FOTA에서 Scatter Loading의 최적화 방법 연구

이희영°, 조준동
성균관대학교 전자전기컴퓨터공학과
heeya0528@hotmail.com , jdcho@skku.ac.kr

A Study on FOTA Upgrade Efficiency by Manipulating a Scatter Loading

Heeyoung Lee , Jundong Cho
VADA Lab. School of Computer Engineering, Sungkyunkwan University

요 약

FOTA는 무선기능이 장착된 Mobile Device에 새로운 Software에 대한 알림기능이 도착하면, Software가 탑재된 서버에 접속하여 Software를 Download 받고, 그 Download한 Software를 Upgrade 하는 기능을 말한다. FOTA 기능을 장착하기 위해서 Mobile Device는 Delta Package의 사이즈를 최소화 하기위한 특별한 Binary 구조를 가지는데, 두 Binary 의 차이를 압축한 것을 Delta Package라고 부르며, Binary 사이에 Upgrade를 위한 여분의 Gap을 두어, 향후 수정된 내용이 있을 때, 수정사항을 공간 내에 포함할 수 있도록 한다. 바이너리를 구성하는 Object들이 Image내에 어떤 위치에 포함될 것인지를 결정하는 Scatter Loader에 따라, Binary의 구조 및 확장성, Delta의 크기를 결정하게 되는데, 이것은 Object의 Type이나 Scatter Loading File내에서 명시한 순서와는 관계가 없고, Execution Region의 분할 개수를 늘릴수록, 각 Object의 Dependency 별로 묶을수록 Delta Size가 작아지는 것을 알게 되었다. 이 논문에서는 위에서 제시한 조건이 Delta Size에 미치는 원인에 대해 분석하고, Scatter Loading File을 최적화시킬 수 있는 방안에 대해서 연구한다.

1. 서 론

FOTA는 무선기능이 장착된 Mobile Device에 새로운 Software에 대한 알림기능이 도착하면, Software가 탑재된 서버에 접속하여 Software를 Download 받고, 그 Download한 Software를 Upgrade 하는 기능을 말한다. 이 때 모든 Software를 통째로 Upgrade 하는 것이 아니라, 두 Software의 수정된 부분만을 추출하여 파일을 생성하는데, Upgrade 전후의 Binary의 차이를 압축한 것을 Delta Package라고 부른다. FOTA에서는 이 Delta Package를 무선 상으로 Download 받아 Upgrade 하는 것이다.

일반적으로 FOTA 기능을 장착하기 위해서 Mobile Device는 Delta Package의 사이즈를 최소화 하기위한 특별한 Binary 구조를 가지는데, Binary 사이에 Upgrade를 위한 여분의 Gap을 두어, 향후 수정된 내용이 있을 때, 수정사항을 공간 내에 포함할 수 있도록 한다. 이 때, 바이너리를 구성하는 Object들이 Image내에 어떤 위치에 포함될 것인지를 결정하는 Scatter Loader에 따라, Binary의 구조 및 확장성, Delta의 크기를 결정하게 된다. 이 논문은 위에서 언급한 FOTA의 특징을 알아보고, Firmware의 Scatter Loading File을 조정하여 Binary를 최대로 확장하기 위해서 Delta Package의 크기를 줄이는 최적의 Scatter Loading File의 조건을 파악하여, FOTA의 Firmware Upgrade의 횟수의 한계를 극복할 수 있는 방법을 연구한다.

2. 관련 연구

2.1 FOTA Solution의 정의 및 특징

FOTA란 Firmware Over The Air의 약자로, 무선기능이 장착된 Mobile Device에 새로운 Software에 대한 알림기능이 도착하면, Software가 탑재된 서버에 접속하여 Software를 Download 받고, Download한 새로운 Version의 Software로 Upgrade 하는 기능을 말한다. FOTA는 Mobile 기능이 제한받지 않는 환경이라면 시간적, 공간적인 제약을 받지 않는다는 장점을 가지고 있다. 특히 Firmware의 치명적인 오류나 결함이 발생했을 때, Firmware를 Upgrade하기 위해서 Web-Site에 접속하여 Firmware를 검색하고, Download 받고, Cable을 연결하여 Upgrade 하는 일련의 복잡한 과정을 거치지 않기 때문에 편리할 뿐만 아니라, 사용자가 서비스센터를 찾아가는 시간적, 경제적 비용을 절약 할 수 있다. 또 단말기 사업자 측면에서는 Customer Service를 위한 별도의 서비스 센터를 운영하지 않아도 되므로 부대비용을 절감할 수 있는 장점이 있다.

