ISSN 1598-0170 (Print)
ISSN 2287-1136 (Online)
http://www.jksii.or.kr

모바일 클라우드 컴퓨팅을 위한 실용적인 오프로딩 기법 및 비용 모델☆

Pratical Offloading Methods and Cost Models for Mobile Cloud Computing

박 민 균　　Piao Zhen Zhe[1]　　라 현 정[1*]　　김 수 동[1]
Min Gyun Park　　Piao Zhen Zhe　　Hyun Jung La　　Soo Dong Kim

요 약

제한된 모바일 디바이스의 자원을 해결하기 위해, 클라우드에 있는 서비스 또는 자원을 활용하는 모바일 클라우드 컴퓨팅(Mobile Cloud Computing, MCC) 연구가 활발히 진행되고 있다. MCC에서는 주로 기능 컴포넌트를 다른 노드로 오프로딩 (Offloading) 시킴으로써, 모바일 노드의 자원 문제를 해결하는 접근법을 주로 사용한다. 그러나, 현재 진행되고 있는 MCC에 대한 연구는 사전에 결정된 노드로 오프로딩 시키는 기법들이 주로 진행되고 있으며, 개념적인 수준에서 기법이 제시되고 있다. 본 논문에서는 복잡도가 높은 모바일 애플리케이션의 성능을 보장하기 위한 4가지 종류의 오프로딩 기법을 제안한다. 제시된 기법은 구현이 가능하도록 실용적인 수준으로 설계되며, 비용 모델을 제시하여 오프로딩을 통한 성능향상이 있음을 정량적으로 증명한다.

주제어 : 모바일 클라우드 컴퓨팅, 정적 오프로딩, 동적 오프로딩, 전체 오프로딩, 부분 오프로딩, 평가 모델

ABSTRACT

As a way of augmenting constrained resources of mobile devices such as CPU and memory, many works on mobile cloud computing (MCC), where mobile devices utilize remote resources of cloud services or PCs, /have been proposed. A typical approach to resolving resource problems of mobile nodes in MCC is to offload functional components to other resource-rich nodes. However, most of the current woks do not consider a characteristic of dynamically changed MCC environment and propose offloading mechanisms in a conceptual level. In this paper, in order to ensure performance of highly complex mobile applications, we propose four different types of offloading mechanisms which can be applied to diverse situations of MCC. And, the proposed offloading mechanisms are practically designed so that they can be implemented with current technologies. Moreover, we define cost models to derive the most sutilable situation of applying each offloading mechanism and prove the performance enhancement through offloadings in a quantitative manner.

☞ keyword : Mobile Cloud Computing, Static Offloading, Dynamic Offloading, Full Offloading, Partial Offloading, CostModel

1. 서 론

모바일 디바이스는 크기가 작기 때문에 PC에 비해 컴퓨팅 자원이 부족하다. 그 결과 자원 소비가 많고 높은 복잡도를 가진 애플리케이션은 모바일 디바이스에서 설치 및 운영되기 어렵다. 그러나, 모바일 디바이스는 풍부한 네트워크 연결 능력이 있어 외부 자원을 사용하기가 용이하므로, 이런 모바일 클라우드 컴퓨팅 (Mobile Cloud Computing, MCC) 연구가 활발히 진행되고 있다 [1][2][3].

MCC는 모바일 디바이스의 유연한 네트워크 연결성을 이용하고 클라우드 환경을 이용한 애플리케이션이 오프로딩이나 서비스의 동적 이동 등을 통해, MCC의 다양한 문제를 해결하는 컴퓨팅 패러다임이다[4][5]. MCC는 품질을 일정 수준으로 관리하는 데에 다음의 이슈를 가지고 있다.

- 모바일 클라우드 환경은 애플리케이션 및 서비스 구성, 네트워크 환경 등이 매우 동적으로 변화함.
- 모바일 애플리케이션의 제한된 자원을 해결하는 효과적인 방법이 필요함.
- 모바일 애플리케이션이 구독하는 서비스는 제한된 가시성 및 관리성을 가지고 있으므로, 이를 해결하기

1 Department of Computer Science, Soongsil University, 511 Sangdo-Dong Dongjak-Ku Seoul, 156-743, Korea
* Corresponding author (hjla80@gmail.com)
[Received 22 January 2013, Reviewed 4 February 2013, Accepted 18 February 2013]
☆ 이 논문은 2012년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(2012R1A1B3004130, 2012R1A6A3A01018389).

http://dx.doi.org/10.7472/jksii.2013.14.2.73

위한 효과적인 방법이 필요함.

MCC에서 발생되는 문제는 대부분 동적으로 환경이 자주 변동되기 때문에 발생하며, 대부분 이동성이 높은 모바일 디바이스가 그 원인을 제공한다. 그러므로, 제한된 자원 또는 불안정한 네트워크를 가지는 모바일 디바이스로부터 일부 컴포넌트를 외부 디바이스로 오프로딩 시킴으로써, 낮은 성능 등의 문제를 해결할 수 있다. 즉, MCC에서는 주로 기능 컴포넌트를 다른 노드로 오프로딩(Offloading) 시킴으로써, 모바일 노드의 자원 문제를 해결하는 접근법을 주로 사용한다. 현재까지 진행된 오프로딩 기법은 다음과 같은 한계점을 가지고 있다.

- 동적으로 변화가 심한 MCC 환경에서 현재 상황을 고려하지 않고, 사전에 결정된 목표 노드로 애플리케이션의 전체 또는 부분을 오프로딩함.
- 모바일 애플리케이션은 고유의 비즈니스 기능과 오프로딩 실행 관련 기능이 혼합되어 설계됨.
- 현재 모바일 애플리케이션의 품질 상황에 맞게 최적의 오프로딩이 실행하는 것을 고려하지 않음.

위의 한계점을 해결하기 위해, 본 논문에서는 MCC 환경에서 실용적으로 오프로딩을 설계하는 기법을 제안한다. 먼저, MCC의 다양한 문제를 해결하기 위해, 다양한 상황에 적용 가능한 4가지 종류의 오프로딩을 소개한다. 각 오프로딩 종류 별로 구조 및 런타임 실행 절차를 기술하고, 각 오프로딩을 수행하는데 소요되는 시간을 평가하는 비용 모델을 제안한다. 제안된 비용 모델은 각 오프로딩으로 인한 이점을 최대화할 수 있는 상황을 도출하는데 사용된다. 그리고, 애플리케이션에 영향을 주지 않고 오프로딩을 수행하는 노드매니저를 설계하기 위한 기법을 제시한다. 마지막으로, 실험을 통하여 네 가지 종류의 오프로딩 별 성능 이점을 비교한다. 비용 모델에서 정의한 여러 요소들을 다양하게 하여 여러 실험 프로그램을 작성하고, 각 요소들이 오프로딩에 미치는 영향을 분석하여 상황 별 가장 적절한 오프로딩을 선정할 수 있는 기준을 제시하도록 한다.

