Software Structures for Updating Mobile Phone Software

Satoshi Mii Ryozo Kiyohara
Mitsubishi Electric Corp. Information Technology R&D Center

The rapid growth of mobile phone software sometimes leads to the necessity of the software revision even after the shipment. To lighten the effect of this problem, OTA (over-the-air) update services have been provided recently. It is essential to shorten the time required for update services, and update time largely depends on the characteristics of mobile phones such as low data transfer speed and usage of NOR Flash ROM.

In this paper, we introduce a software structure that is effective for the reduction of the update time. This software structure is based on the following idea: divide components of software into modules and set each module independent from other modules. We also show effects of this software structure and introduce a way to decide the appropriate number of modules.

1 Introduction

Recently, as the amount of mobile phone software has been increased, sometimes bugs are reported after the phone has been deployed, and manufacturers or carriers are urged to provide update services. To improve the service to mobile phone users and lower costs, update services which transfer update data over-the-air (OTA) are emerging [9]. To provide an OTA update service, it is essential to shorten update time so as to satisfy users.

Update time mainly consists of download time and install time. It is obvious that a "differential update" technique can make both download and install time short. Here, differential update means that we use differences between two versions of software, or executables, as update data. We also call data representing differences "delta data".

Download time depends on the size of the transferred data and data transfer speed. Since data transfer speed in wireless communication networks is not fast enough to transfer large amounts of data, it is important to make the size of the transferred data, or delta data, as small as possible. There are many studies on algorithms that make the size of the delta data small [1][8]. But there is no guarantee to make the delta data always small if two versions of executables differ considerably from each other. So, we must prepare executables where any kind of update includes a small change only.

Install time depends on how many sectors to be erased rather than how many bytes are changed, since most mobile phones use NOR Flash ROM to store executables. Accordingly, we must prepare executables where any kind of update induces changes in limited areas only.

Thus, a key to shorten update time of an OTA update service is to prepare executables where updates cause small changes in limited areas. To construct such executables, we introduce the idea of "module structure" and show how to build an executable according to a module structure. We also show effects of adopting a module structure and introduce a method to determine the appropriate parameter that characterizes a module structure.

2 Related Works

In most differential update techniques, we represent delta data as a combination of "copy" commands and "add" commands [7]. In this approach, a key to reduce the delta data size is in both finding as much copy commands as possible and encoding commands efficiently. For example, *zdelta* [8] utilizes the *zlib* compression library [4] to fulfill the above requirements, and shows good performance on the delta data size. Other than that, there are additional techniques specializing in software updates. These techniques take into account platform-dependent information, such as symbol references [2] or register assignments [6].

As described above, there are many studies on size reduction of the delta data. Thus, we can shorten download time using these techniques. But there are few studies that concentrate on install time or efficiency. For example, a data structure is introduced to handle insert and delete operations efficiently [3], but it is not suitable for storing executables in NOR Flash ROM.

In this paper, we introduce a module structure to reduce install time. We note that the module structure is also meaningful to reduce the delta data size, or download time.

3 Overview of Module Structure

3.1 Premise

As a program loading mechanism, we assume a mechanism as follows:

(1) All symbol references are statically resolved,
(2) Each program in the executable form is executed directly on NOR Flash ROM.

3.2 Outline of Module Structure

We explain an outline of the module structure in this section. We also show an image of the module structure in Figure 1.

3.2.1 Fixation of Program Allocation

As programs are written into NOR Flash ROM, we must rewrite the whole program even if the content does not change and only the allocation changes. If we

allocate all programs adjacent to each other, any change in the size of one program causes a rewriting of all programs allocated behind it. So, it is preferable to allocate each program on a fixed address to reduce unnecessary re-allocation. But if the number of programs is large, it becomes difficult to manage the allocation of all the programs.

So we propose to manage the allocation in a unit called "module". Each module consists of one or more programs, and we fix the order of programs in each module. We also fix the start address of each module and allocate programs in their order. To fix the allocation of all modules, we set a gap zone between each pair of adjacent modules.