2.2 OMA-DM (Open Mobile Alliance Device Management)

Open Mobile Alliance는 국제 모바일 규격 표준화 단체로 솔루션 간 호환성을 확보하기 위해 FOTA와 관련

된 산업 표준을 제정하고 있다. OMA-DM Client는 OMA에서 정의하는 표준 Command를 해석하고 실행하고 Management 서버로 Action을 취할 수 있는 Client를 일컫는다. FOTA를 지원하기 위해, 각 Mobile Device는 Management Objects를 Tree의 형태로 관리한다. 이것은 Device 내에 Binary File의 형태로 저장되면서 Device를 관리하기 위한 Node들의 정보를 포함하고 있다. Mobile Device는 Server와 정보를 교환하고, 유지하며, 갱신한다. [1,2]

2.3 Firmware Download & Upgrade

User는 FOTA Upgrade에 필요성을 인식한다. 이것은 Notification Server로부터 Scheduling 된 FOTA Notification을 받거나, User가 직접 Server 에 접속하여 Upgrade할 Firmware를 확인할 수도 있다. Mobile Device는 DM Session을 열어 Server와 관리정보를 주고받으면서, Firmware를 Upgrade 여부를 판단한다. 이때 단말기의 제조사, 모델정보, Firmware의 버전이 서버에 등록된 Package의 정보와 일치하는지 확인하고 일치할 경우에만 Download를 Session을 Trigger한다. Mobile Device는 Download Server로부터 Download Descriptor(DD)를 전달 받는데, 이것은 Download 받을 Package에 대한 정보를 포함하고 있다. Mobile Device는 DD내에 정의된 Information을 분석하여 Firmware 버전이 단말의 Management Tree내에 저장된 버전보다 최신 버전일 경우에만 Delta Package를 받는다. Delta Package가 Download되면, Mobile Device는 이것을 ROM 영역에 Copy하고, 설치한다. Mobile Device는 Server에 Download 및 설치가 성공되었는지에 대한 결과를 Report한다. [3]

2.4 Scatter Loading & Arm Linker

Image의 Memory Map을 구성하기 위해서, 사용자는 어떤 Output Section을 Grouping하여 Region을 구성할 것인지 정보를 수집하고, Memory내에서 각 Region의 Information이 Image의 Memory에 어느 Address에 위치해야할지 결정해야 한다. 이것은 사용자가 Scatter Loading Description File을 작성함으로서 Linker에 명시할 수 있다. Arm Linker는 Program의 Object와 Library를 실행 가능한 Image로 만들어 주며, Code 와 Data를 Memory 상에 어느 곳에 위치시킬지를 결정한다. 또한 Link 된 파일간의 Debug 정보와 Reference 정보를 생성한다. Arm Linker는 앞서 설명한 Scatter Loading File에 명시된 Input Section 내에 정의된 RO, RW, ZI 정보를 바탕으로 Output Section, Region, Image라는 큰 Block을 만들어내는 역할을 한다. [4]

FOTA에서 Scatter Loading File은 Firmware의 Memory Map상에서 각 Object들의 위치를 명시하고, 관련된 Object들을 하나의 Region으로 묶어 Block을 형성하고, Block간의 위치를 고정시키며, Upgrade를 위한 여분의 공간인 GAP을 만들어내는 역할을 한다.