2. 관련 연구

Cuerov의 연구에서는 목표 애플리케이션의 수정을 최소화하고, 모바일 디바이스의 에너지 효율을 높이기 위해 자원이 풍부한 서버 노드로 오프로드시키기 위한 MAUI 시스템 아키텍처를 제안하였다 [6]. MAUI는 Proxy, Profiler, Solver라는 3개의 컴포넌트로 구성된다. Profiler는 장치 프로파일, 프로그램 프로파일, 네트워크 프로파일을 이용하여 해당 메소드 호출을 로컬 또는 원격에서 실행시킬지를 결정하고, Solver는 에너지 소비량과 지연시간 (Latency)을 고려하여 최적화된 프로그램 분할 전략을 수립한다. 리플렉션(Reflection) 기법을 이용하여, 서버 측에 위치한 Proxy는 원격 메소드를 추출하여 이를 실행시킨다. 이 연구는 정적 오프로딩만 고려하고 있어, 제한된 상황에만 해당 기법을 적용할 수 있다.

Kanad Sinha와 Milind Kulkarni의 연구에서는 모바일 노드와 서버 노드에서 프로그램을 분할하기 위한 알고리즘을 제시한 후 모듈화된 컴포넌트들을 여러 서버 노드에 오프로딩하는 기법을 제시하였다 [7]. 그러나, 이 연구에서는 시뮬레이션을 진행한 실험데이터들을 기반으로 알고리즘의 효율성을 증명하고는 있었지만, 동적 오프로딩 시 모바일 노드와 서버 노드간의 플랫폼이 다른 경우 이를 고려한 오프로딩은 고려하지 않았다.

Byung-Gon Chun등의 연구에서는 동적으로 작업량을 적응시키기 위해, 제약된 자원을 가지는 모바일 디바이스와 클라우드 간에 애플리케이션을 동적으로 분할하는 방법을 제안하였다 [8]. 그리고, 서버 노드에 가상머신을 설치시킨 후 모바일 노드에서 실행되는 프로그램을 서버노드에 이주시켜 오프로딩을 진행하는 CloneCloud시스템을 제안하였다 [9]. 이 시스템은 Static Analyzer, Dynamic Profiler, Optimization Solver로 구성된 프로그램 분할-분석 프레임워크를 설계하여 컴포넌트 오프로딩 기법을 제안하였다. 그러나, 두 연구는 모두 동적-부분 오프로딩만 고려하며, 핵심 알고리즘은 개략적으로 정의하였다

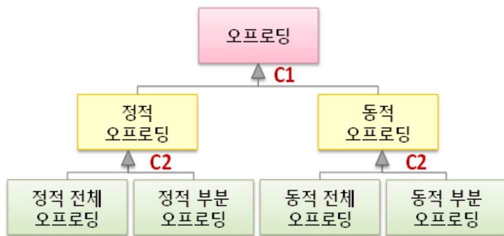
Karthik Kumar와 Yung-Hsiang Lu의 연구에서는 모바일 디바이스의 자원부족 등 문제점을 제시하고 오프로딩을 적용할때와 적용하지 않을때의 모바일 디바이스의 배터리 사용관련 메트릭을 정의하여 비교하였다 [10]. 이 연구는 오프로딩을 적용할 때의 성능 향상을 수식적으로 증명하지만, 오프로딩을 위한 설계 기법 등은 다루고 있지 않다. Wolski는 오프로딩을 수행하는 오프로딩 결정(Offloading-Decision) 프레임워크를 제시한다 [11]. 그리고, 제시한 프레임워크를 기반으로 실험을 진행하여 관련 데이터들을 추출하였고, 제시한 프레임워크가 그리드 환경에서 오프로딩 기법의 유효성을 실험결과로 보여주고 있다. 그러나, 모바일 고유의 특성을 고려하여 기법이나 프레임워크를 수정해야 한다.

3. 오프로딩 (Offloading) 기법의 분류

일반적으로 오프로딩 기법은 오프로딩 실행에 따른 오버헤드가 발생한다. 오프로딩으로 인한 오버헤드에 영향을 주는 요소를 고려하여 다음 2가지 기준을 정의한다.

- C1. 오프로드 될 컴포넌트를 전송하는 시점: 오프로드 될 컴포넌트를 사전에 또는 필요시 동적으로 전송하는 경우에 따라 런타임 오버헤드에 영향을 준다.
- C2. 오프로드 될 컴포넌트의 비율: 오프로드 되는 컴포넌트는 네트워크를 통해 전송되므로, 그 크기가 작을수록 네트워크 오버헤드가 감소한다. 이를 고려하여 전체 애플리케이션 또는 애플리케이션의 일부를 오프로드 할 수 있다.

C1에 따라 정적(Static) 오프로딩과 동적 (Dynamic) 오프로딩으로 분류하고, C2에 따라 정적 및 동적 오프로딩을 전체 (Full) 오프로딩과 부분 (Partial) 오프로딩으로 분류하며, 그 결과는 (그림 1)과 같다.



(그림 1) 4가지 종류의 오프로딩
(Figure 1) Four Types of Offloading Scheme

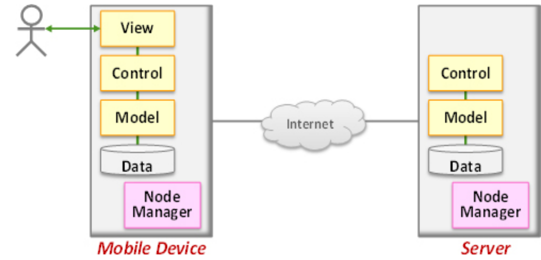
4. 정적 오프로딩 (Static Offloading)

대부분의 애플리케이션은 MVC 패턴(Model-View-Control)을 적용하여 개발되므로, MVC 패턴을 기반으로 한 정적 오프로딩의 구조와 오프로딩으로 인한 실행 시간을 측정하는 비용 모델을 제시한다.

4.1. 정적-전체 오프로딩(Static Full Offloading)

4.1.1. 정적-전체 오프로딩 개요

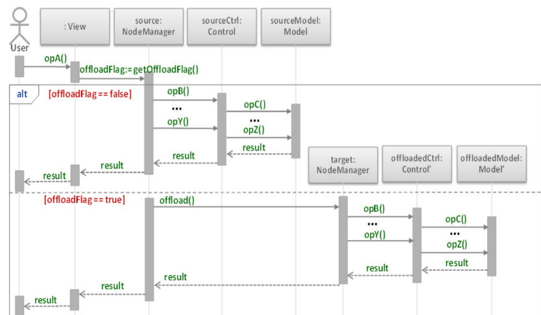
정적-전체 오프로딩은 사전에 오프로드 가능한 전체 애플리케이션을 서버 노드에 위치시키며, 이를 위한 구조는 (그림 2)와 같다.



