluster On-Board Software Maintenance Concept". ESA bulletin, 1997, vol.91, pp. 18~24.
[4] Junshe An, Yanqiu Liu, Huixian Sun, "Implementation of On-board Software Maintenance", Computer Engineering, Shanghai, China, Feb.2003, vol.29, No.2, 238~239.
[5] C. Steiger, R. Furnell, J. Morales, "OBSM Operations Automation Through The Use of On-board Control Procedures", SpaceOps 2004, Montreal, Canada, 2004.
[6] A. T. Tai and L. Alkalai, "On-board Maintenance for Long-life Systems", in Proceedings of the IEEE Workshop on Application-Specific Software Engineering and Technology (ASSET'98), Apr. 1998, pp. 69–74.
[7] A. T. Tai, Kam S.Tso, "Leon Alkalai, Savio N. Chau, William H. Sanders, Low-Cost Error Containment and Recovery for Onboard Guarded Software Upgrading and Beyond". IEEE Transactions on computer, 2002, vol.51 No.2
[8] Ann T. Tai, Kam S.Tso, Leon Alkalai, Savio N. Chau, William H. Sanders, "On the Effectiveness of a Message-Driven Confidence-Driven Protocol for Guarded Software Upgrading". Perormance Evaluation, 2001, Vol.44, pp.211-236.
[9] A.T.Tai, K.S. Tso, L.Alkalai, S.N.Chau, and W.H.Sanders, "On Low-Cost Error Containment and Recovery Methods for Guarded Softeare Upgradting, Proc". 20th Int'l Conf. Distributed Computing Systems(ICDCs 2000), 2000, pp.548-555.
[10] Jeff Magee and Jeff Kramer, "Dynamic Structure in Software Architectures", Fourth SIGSOFT Symposium on the Foundations of Software Engineering (FSE), San Francisco, October 1996, pp. 3-14.
[11] Peyman Oreizy and Richard N. Taylor, "On the Role of Software Architectures in Runtime System Reconfiguration", Proceedings of the International Conference on Configurable Distributed Systems (ICCDS 4), Annapolis, Maryland, May 1998.
[12] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor, "Architecture-Based Runtime Software Evolution", IEEE/ACM International Conference on Software Engineering (ICSE '98), Kyoto, Japan, April 19-25, 1998, pp. 177-186.
[13] Object Management Group, "The common Object Request Broker: Architecture and specification, 2.2 edition", OMG Technical Committee Document formal/98-07-01, Feb 1998.
[14] Microsoft Corporation, Various COM documents, MSDN library, 1998.
[15] Lu Dongxin, Teng Lijuan, Hong Bingrong, Gao Feng, "Design of a Five Level Protection Sytem Based on Watchdog For Sepacraft Computer", Journal of HarBin Institute of technology, Feb.2001, Vol.33, No.1, pp.13-19.
[16] M.M.lrvine A.Dartnell, "The Use Of Emulator-Based Simulators For On-Board Software Maintenance", European Space Agency, ESA SP, 2001, n 509, p 150-155

Authorized licensed use limited to: University of Florida. Downloaded on February 12, 2010 at 18:30 from IEEE Xplore. Restrictions apply.