2.5 FOTA Scatter Loading

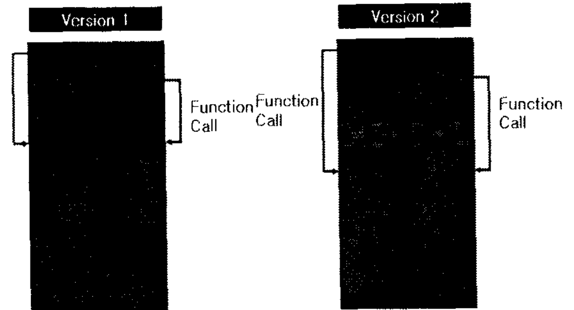

그림 1. FOTA를 적용하기 이전 Memory Map

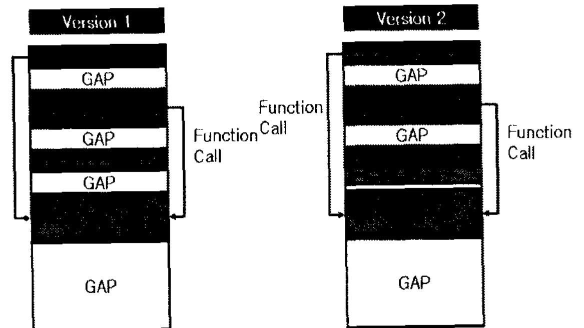

그림 2. FOTA를 적용하기 위한 Memory Map

그림 1, 그림 2와 같이 2개의 Software Version 이 있다고 가정하자. Version1은 A, B, C, D 4개의 Function을 포함하고 있고, A와 B는 D를 호출한다. 이때, Version1의 C Function에서 오류가 있음을 인지하고, 수정하여 C'을 만들었을 때, Memory Map 상에서는 A, B Function이 호출하는 D Function 의 Address가 고정되지 않았으므로 D Function의 Address는 C의 수정사항만큼 뒤로 밀려나게 된다. 이것은 D의 Address 뿐만 아니라, D를 호출하는 A, B의 Address에도 영향을 미치므로, Binary 전체적으로 수정사항이 발생하게 된다. FOTA에서는 그림 2와 같은 Memory Map의 구조를 권장한다. 각 Object의 Address를 고정하고 각 Object 사이에 수정사항에 대비한 Gap을 두어, 수정사항이 발생 했을 경우, Gap 내에 포함시킬 수 있도록 한다. 이렇게 하면 전체 바이너리에서 최대한 C'만큼 변경이 되기 때문에, 두

바이너리의 차이가 줄어들게 된다. [5]

3. 본 론

3.1 Input Section Type 별 영역 분리

각 Object 파일들은 Relocatable Object File의 형식을 따르는데, 이것은 Binary Code와 Data를 가지고 실행 가능한(Executable) Object 파일을 만들기 위해 Compile-Time에 다른 Object File과 결합할 수 있는 형태를 말한다. Compiler는 이 Object File 들을 합쳐 Memory 에 Load 될 수 있는 실행될 수 있도록 한다. 각 Region 내의 Relocatable Object 파일들은 Linker에 의해서 Input Section Type에 따라 하나로 합쳐진다. 즉, Object들의 Code Section 끼리 묶어 Executable Object내의 하나의 Code Section을 만든다. 또, Data, ZI, RW의 속성을 가지는 Section Type별로 묶어, 새로운 Executable Object를 구성한다. 그림 3을 보면 Relocatable Object들이 여러 개의 Region 별로 나누어져 있을 때, 각 Region 내의 Object들이 Executable Object를 구성하기 위해, Section 속성 별로 재배치되는 것을 알 수 있다. [6]

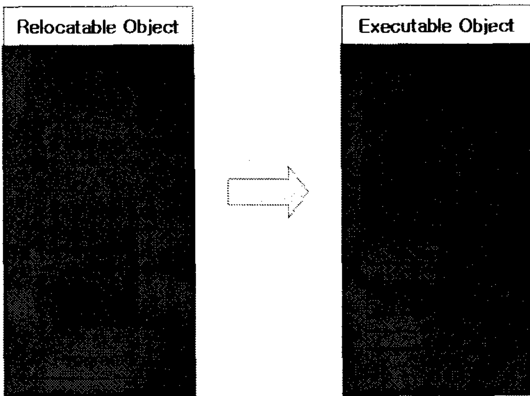

그림 3. Relocatable Object의 Executable Object 변환 과정

```
EXEC_CONST__TEST1 0xA0840000 FIXED
{
    application1.o (+RO-CODE) ;
    application1.o (+CONST) ;
}
EXEC_CONST__NEXT 0xA0860000 FIXED
{
...
}
```