By managing allocation of programs in the unit of the module, we can reduce negative effects on install time caused by changes in program size. But there is another problem in shortening the install time. That is the symbol references across different modules.

Assume that one program is corrected and the address of a function in that program changes. Then, most of the references that refer to the function also change. This means that one correction in a program may cause changes in not only the module to which the program belongs but also other modules. In this case, we have to rewrite many sectors and so we need a long install time. We also need a long download time because changes are spread throughout the whole executable and the size of the delta data becomes large.

To avoid the effects of symbol references, we introduce another feature of the module structure below.

3.2.2 Function References

To resolve function references, we set a "vector table" in each module. A vector table is a set of vectors, consisting of branch instructions to global functions in the module. To resolve a function reference across two modules, we let the reference refer to the vector that indicates the target function rather than the target function itself (Figure 2). By forcing the address of each vector to be fixed, address changes of global functions in one module do not affect references in other modules.

We do not apply vector jumps to function references resolved inside the same module. This rule is based on two reasons as follows:

(1) Vector jumps cause overhead in execution time,
(2) Programs in the same module tend to be allocated in the same sector, so install time does not increase so much without vector jumps.

3.2.3 Data References

As for data references, we cannot apply any ideas like a vector table because of their characteristics. But from pre-analysis, the size of the data rarely changes due to updates. So it is enough to pack all the data in each module together and allocate them in a fixed area separated from the code in the module (Figure 1).

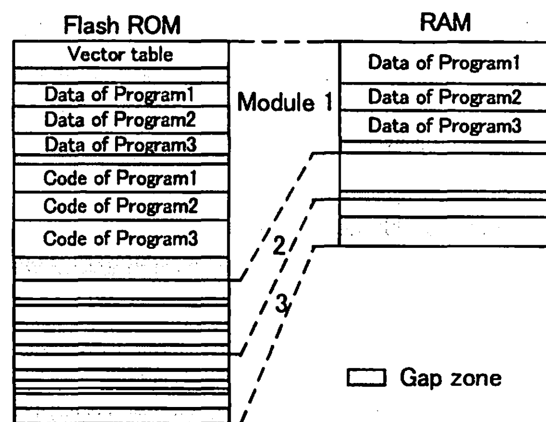

Figure 1: Allocation of programs

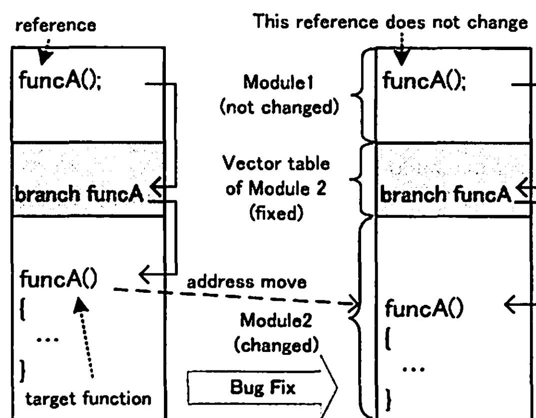

Figure 2: Vector table

If we have to treat data references strictly, changing all global data references across two modules via functions is one solution. But it is not practical because we must correct many source files. It also creates overhead in execution time.

3.3 Executable Construction

A procedure to construct an executable according to the module structure is divided into two phases, initial phase and update phase. We have proven this procedure using GNU compiler collection and binary utilities.

3.3.1 Initial Phase

The initial phase constructs the initial executables. The initial phase has five steps and each step is carried out for each module simultaneously.

(1) Compile

Compile each program without resolving symbol references.

(2) Temporary link

Link compiled programs with a link script without resolving symbol references and generate a temporary module. The link script describes the allocation, and we confirm addresses of global

functions and data in this process. We also set gap zones in this step.