(그림 2) 정적-전체 오프로딩의 정적 뷰
(Figure 2) Structure of Static-Full Offloading

모바일 디바이스에는 전체 애플리케이션의 Model, Control, View 계층과 Data를 배치한다. 서버에는 Control, Model 계층에 속한 컴포넌트와 Data를 배치하고 있다. View 계층은 사용자와 밀접한 상호작용을 하기 때문에, 이를 오프로드 할 경우 네트워크 오버헤드가 크므로 오프로딩 범위에서 제외한다.

모바일 디바이스에 특정 문제가 발생하면 정적-전체 오프로딩이 실행되며, (그림 3)은 이에 대한 절차를 보여준다. 이 그림은 소스 노드에 있는 애플리케이션(Application_n)의 opA()가 사용자 요청을 받으면, 내부적으로 opB()⋯opZ()를 실행하여 결과값을 반환한다고 가정하고 작성된 것이다. 사용자가 소스 노드에 있는 애플리케이션의 opA()를 요청하면 노드매니저인 source:NodeManager가 호출된다. source:NodeManager에서 getOffloadFlag()가 호출되어 오프로딩 수행여부를 판단한다. 만약 오프로딩이 필요한 경우에 offload()를 통해 사용자의 입력값을 서버로 전달한다. 그리고, 서버는 opB()⋯opZ()의 기능을 수행하고, 이에 대한 결과값을 모바일 디바이스에 반환한다.



(그림 3) 정적-전체 오프로딩의 동적 뷰
(Figure 3) Behavior of Static-Full Offloading

4.1.2. 정적-전체 오프로딩 비용 모델

본 장에서는 정적 오프로딩으로 인한 QoS 이점을 수식을 통해 증명한다. 먼저, 오프로딩을 실행하지 않는 경우의 비용 모델을 위해, 다음과 같이 관련 용어를 정의한다.

- 모바일 노드의 프로세서 속도와 서버 노드의 프로세서 속도: S_{Mobile}와 S_{Server}
- 모바일 노드에서 수행되는 전체 작업량 (Workload): W_{Mobile}
- 애플리케이션이 수행할 전체 작업량: W_{Total}

오프로딩이 실행되지 않은 경우 실행시간 ($RT_{withoutOffloading}$)은 다음과 같다. 여기서, 전체 작업량이 모바일 노드에서 실행되므로, $W_{Total} = W_{Mobile}$이다.

$$RT_{withoutOffloading} = \frac{W_{Total}}{S_{Mobile}}$$

정적-전체 오프로딩 비용 모델을 정의하기 위해, 다음의 용어를 추가적으로 정의한다.

- 모바일과 서버 간 네트워크 대역폭: $Bandwidth\,(Mobile, Server)$ 또는 $Bandwidth\,(Server, Mobile)$
- 모바일 노드에서 서버 노드로 전송되는 사용자 입력값의 크기: $Size\,(InputData)$
- 서버 노드에서 모바일 노드로 전송되는 오프로드 프로그램 반환값 크기: $Size\,(ReturnValue)$
- View 계층 작업량 (모두 모바일 노드에서 실행됨): $W.View_{Mobile}$
- Model 계층 작업량 (정적-전체 오프로딩 시 서버 노드에서 실행됨): $W.Model_{Server}$
- Control 계층 작업량 (정적-전체 오프로딩 시 서버 노드에서 실행됨): $W.Control_{Server}$
- 정적-전체 오프로딩 시 서버 노드에서 수행되는 작업량: $W_{Server} = W.Model_{Server} + W.Control_{Server}$

이 경우의 총 실행시간($RT_{staticFullOffloading}$)은 모바일 노드와 서버노드에서 작업을 처리하는 시간, 모바일 노드와 서버 노드 간의 데이터 전송시간을 포함하며, 이는 다음과 같다.

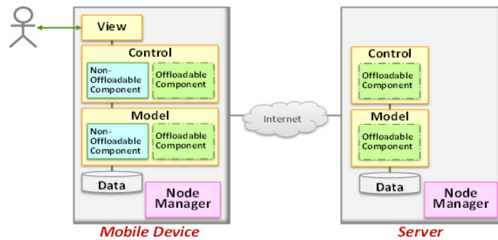
$$RT_{staticFullOffloading} = \frac{W.View_{Mobile}}{S_{Mobile}} + \frac{Size\,(InputData)}{Bandwidth\,(Mobile, Server)} + \frac{(W.Control_{Server} + W.Model_{Server})}{S_{Server}} + \frac{Size\,(ReturnValue)}{Bandwidth\,(Server, Mobile)}$$

$S_{Mobile} \ll S_{Server}$이기 때문에, W.Control과 W.Model을 처리하는 시간은 $(W.Control_{Mobile} + W.Model_{Mobile})/S_{Mobile} \gg W_{Server}/S_{Server}$가 된다. 정적-전체 오프로딩의 이점을 높이기 위해, $Size\,(InputData)$이 작고, $Bandwidth\,(Server, Mobile)$ 또는 $Bandwidth\,(Mobile, Server)$가 커야 한다.

4.2. 정적-부분 오프로딩(Static Partial Offloading)

4.2.1. 정적-부분 오프로딩 개요

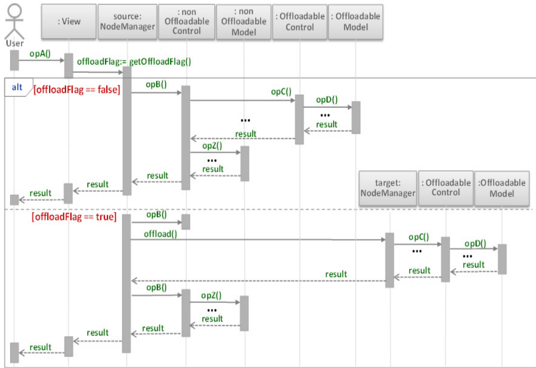
정적-부분 오프로딩은 사전에 오프로드 가능한 부분 애플리케이션을 서버 노드에 위치시키며, 이를 위한 구조는 (그림 4)와 같다.



(그림 4) 정적-부분 오프로딩의 정적 뷰
(Figure 4) Structure of Static-Partial Offloading

모바일 디바이스 노드에 전체 애플리케이션의 Model, View, Control 계층이 설치되어 있으며, 서버 노드에는 Control과 Model 계층이 설치되어 있는 구조는 정적-전체 오프로딩과 유사하다. 그러나, 모바일 노드의 Control과 Model 계층이 오프로딩 가능한 컴포넌트(Offloadable Component)와 오프로드 가능하지 않은 컴포넌트 (Non-Offloadable Component)로 분리되어 있으며, 서버 측에는 오프로드 가능한 부분만 배치되어 있음이 다르다. 오프로딩 가능한 컴포넌트는 1) 오프로드 가능하지 않은 컴포넌트 메소드 실행 전, 2) 오프로드 가능하지 않은 컴포넌트 메소드 실행 중간, 3) 오프로드 가능하지 않은 컴포넌트 메소드 실행 후에 호출될 수 있다. (그림 5)는 2번째 경우의 정적-부분 오프로딩 수행절차를 보여준다.