그림 4. RO 영역의 CODE, CONST 합침 SCL

```
EXEC_CONST__TEST1 0xA0840000 FIXED
{
    application1.o (+RO-CODE) ;
}
EXEC_CONST__TEST2 0xA0850000 FIXED
{
    application1.o (+CONST)
}
EXEC_CONST__NEXT 0xA0860000 FIXED
{
...
}
```

그림 5. RO 영역의 CODE, CONST 분리 SCL

그림 4와 같이 Application 1의 Code와 Const type을 하나의 Region에 위치시킨 것과, 그림 5와 같이 Code와 Const Type을 서로 다른 Region에 분리시킨 Scatter Loading File을 작성하고, 동일한 수정사항을 발생시켰을 때(Application1에 Code와 Const를 추가함), Object의 Type을 분리한 것과 병합한 것의 Delta 생성 시에 극명한 차이가 없음을 알 수 있었다. 그림 5를 보면 Application1의 .text와 .rodata에서 수정사항이 발생했을 때 수정영역을 빗금으로 나타내면, 수정이 발생한 Address 이후의 text와 rodata의 address가 변경 된다. 병합되었을 때는 .text와 .rodata 영역이 연달아 있기 때문에, a 만큼의 주소가 변경된다. 그러나 .text와 .rodata가 두 개의 영역으로 분리되어 있을 때, .rodata의 시작 주소가 0xA0850000 으로 고정되어 있기 때문에, .rodata 수정사항 이전의 주소는 변경되지 않게 된다.

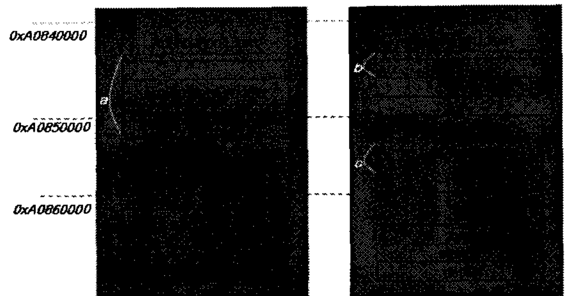

그림 5. 병합, 분리 SCL 작성 시 Executable Object에서 수정 영역 비교

$a \div b + c$

3.2 Execution Region내의 Object 들의 순서

Linker는 Scatter Loading File의 Execution Region

내에 명시된 Object의 List를 scan하여 Memory에 적재될 위치를 명시 하는데, Link될 Object들의 속성과 Scatter Loading File을 기초로 적당한 Variant를 선택한다. Scatter Loading File내에서 각 Region에 명시된 Object들은 Compile-Time에 Make File에 정의된 순서대로 구성되며, Linker는 생성된 Object들을 Section 단위로 쪼개어, Scatter Loading File에 명시된 Object와 일치하는 섹션들의 위치를 결정하게 된다. 따라서 Execution Region 내의 List-up된 Object들의 순서를 바꾸는 것은 Delta 크기와 관련이 없다. 그림 6은 A라는 Module을 구성하는 Object App1,App2,App5,App4,App6,App3이 순서대로 Make File내에서 명시되어 있을 때, 이 모듈을 Scatter Loading에서 Object의 List-up순서를 바꾸어 명시하더라도, ELF 내에서의 List-up 순서는 Make File에서 명시한 순서대로 구성됨을 나타낸다.