(3) Vector table generation
Extract global functions from the temporary module and build a vector table. Each vector table is written in assembly language and then assembled.

(4) Symbol file generation
Extract global data from the temporary module and write their symbol names and addresses into a symbol file, which is written in a format of link script. Also write the symbol name of each global function and address of the corresponding vector.

(5) Final Link
Link the temporary module with the vector table and symbol files of other modules. In this step, we resolve all symbol references and generate a final executable for each module.

3.3.2 Update Phase

Update phase is carried out whenever updates are required. Basically, this phase is applied to corrected modules only.

(1) Recompile
Compile corrected programs without resolving symbol references.

(2) Temporary link

(3) Vector table generation
Make a vector table inheriting the address of each vector from the vector table of the previous version.

(4) Symbol file generation
If the address of a global data is changed, the update phase has to be carried out for modules that are not corrected.

(5) Final link

4 Estimation of Update Time

The main effect of the module structure is that we can shorten the update time. Here, we roughly estimate the update time and show the benefits of the module structure.

4.1 Definitions and Assumptions

We define system specific figures in Table 1. The column marked "Sample" is explained in Chapter 5. Then we make assumptions listed below for simplicity:

- All programs are in the same size,
- Programs are assigned to modules equally,
- Symbols and references are uniformly distributed in each program,
- One update causes a correction in one program,
- Corrections occur in each program at random.

It is necessary to know how many modules are corrected to estimate the update time. Let x denote the number of modules and y denote the frequency of updates. Then, the average number of corrected modules under x and y is

$$u_{xy} = \frac{x^y - (x-1)^y}{x^{y-1}}. \quad (1)$$

4.2 Install Time

Corrections in one module do not always require erasing and rewriting to all sections in the module, and the average number of sections to be erased and rewritten depends on how many times corrections occur in the module. But for simplicity, we assume that we need to erase and rewrite all sections in one module if at least one correction occurs in the module.

Here, the average number of sectors in each module is S/x. So, the average install time is

$$T_i(x,y) = \frac{S}{x} \cdot E \cdot u_{xy}. \quad (2)$$

4.3 Download Time

Download time depends on the size of the delta data and average data transfer speed. Since data transfer speed is specific to a service, we discuss the size of the delta data only.

4.3.1 Causes of Differences

The size of the delta data mostly depends on how many bytes and how many parts are changed, and changes are classified into two types. One is caused by a correction itself, like adding one conditional branch. We call this kind of change "direct changes".

Another is caused by symbol references. As we have explained in Section 3.2, all references in one module are not changed when the other modules are corrected. But when corrections happen in their module itself, some of the references are changed. We call this kind of change "indirect changes". In indirect changes, there are three types of changes as follows:

(1) Absolute references to moved area from the whole module,
(2) Relative references between moved area and fixed area,
(3) Relative references to another module from moved area.

Figure 3 shows an image of these changes. Note that the third type includes vector jumps.

Now, We estimate download time and the size of the delta data under the number of modules x and the frequency of updates y.

4.3.2 Direct Changes

The size of delta data caused by direct changes depends on how corrections are made. The precise estimation of the size depends on the actual cases. Therefore, we define the average delta data size per correction A as a system specific figure. Then, the size of the delta data caused by direct changes $D_d(x,y)$ is given as

$$D_d(x,y) = A \cdot y. \quad (3)$$

Table 1: System specific figures

Symbol	Descriptions	Sample
S	The number of sectors in NOR Flash ROM	224
E	Time to erase and rewrite one sector (in sec)	2.0
Z	Average data transfer speed (in byte/sec)	12800
P	The number of programs in targe software	3417
A	The average size of delta data per one direct change (in byte)	10240
G	The number of global functions	23833
Rir	The number of relative references inside each program	13940
Ria	The number of absolute references inside each program	350884
Rar	The number of relative references across different programs	214980
Raa	The number of absolute references across different programs	70464

4.3.3 Indirect Changes

In the case of indirect changes, the delta data size mostly depends on how many references are changed. Here we use add commands and copy commands introduced in [7] for the estimation. We assume that the size of each add command is *5* bytes and the size of each copy command is *4* bytes. Then, for the number of changed references *r*, the delta data size is nearly *(5+4)r* bytes. Although there are cases that some of the copy commands are not necessary and some of the add commands can be unified, we ignore those cases for simplicity.