전체적인 수행절차는 정적-전체 오프로딩과 유사하지만, 오프로딩이 필요한 경우에 source:NodeManager는 오프로드 가능하지 않은 컴포넌트 실행 전후에 모바일 디바이스가 아니라 서버 노드에 설치된 Offloadable Control과 Offloadable Model의 opC(), opD()… 메소드를 호출한다는 점이 다르다.



(그림 5) 정적-부분 오프로딩의 동적 뷰
(Figure 5) Behavior of Static-Partial Offloading

4.2.2. 정적-부분 오프로딩 비용 모델

정적-부분 오프로딩의 비용 모델 정의를 위해, 오프로드 되어 있는 컴포넌트와 오프로드 되지 않은 컴포넌트에 대한 용어를 다음과 같이 정의한다.

- 모바일 노드에서 오프로드 되지 않는 Control 계층의 작업량: $W.Control_{Mobile}^{Fixed}$
- 모바일 노드에서 오프로드 되지 않는 Model 계층의 작업량: $W.Model_{Mobile}^{Fixed}$
- 서버 노드에 오프로드 되는 Control 계층의 작업량: $W.Control_{Server}^{Offloadable}$
- 서버 노드에 오프로드 되는 Model 계층의 작업량: $W.Model_{Server}^{Offloadable}$
- 모바일 노드에서 일부분 처리한 결과값: $Size\,(States)$

정적-부분 오프로딩이 실행되는 경우의 총 실행시간 ($RT_{staticPartialOffloading}$)은 다음과 같이 정의한다.

$$RT_{staticPartialOffloading} = \frac{\left(W.View_{Mobile} + W.Control_{Mobile}^{Fixed} + W.Model_{Mobile}^{Fixed}\right)}{S_{Mobile}} + \frac{Size\,(InputData) + Size\,(States)}{Bandwidth\,(Mobile, Server)} + \frac{\left(W.Control_{Server}^{Offloadable} + W.Model_{Server}^{Offloadable}\right)}{S_{Server}} + \frac{Size\,(ReturnValue)}{Bandwidth\,(Server, Mobile)}$$

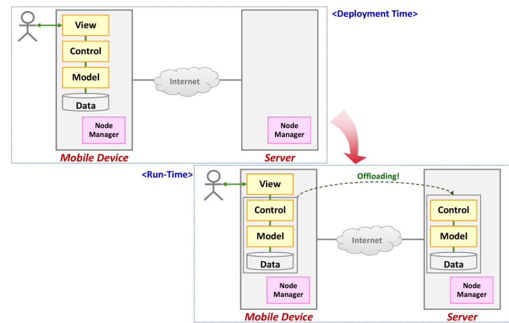
$S_{Mobile} \ll S_{Server}$이기 때문에, $W_{Mobile} < W_{Server}$의 경우라면 $W_{Mobile}/S_{Mobile} \gg W_{Server}/S_{Server}$가 된다. 정적-부분 오프로딩의 이점을 높이기 위해서는 W_{Total} 중 W_{Server}가 많은 비율을 차지하고, *Size(InputData)*와 *Size(States)*가 작아야 한다.

5. 동적-오프로딩(Dynamic Offloading)

5.1. 동적-전체 오프로딩(Dynamic Full Offloading)

5.1.1. 동적-전체 오프로딩 개요

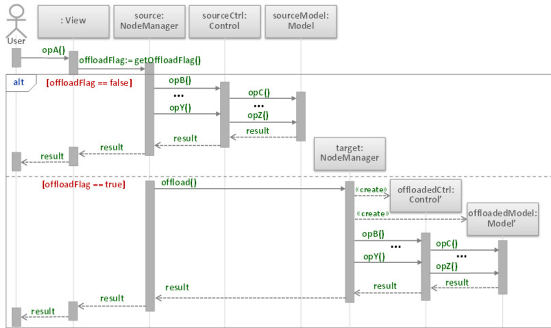
동적-전체 오프로딩 기법은 런타임시에 전체 애플리케이션을 오프로딩 한다. (그림 6)은 동적-전체 오프로딩의 정적 뷰를 보여주며, 런타임시에 Control과 Model 계층의 전체 컴포넌트와 이 애플리케이션이 사용하는 Data가 서버 노드에 오프로드 되어 설치됨을 확인할 수 있다.



(그림 6) 동적-전체 오프로딩의 정적 뷰
(Figure 6) Structure of Dynamic-Full Offloading

(그림 7)은 동적-전체 오프로딩을 수행하는 동적 뷰를 보여준다.

동적 전체 오프로딩의 전체적인 흐름은 정적-전체 오프로딩과 유사하지만, 오프로딩 수행 시 target:NodeManager가 source:NodeManager로부터 offload()의 매개변수로 전달된 오프로드 될 컴포넌트 패키지(즉, Control와 Model 계층 컴포넌트)를 설치하고 인스턴스를 생성한 뒤 opB()⋯ opZ()를 호출한다는 점이 다르다.



(그림 7) 동적-전체 오프로딩의 동적 뷰
(Figure 7) Behavior of Dynamic-Full Offloading

5.1.2. 동적-전체 오프로딩 비용 모델

동적-전체 오프로딩의 비용 모델을 정의하기 위해, 다음의 용어를 추가적으로 정의한다.

- 오프로드 될 컴포넌트 패키지: $Size\,(OffPackage)$

동적 오프로딩이 실행되는 경우, 정적 오프로딩 수행 시 필요한 시간 외에 오프로드 된 컴포넌트 전송 시간을 고려해야 하며, 동적-전체 오프로딩의 총 실행시간 ($RT_{dynFullOffloading}$)은 다음과 같다.

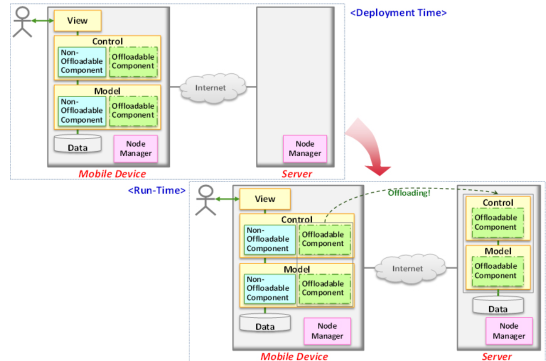
$$RT_{dynFullOffloading} = \frac{W.View_{Mobile}}{S_{Mobile}} + \frac{Size\,(OffPackage) + Size\,(InputData)}{Bandwidth\,(Mobile, Server)} + \frac{(W.Control_{Server} + W.Model_{Server})}{S_{Server}} + \frac{Size\,(ReturnValue)}{Bandwidth\,(Server, Mobile)}$$

정적-전체 오프로딩에서 고려한 사항 이외에 동적-전체 오프로딩의 이점을 높이기 위해서는 *Size(OffPackage)*의 값이 작아야 한다.