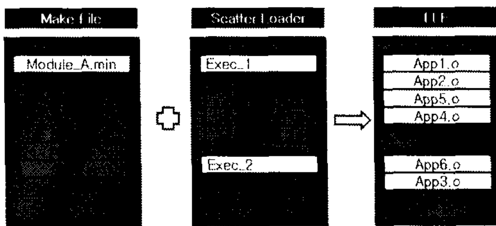

그림 6. Make File과 Scatter Loading에 명시된 Object List 순서와 ELF의 관계

3.3 Execution Region 의 개수

Scatter Loading File 작성 시, Delta Size 변화에 가장 큰 영향을 미치는 것은 Execution Region의 개수이다. 여러 개의 Object를 포함하는 Execution Region이 있을 때, 하나의 Region으로 묶는 것 보다, 각각의 Object를 하나의 Region으로 분할할수록, Delta의 Size는 작아진다. 그림 7은 Execution Region 의 개수가 1개일 때와 4개일 때, Application1.o에서 수정사항이 발생한 경우에 대한 도식이다. 1개의 Execution Region에서는 수정사항이 발생한 위치부터 Gap내의 모든 address가 수정사항에 의해 변경된다. 그러나 각각의 Object당 하나의 Region으로 분리하면, Application1에서 수정사항이 발생했을 때, 다음 Region의 Address가 FIXED Keyword에 의해 고정되어 있으므로, 수정사항이 발생한 Address부터 첫 번째 Region의 Gap까지만 변경된다. 실제 Execution 영역의 개수를 1개, 2개, 3개, 4개로 나누면서 동일한 수정사항에 대한 Delta Size를 확인한 결과, Execution 영역이 쪼개어 질수록, Delta Size는 작아지는 결과를 얻을 수 있었다.

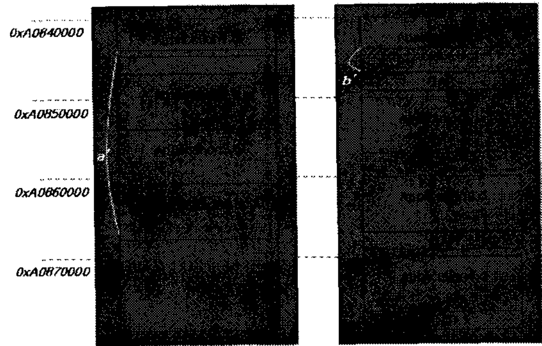

그림 7. Execution Region 1개와 4개일 때, Delta 크기 비교

a' > b'

3.4 Execution Region 내의 Function Dependency

Firmware의 Binary에서 수정사항이 발생했을 때, 서로 관련성(Dependency) 있는 Object가 수정될 가능성이 높다. 그림 8을 보면, Application1~4의 Object가 함께 수정될 수 있는 가능성이 있을 때, 수정사항이 각각의 Region으로 분산되어 있을 때보다 하나의 Region으로 묶여 있을 때, Delta의 Size가 작아짐을 알 수 있다. 또 Arm Linker는 처리속도를 향상 시키기 위해서 ARM Code와 Thumb Code를 Linking하기 위한 veneer code를 생성하는데, Branch Instruction을 확장하기 위해 Long Branch Veneer Code를 생성하기도 한다. 함수의 호출관계가 있는 Object를 서로 다른 Execution 영역에 분포시켰을 때에, Veneer Code가 증가하므로 이것은 결과적으로 Code Size가 증가된다. 따라서 Dependency가 있는 모듈끼리 묶은 Scatter Loading File과 Random 순서의 Object를 묶은 Scatter Loading File을 같은 크기로 나누었을 때, 실제 Dependency 별로 합친 영역에서 발생한 Delta Size가 그렇지 않은 영역보다 40KB 가량

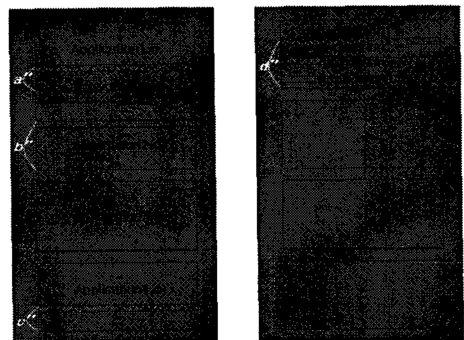

그림 8. Dependency 별 Delta 크기 비교

감소되는 것을 확인할 수 있었다.

a'' + b'' + c'' > d''