Now we estimate the number of changed references to calculate the delta data size caused by indirect changes. There are three types of indirect changes, and the number of changes for each type largely depends on where and how many times corrections occur in each module. Although it is not impossible to calculate the average for each type, we only use the maximum because calculating the average is rather complicated.

For the first type of changes, the maximum number of changed references in one module r_{x1} is the number of absolute references where both referring and referred sides are in the module. Since we assume uniform distributions of references, the equation below holds.

$$r_{x1} = \left(R_{ia} + \frac{P-x}{Px} \cdot R_{aa} \right) \Big/ x. \quad (4)$$

Here *(P-x)/Px* means the probability that another program comes to be in the same module from a viewpoint of each program.

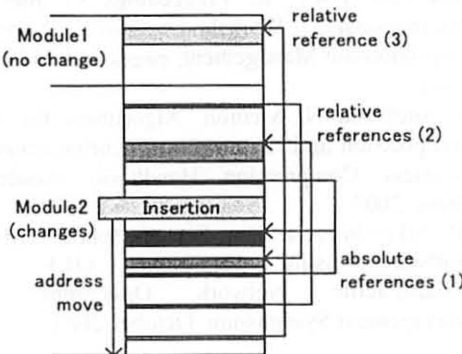

Figure 3: Image of indirect changes

For the second one, the maximum number r_{x2} is the number of relative references where both sides are in the module. Taking the number of vectors *g/x* into account,

$$r_{x2} = \left(R_{ir} + \frac{P-x}{Px} \cdot R_{ar} + g \right) \Big/ x. \quad (5)$$

For the third one, the maximum number r_{x3} is the number of relative references where referred sides are in other modules, and so

$$r_{x3} = \left(1 - \frac{P-x}{P \cdot x} \right) \cdot R_{ar} \Big/ x. \quad (6)$$

Considering the average number of corrected modules u_{xy}, the size of the delta data caused by indirect changes is

$$D_i(x,y) = 9 \cdot u_{xy} \cdot (r_{x1} + r_{x2} + r_{x3}). \quad (7)$$

In this way, the size of the delta data is

$$D(x,y) = D_d(x,y) + D_i(x,y), \quad (8)$$

and the download time is

$$T_d(x,y) = \frac{D(x,y)}{Z}, \quad (9)$$

where *Z* is the average data transfer speed.

4.4 Total Update Time

From Section 4.2 and 4.3, we can obtain the total update time *T(x,y)* as described below:

$$T(x,y) = T_i(x,y) + T_d(x,y). \quad (10)$$

It is obvious that a larger number of modules *x* results in a shorter update time *T(x,y)* for every frequency of updates *y*. See Chapter 6 and Figure 4 where update times are shown for a sample software.

5 Appropriate Number of Modules

We have shown the advantage of the module structure in Chapter 4. But there also exists disadvantages of the module structure. The main disadvantages are overhead in execution time and memory usage for gap zones. It is clear that a larger number of modules makes the disadvantages worse.

Thus, there is a tradeoff related with the number of modules, and so it is important to decide the appropriate number of modules. In this chapter, we introduce a method to determine the appropriate number of modules.

5.1 Approach

In order to decide the appropriate number of modules, we adopt an approach as follows.

(1) Formulate a correlation among the number of modules, the frequency of updates and update time.
(2) Set permissible amounts on the frequency of updates and update time.
(3) Calculate the minimal number of modules that satisfies the permissible amounts set in (2).
(4) Adopt the number calculated in (3) as the appropriate number of modules.