5.2. 동적-부분 오프로딩(Dynamic Partial Offloading)

5.2.1. 동적-부분 오프로딩 개요

동적 부분 오프로딩은 런타임시 오프로드 가능한 애플리케이션의 일부분을 서버에 전송 및 배치시키며, 이를 위한 구조는 (그림 8)과 같다. 전반적인 구조는 정적-부분 오프로딩과 유사하지만, 오프로드 가능한 컴포넌트들이 런타임 시에 서버노드로 전송되고 배치되었다는 부분이 다르다.



(그림 8) 동적 부분 오프로딩의 정적 뷰
(Figure 8) Structure of Dynamic-Partial Offloading

전체적인 수행절차는 정적-부분 오프로딩과 유사하지만, *target:NodeManager*가 설치에 필요한 *Offloadable Control*과 *Offloadable Model* 컴포넌트들의 일부분을 전달받은 후, 설치 및 배치된다. 그리고 정적-부분 오프로딩과도 유사하지만, 소스 노드의 결과값을 받은 후, (그림 7)의 opB()⋯ opZ()와 같은 나머지 기능을 수행함으로써, 사용자의 요청에 따른 최종값을 반환한다는 점이 다르다.

5.2.2. 동적-부분 오프로딩 비용 모델

동적-전체 오프로딩과 유사하게, 정적-부분 오프로딩 수행 시 필요한 시간 외에 오프로드 된 컴포넌트 전송 시간을 고려해야 하며, 동적-부분 오프로딩의 총 실행시간 ($RT_{dynPartialOffloading}$)은 다음과 같다.

$$RT_{dynPartialOffloading} = \frac{(W.View_{Mobile} + W.Control^{Fixed}_{Mobile} + W.Model^{Fixed}_{Mobile})}{S_{Mobile}} + \frac{Size\,(InputData) + Size\,(OffPackage) + Size\,(States)}{Bandwidth\,(Mobile, Server)} + \frac{(W.Control^{Offloadable}_{Server} + W.Model^{Offloadable}_{Server})}{S_{Server}} + \frac{Size\,(ReturnValue)}{Bandwidth\,(Server, Mobile)}$$

동적-부분 오프로딩의 경우 동적-전체 오프로딩과 비교시 $OffPackage$ 크기가 작다. 오프로딩의 이점을 높이기 위해서는, 서버 노드에서 실행되는 작업량이 모바일 노드보다 상대적으로 커야 하며, *OffPackage*의 크기는 작을수록 유리하다. 그리고 $Size\,(States)$의 값이 작아야 한다.

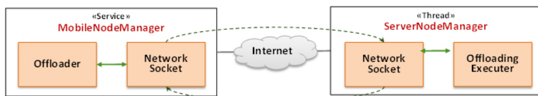
6. 오프로딩 적용 기법

오프로딩 적용 기법은 사용되는 프로그래밍 언어에 매우 의존적이다. 본 논문에서는 안드로이드 플랫폼과 자바

를 고려하여 오프로딩 적용 기법을 제안한다.

6.1. 노드 매니저(Node Manager) 설계 기법

오프로딩을 런타임에 수행하기 위해서 노드 매니저의 역할이 매우 중요하다. 노드 매니저는 백그라운드 형태로(Background) 각 노드에 상주하며, 특정 문제가 발생하면 오프로딩을 수행해야 한다. 이를 위해, 자바의 쓰레드(Thread)와 안드로이드 플랫폼의 서비스(Service)를 이용하여 노드 매니저를 설계한다. (그림 9)는 모바일 노드와 서버 노드에 설치되어 있는 노드 매니저의 전체 구조를 나타낸다.



(그림 9) 모바일 노드와 서버 노드의 노드 매니저의 전체 구조
(Figure 9) Overall Structure of Node Managers on Mobile Node and Server Node

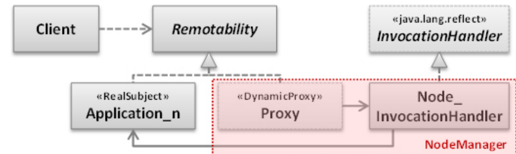
Offloader는 모바일 디바이스에 설치된 각 View, Control, Model, Data와 상호작용을 통하여 필요한 기능을 요청하며, Network Socket에게 데이터를 제공 또는 전달하는 기능을 한다. Network Socket는 실제로 모바일 노드와 서버 노드간의 통신을 수행하는 역할을 한다. 소켓 통신을 사용하여 오프로딩에 필요한 데이터 및 파일을 전송하고, 결과값을 전송받는 역할을 한다.

Offloading Executer는 제네릭(Generic) 클래스를 사용하여 전달받은 데이터의 타입에 따라서 4가지 종류의 오프로딩을 범용적으로 수행한다. 먼저 Network Socket으로부터 전달 받은 데이터 타입을 확인하여 데이터를 객체 또는 파일로 구분한다. 데이터가 객체일 경우에는 정적 오프로딩을 수행하며, 파일인 경우에는 동적 오프로딩을 수행한다.

6.2. Proxy 패턴 기반의 투명한 방식의 오프로딩 수행

오프로딩 수행이 사용자에게 투명하게 수행되기 위해서, 노드 매니저는 모바일 애플리케이션의 호출을 가로채어 그 기능을 수행해야 한다. 백그라운드로 실행되고 있는 노드 매니저가 사용자의 요청을 상황에 알맞게 처리하기 위해 Proxy패턴을 적용한다.

(그림 10)은 동적 프록시를 지원하는 java.lang.reflect.proxy[12]를 이용해 NodeManager를 설계한 모델을 보여준다. NodeManager는 java.lang.reflect.InvocationHandler를 구현한 Node_InvocationHandler가 정의한 invoke( ) 메소드를 통해 오프로딩 여부에 따라 Application_n의 메소드를 실행한다.



(그림 10) java.lang.reflect.proxy 이용한 노드매니저
(Figure 10) A Design of NodeManager by using java.lang.reflect.proxy

목록 1은 java.lang.reflect.proxy를 사용하여 노드 매니저의 새로운 인스턴스가 생성되는 소스 코드의 일부분을 보여준다. 생성된 인스턴스를 사용함으로써, 노드 매니저가 호출을 가로 채어 소스노드의 QoS 저하, 자원 부족 등 특정 문제가 발생했을때 오프로딩 수행을 결정한다. 오프로딩을 수행한 결과 값은 사용자에게 투명한 방법으로 수행되었기 때문에 오프로딩 여부를 알수 없게 된다.