4. 결 론

FOTA Binary로 Upgrade하기 위해서 Scatter Loading File을 조정할 때, Dependency가 있는 Object끼리 묶을수록, Region의 개수를 분할할수록 Delta Size를 줄일 수 있다는 것을 알 수 있었다. 또 Scatter Loading File내에서 Object의 Type 별로 묶는 것과 같은 Region 내의 순서를 변경하는 것은 Delta 크기에 영향을 미치지 않는다는 것을 알 수 있었다. 그림 9와 그림 10은 Delta Size와 Scatter Loading File 내의 Region의 개수, Dependency 와의 관계를 그래프로 나타낸 것이다.

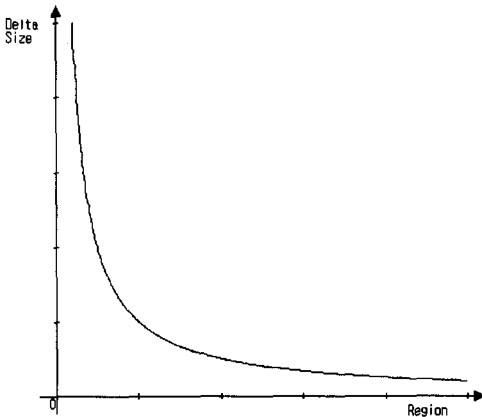

그림 9. Region 개수와 Delta Size의 관계

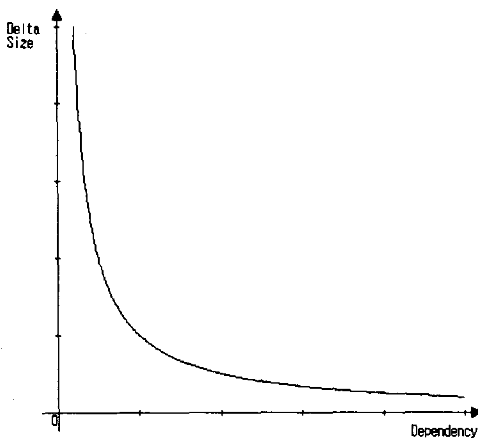

그림 10. Dependency와 Delta Size의 관계

FOTA Binary의 Delta Size를 줄이기 위해서 가장 주목해야할 점은 수정사항이 발생한 위치부터 고정된 다음 Object까지 List-up 된 모든 Object들의 Address가 변경된다는 점이다. Scatter Loading File내의 Execution Region을 최대한 나누어 Object의 Address를 고정시킴으로서, 다음 영역의 Object들이 수정에 의해 최소한의 영향을 받을 수 있도록 하는 것이 가장 중요하다. 그러나 Mobile Device의 특성상 Memory의 Size는 Device의 가격 상승과 밀접한 관련이 있기 때문에, 무한하게 확장할 수 없으며, 한정된 Memory내에서 Region의 개수를 증가시킬수록 보존되어야 할 Gap의 크기가 줄어들게 되는 단점이 있다. 또 Function 간의 Dependency를 파악하여 향후 바이너리의 수정 가능성과 방향성을 예측하는 것 또한 매우 중요하다. 각 개발 모델마다 Version 발행 시에 발생되는 수정사항에 대한 자료를 토대로, 바이너리의 수정 방향성과 가능성을 근거로 하여 Scatter Loading File의 각 Region 별로 List-up 할 Object의 관련성과 Gap의 Size를 조정할 수 있는 통계적인 분석이 필요할 것이다.

5. 참고 문헌

[1] Open Mobile Alliance Device Management Ver. 1.2

[2] Over-The-Air Mobile Device Management Specification Ver. 8.0 , Verizon Wireless , 2007

[3] Open Mobile Alliance Download Over The Air Ver. 2.0

[4] ARM Developer Suite, Linker and Utilites Guide, ARM , 2001

[5] CompTool User Guide, Innopath Software Inc, 2006

[6] Jone R. Levine, Linkers & Loaders , Morgan Kaufmann Publishers , 2000

[7] Sandeep Grover, Linkers and Loaders, Linux Journal, 2002