5.2 Method

Since we have already shown the correlation in Chapter 4, we can describe a method to determine the appropriate number of modules X according to the permissible frequency of updates Y and the permissible update time T as follows:

(1) Set $x:=1$,
(2) Calculate $T(x,Y)$,
(3) If $T(x,Y) > T$, then set $X:=x-1$ and end,
(4) Set $x:=x+1$ and go back to (2).

6 Example and Evaluation

We show an example of update time estimation using data from a certain mobile phone. We show system specific figures in the "Sample" column of Table 1. Among those figures, S, E, P, R_{ir}, R_{ia}, R_{ar}, R_{aa} and G are obtained from the mobile phone. We determine Z assuming a 3G service, and A from results of pre-analysis.

Figure 4 shows the estimation of the total update time. In the figure, we plot for four Y values, 1, 10, 30 and 50. We can see that a large number of modules result in a short update time. But for small y, a large number of modules is not necessarily required. For example, there is little difference between $x=60$ and $x=80$ for $y=1$.

From Figure 4, we can determine the appropriate number of modules according to the method introduced in Section 5.2. For example, the appropriate number of modules is about 50 under the condition $Y=30$ and $T=400$.

7 Conclusion and Future Work

In this paper, we introduce a module structure aiming at the reduction of update time for OTA update services. We also introduce a method to build executables according to the module structure, and show the effects of the module structure and a method to determine the appropriate number of modules from a standpoint of update times.

For future works, we have listed topics below:

- Evaluation of the module structure using real data. This includes not only update time, but also execution time and memory usage.
- Detailed estimation of update time.

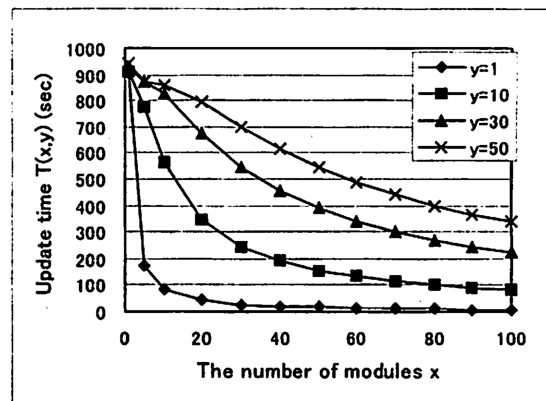

Figure 4: Update time

- Consideration of program assignments.
- Estimation of a correlation between the number modules and execution time.

References

[1] M. Ajtai, et al. Compactly encoding unstructured input with differential compression. Journal of the ACM, 49(3):318-367, May 2002.

[2] B. S. Baker, U. Manber and R. Muth. Compressing Differences of Executable Code. In Proceedings of ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS '99). May 1999.

[3] P. Eaton, E. Ong and J. Kubiatowicz. Improving Bandwidth Efficiency of Peer-to-Peer Storage. In Proceedings of the Fourth IEEE International Conference on Peer-to-Peer Computing (IEEE P2P '04). August 2004.

[4] J. Gailly and M. Adler. zlib compression library. Available at http://www.zlib.net/.

[5] J. J. Hunt and W. F. Tichy. Delta Algorithms: An Empirical Analysis. ACM Transactions on Software Engineering and Methodology, 7(2):192-214. April 1998.

[6] Y. Okada and K. Terazono. A New Delta Compression Algorithm Suitable for Program Update in Embedded Systems. In Proceedings of the Data Compression Conference (DCC '04). March 2003.

[7] C. Reichenberger. Delta Storage for Arbitrary Non-Text Files. In Proceedings of the 3rd International Workshop on Software Configuration Management, pages 144-152. June 1991

[8] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In Lossless Compression Handbook. Academic Press. 2002.

[9] M. Takeichi, et al. Bug Fix of Mobile Terminal Software using Download OTA. The Asian-Pacific Network Operations and Management Symposium. October 2003.