(목록 1) 노드매니저 인스턴스 생성 코드
(List 1) Code Snippet for Creating NodeManager Instances

```
1.  public static Object newInstance (Object obj){
2.    return     java.lang.reflect.Proxy.newProxyInstance(obj.getClass().getClassLoader(),
    obj.getClass().getInterfaces(), new NodeManager(obj)); }
```

6.3. 플랫폼에 독립적인 동적 오프로딩 실행

동적 오프로딩은 런타임에 애플리케이션을 동적으로 설치 및 실행해야 하기 때문에, 그 설계가 플랫폼에 매우 의존적이다. 그러므로, 서버 노드의 플랫폼에 독립적인 오프로딩 설계가 필요하며, 이를 위해 다음 두가지 상황을 고려하여 동적 오프로딩을 설계한다.

- **동일한 플랫폼을 가지고 있지 않을 경우(자바 플랫폼 기반)의 서버 측 노드 매니저 설계**

서버 노드가 모바일 노드와 동일한 안드로이드 플랫폼이 아닌 자바 플랫폼 기반인 경우, 실시간으로 전송된 jar 파일에 포함되어있는 클래스 파일을 실행시키기 위해 URLClassLoader를 이용하고, 런타임시 실시간으로 바인

딩 시켜주기 위한 java Reflection을 사용하여 동적 오프로딩을 설계한다. 목록 2는 URLClassLoader를 사용한 동적 바인딩 소스코드의 일부분을 보여준다.

(목록 2) URLClassLoader를 이용한 동적 바인딩 소스코드
(List 2) Code Snippet for Dynamically Installing Classes with URLClassLoader

```
1.  public void DynamicBinding() throws Throwable   {
2.      URL classURL = new URL("jar:" + file.toURI().toURL() + "!/");
3.      URLClassLoader classLoader = new URLClassLoader(new URL [] {classURL});
4.      Class class = classLoader.loadClass("com.example.firstpgm. multiplymatrix ");
5.      Object obj = class.newInstance();
6.      Method setSize = class.getMethod("setSize", new Class[] {int.class});
7.      setSize.invoke(obj, new Object[] {receiveData.getSize()}); }
```

3번째 줄에서 생성된 classURL로 URLClassLoader의 인스턴스를 생성한다. 4번째 줄에서는 생성된 URLClassLoader 인스턴스를 이용하여 해당 클래스를 로드하며, 5번째 줄에서는 로드한 클래스를 사용하여 객체를 생성한다. 7번째 줄에서는 생성된 객체와 입력값을 이용하여 메소드를 호출한다.

• **동일한 플랫폼을 가지고 있을 경우(안드로이드 플랫폼 기반)의 노드 매니저**

서버 노드가 모바일 노드와 동일한 안드로이드 플랫폼을 설치하고 있는 경우, 실시간으로 전송된 apk파일을 설치하기 위해 해당 URL로 파일을 생성하고, intent를 사용하여 파일을 입력값으로 installer 를 호출한다. 목록 3은 apk파일을 설치하기 위한 installer 호출 소스 코드의 일부분을 보여준다.

(목록 3) apk파일을 설치하기 위한 installer 호출 코드
(List 3) Code Snippet for Dynamically Installing APK Files with Intent

```
1.  public void ApkInstaller(){
2.      File apkFile = new File(NodeManager.URL);
3.      Intent intent = new Intent(Intent.ACTION_VIEW);
4.      intent.setDataAndType(  Uri.fromFile(apkFile),  "application/vnd.android.package-
    archive");
5.       startActivity(intent); }
```

2번째 줄에서 전송된 apk파일을 사용하여 File 인스턴스를 생성한다. 3번째 줄에서는 Intent 인스턴스를 생성하며, 4번째 줄에서는 생성된 File 인스턴스를 이용하여 installer를 실행하기 위한 입력값을 설정한다. 5번째 줄에서는 설정된 intent를 이용하여 installer를 호출한다.

6.4. 네트워크 오버헤드를 줄이기 위한 효율적인 오프로딩 실행

오프로딩 비용모델에서 정의하였듯이, 모바일 노드와 서버 노드간의 네트워크 오버헤드가 적을수록 오프로딩 이점이 많다. 그러므로, 두 노드 간의 통신 횟수를 감소하도록 오프로딩을 설계해야 한다.

이를 위해, 오프로딩 되는 기능 수행 시 필요한 모든 데이터를 Data 클래스로 묶어서 서버 노드로 전송하며, Data 클래스는 네트워크를 통해 전송되기 위해 Serializable 인터페이스를 Implement하여 구현한다. Data클래스는 java.io.Serializable을 사용하였고, 내부에 Vector객체를 인스턴수 변수로 선언하여, 노드 매니저가 할당하는 데이터들을 패키지화 하여 전송함으로써 범용적인 수준으로 구현하였다. 목록 4는 모바일 노드매니저가 Data인스턴스를 서버 노드에 전송하는 소스 코드의 일부분을 보여준다.

(목록 4) 모바일 노드매니저 클래스에서 Data인스턴스를 전송해주는 메소드
(List 4) A Method for Transferring a Data Instance on Mobile Node

```
1.  public class NodeManager implements Runnable, InvocationHandler{
2.      private Data data;
3.      Public void sendData(){
4.          ObjectOutputStream oos = new ObjectOutputStream (socket.getOutputStream());
5.          oos.writeObject(data); } }
```

노드 매니저에서 Data객체를 생성후 서버 노드에 전송하여줌으로써 사용자가 입력하는 값들을 매번 전송하는 것이 아니라 정의한 인스턴스 변수에 할당한후 Data객체를 한번만 서버에 전송함으로써, 네트워크 오버헤드를 줄일 수 있도록 구현하였다.

7. 실용적인 오프로딩 실험 및 평가

본 장에서는 안드로이드 기반 모바일 환경과 윈도우즈 서버 환경을 기반으로 정적 오프로딩과 동적 오프로딩을 적용하였을때 실행시간의 결과를 평가하여 오프로딩 기법의 적용성과 유효성을 증명한다.

7.1. 실험 환경 및 설정

본 실험을 위해 각 구성 요소의 상세한 기술은 다음과 같다.

• 모바일 노드: Android 2.1 기반의 삼성 갤럭시 1, CPU

1 GHz, 메모리 512 MB

- 서버 노드: Windows 7 기반의 PC, CPU 인텔 i5 3GHz, 메모리 4GB
- 네트워크: 3G 또는 Wifi

테스트 프로그램의 종류는 4장과 5장에 제시된 메트릭을 기반으로 결정한다.

- 애플리케이션의 전체 작업량 W_{Total}(4.1.2절)
- W_{Total} 중 오프로드가 되지 않아 모바일 디바이스에서 실행되는 작업량 W_{Total}
  - W_{fixed}(전체 오프로딩 작업량) = $W.View_{Mobile}$ (4.1.2절)
  - W_{fixed}(부분 오프로딩 작업량) = $W.View_{Mobile}$ (4.1.2절) + $W.Control_{Mobile}^{Fixed}$(4.2.2절)+ $W.Model_{Mobile}^{Fixed}$(4.2.2절)
- W_{Total} 중 오프로드가 되어 서버 노드에서 실행되는 작업량 $W_{offload}$
  - $W_{offload}$(전체 오프로딩 작업량) = $W.Control_{Server}$(4.1.2절) + $W.Model_{Server}$(4.1.2절)
  - $W_{offload}$(부분 오프로딩 작업량) = $W.Control_{Server}^{Offloadable}$(4.2.2절) + $W.Model_{Server}^{Offloadable}$(4.2.2절)
- 모바일 노드와 서버 노드 간의 네트워크를 통해 전송되는 데이터량: $Size\ (InputData)$(4.1.2절)
- 전송되는 프로그램의 크기: $Size\ (OffPackage)$(5.1.2절)
- 모바일과 서버 간 네트워크 대역폭: $Bandwidth\ (Mobile, Server)$(4.1.2절)

각 6가지 요소에 대해 값이 큰 경우와 작은 경우로 변화를 주어, 전체 32개의 실험 프로그램을 작성하였다. 주의할 점은 $W_{Total} = W_{fixed} + W_{offload}$이기 때문에, W_{fixed}와 $W_{offload}$ 모두 큰 값이거나 모두 작은 값인 경우는 실험에 고려하지 않았다.

위와 같은 6가지 요소를 고려한 테스트 프로그램의 유형은 다음 (표 1)과 같다.

PGM_{HHLHHH}과 PGM_{HHLHLL}은 애플리케이션 전체 작업량이 높은 프로그램이 사용되며, 주어진 입력값을 통해 그래프를 생성한 후, 그래프에서 서로 인접한 정점들에게 다른 색을 가지도록 설정하는 Graph Coloring 프로그램을 사용한다. 다만 두 프로그램은 오프로딩을 위해 전송되는 프로그램의 크기와 네트워크 대역폭(High는 Wifi, Low는 3G)이 다르다.

PGM_{HLHLLH}과 PGM_{LHLHHL}는 서버 노드에서는 Matrix Multiplication 프로그램이 사용되며, 모바일 노드에서는 전달받은 결과 배열을 정렬하는 Sort 프로그램을 사용한다. Matrix Multiplication의 기능을 수행하기 위해 행의 값과 열의 값, 원소의 최대값을 모바일 노드로 부터 제공 받아 기능을 위한 입력값 으로 설정할 수 있다. PGM_{HLHLLH}은 모바일 노드의 높은 작업량을 위해 Bubble Sort가 사용되며, PGM_{LHLHHL}모바일 노드의 낮은 작업량을 위해 Quick Sort가 사용된다. 전송 데이터의 크기는 PGM_{HLHLLH}에서는 10x10의 배열을, PGM_{LHLHHL}에서는 1,000x1,000의 배열을 전송하고, 네트워크 대역폭을 달리 사용하였다.

PGM_{LLHLHH} 과 PGM_{LLHLLL}은 애플리케이션 전체 작업량이 낮은 프로그램을 선정하였으며, 행과 열을 곱하여 결과값을 가져오는 Matrix Multiplication 프로그램을 사용한다. 행의 값과 열의 값 그리고 원소의 최대값을 입력으로 설정할 수 있으며, 전송 데이터의 크기가 낮도록 설정 하기 위해서 계산된 배열 Median값만 가져온다. 다만 두 프

(표 1) 6가지 요소를 고려한 프로그램 유형
(Table 1) Types of Test Programs by Considering Six Elements

요소 / 프로그램 ID	W_{Total}	W_{fixed}	$W_{offload}$	Size(InputData)	Size(OffPackage)	Bandwidth (Mobile,Server)
PGM_{HHLHHH}	High	High	Low	High	High	High
PGM_{HHLHLL}	High	High	Low	High	Low	Low
…	…	…	…	…	…	…
PGM_{HLHLLH}	High	Low	High	Low	Low	High
PGM_{LHLHHL}	Low	High	Low	High	High	Low
…	…	…	…	…	…	…
PGM_{LLHLHH}	Low	Low	High	Low	High	High
PGM_{LLHLLL}	Low	Low	High	Low	Low	Low

로그램은 오프로딩을 위해 전송되는 프로그램의 크기와 네트워크 대역폭이 다르다.

7.2. 실험 결과 및 평가

앞 장에서 설정한 테스트 프로그램을 다음 방식으로 실행하였다.

- 모바일 디바이스에서 실행 (이 경우 Android로 프로그램을 작성) - 오프로딩 하지 않은 경우
- 네 가지 오프로딩을 적용한 경우 (이 경우 Android 프로그램과 Java 프로그램 작성)

(표 2)는 각 테스트 프로그램별 오프로딩을 수행하지 않은 경우, 모바일 노드의 문제 발생 시 오프로딩을 수행하지 않은 경우, 네 가지 오프로딩을 수행한 경우에 대한 실험 결과를 보여준다.

이 표에 나타난 실행시간을 그래프로 표현하면 (그림 11)과 같다.

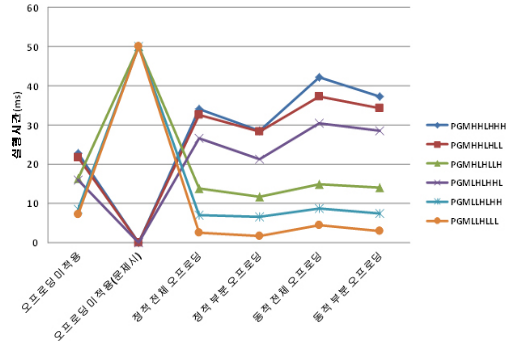
위의 그래프를 해석하면 다음과 같은 결과를 도출할 수 있다.

- 오프로딩을 적용했을 때와 적용하지 않았을 때의 성능 상의 차이점

  (그림 11)에서 보면 모바일 디바이스에 특정 문제가 발생했을 때에는 실행 속도가 현저히 떨어지거나, 결과값이 반환되지 않기 때문에, 오프로딩을 실행할 경우 성능 이점이 높음을 확인할 수 있다.

  정상 상태인 경우에는 대부분 오프로딩을 수행하는 경우가 성능이 더 떨어지지만 모바일 노드와 서버 노드간의 네트워크를 통해 전송되는 데이터량과 전송되는 프로그램의 크기가 작거나 애플리케이션의 복잡도(작업량)가 높은 경우에는 오프로딩으로 인한 성능 향상이 있음을 확인하였다.

  결론적으로 대부분의 경우 오프로딩으로 인한 이점이 향상되므로, 오프로딩으로 인한 오버헤드를 줄임으로써 그 이점을 극대화할 수 있을 것으로 기대한다.



(그림 11) 오프로딩 방식에 따른 실행시간 비교

(Figure 11) Comparing Response Time by Offloading Schemes

- 정적 오프로딩과 동적 오프로딩의 성능 상의 차이점

(표 3)은 정적 오프로딩과 동적 오프로딩의 실행시간의 차이를 보여준다.

동적 오프로딩을 실행하기 위해서는 프로그램을 전송해야 함으로써, 준비에 걸리는 시간을 포함하여야 한다. 그러므로 동적 오프로딩이 상대적으로 정적 오프로딩 실행 속도보다 오히려 저하될 수 있으며, 전송되는 프로그램의 크기에 따라서, 정적 오프로딩의 실행 속도가 (표 3)의 차이와 같이 빠르다는 것을 확인 할 수 있었다.

(표 2) 실험 결과 (실행시간, 단위: ms)

(Table 2) Experiment Result (Response Time in ms)

프로그램 ID	오프로딩 미적용	오프로딩 미적용 (문제 발생시)	정적-전체 오프로딩	정적-부분 오프로딩	동적-전체 오프로딩	동적-부분 오프로딩
PGM_{HHLHHH}	22.78	0(Error)	34.11	28.43	42.13	37.34
PGM_{HHLHLL}	21.61	0(Error)	32.53	28.34	37.34	34.27
…	…	…	…	…	…	…
PGM_{HLHLLH}	16.43	50(Excess)	13.86	11.65	14.84	13.92
PGM_{LHLHHL}	15.94	0(Error)	26.67	21.34	30.45	28.45
…	…	…	…	…	…	…
PGM_{LLHLHH}	8.34	50(Excess)	7.03	6.45	8.65	7.34
PGM_{LLHLLL}	7.21	50(Excess)	2.37	1.64	4.36	2.87

(표 3) 정적 오프로딩과 동적 오프로딩의 실행시간의 차이 결과 (실행시간, 단위: ms)
(Table 3) Differences in Response Time between Static Offloading and Dynamic Offloading (Unit: ms)

분류 요소	PGM_{HHLHHH}	PGM_{HHLHLL}	PGM_{HLHLLH}	PGM_{LHLHHL}	PGM_{LLHLHH}	PGM_{LLHLLL}
정적-전체과 동적-전체 차이	8.02	4.81	0.98	3.78	1.62	1.99
정적-부분과 동적-부분 차이	8.91	5.93	2.27	7.11	0.89	1.23

(표 4) 전체 오프로딩과 부분 오프로딩의 실행시간의 차이 결과 (실행시간, 단위: ms)
(Table 4) Differences in Response Time between Full Offloading and Partial Offloading (Unit: ms)

분류 요소	PGM_{HHLHHH}	PGM_{HHLHLL}	PGM_{HLHLLH}	PGM_{LHLHHL}	PGM_{LLHLHH}	PGM_{LLHLLL}
정적-부분과 정적-전체 차이	5.68	4.19	2.21	5.33	0.58	0.77
동적-부분과 동적-전체 차이	4.79	3.07	0.29	2	1.31	1.49

그러므로, 사전에 애플리케이션이 설치되어 있는 노드가 없는 경우에만 동적 전체 또는 동적 부분 오프로딩을 수행해야 하며, 그 외의 경우에는 정적 오프로딩을 적용해야 오프로딩으로 인한 성능을 극대화할 수 있다.

• 전체 오프로딩과 부분 오프로딩의 성능 상의 차이점 (W_{fixed}와 $W_{offload}$ 따른 오프로딩 성능 차이)

(표 4)는 전체 오프로딩과 부분 오프로딩의 실행시간의 차이를 보여준다.

모바일 노드와 서버 노드의 자원을 함께 사용함으로써, 부분 오프로딩이 상대적으로 전체 오프로딩 실행 속도보다 실행 속도 면에서 (표 4)의 차이와 같이 빠르다는 것을 확인할 수 있었다. 그러므로, 노드매니저는 해당 애플리케이션의 오프로딩 가능한 컴포넌트를 분석하여, 가급적이면 부분 오프로딩을 선택해야 한다.

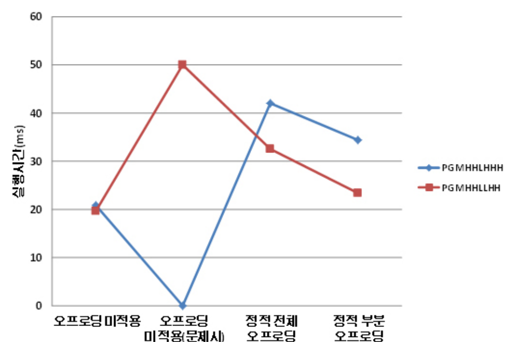
• *Size(InputData)*에 따른 오프로딩 성능 비교

(그림 12)는 *Size(InputData)*에 따른 오프로딩 실행시간의 차이를 보여준다.

그림에서 보면 문제 상황의 오프로딩 미적용시, PGM_{HHLHHH}은 특정 문제로 인해 오류가 발생하였고 PGM_{HHLLHH}은 50이상 소요되었기 때문에, 실행 시간 측정에서 제외하였다. 이 외에 두 프로그램의 정적 전체 오프로딩과 정적 부분 오프로딩의 실행시간을 비교한 결과, 두 노드 사이에 전송되는 데이터가 크면 클수록 오프로딩으로 인한 이점이 감소됨을 확인할 수 있었다.

• *Size(OffPackage)*에 따른 오프로딩 성능 비교

*Size(OffPackage)*에 따른 오프로딩 이점을 비교하기 위해 PGM_{HHLHHH}와 PGM_{HHLHLH}를 사용하였다. 이 두 프로그램은 동적 전체 오프로딩시에는 38.11과 31.53, 동적 부분 오프로딩 시에는 32.43과 23.34의 시간이 소요되었다. 결과적으로 패키지 크기가 클수록 오프로딩으로 인한 이점이 감소됨을 확인할 수 있었다.



(그림 12) *Size(InputData)*에 따른 오프로딩 실행시간 비교
(Figure 12) Comparing Response Time by Size (InputData)

• *Bandwidth(Mobile, Server)*에 따른 오프로딩 성능 비교

*Bandwidth(Mobile, Server)*에 따른 오프로딩 이점을 비교하기 위해, *Size(InputData)*가 큰 경우 PGM_{HHLHHH}, PGM_{HHLHHL} 2가지와 *Size(InputData)*가 작은 경우 PGM_{HHLLHH}, PGM_{HHLLHL} 2가지를 비교하였다. 그 결과 *Bandwidth (Mobile, Server)*가 높은 경우가 낮은 경우보다 그 속도가 약 3배정도 빠르다는 것을 확인할 수 있었다. 결과적으로 두 노드 사이의 네트워크 대역폭이 클수록 오프로딩으로 인한 이점이 증가됨을 확인할 수 있었다.

위의 결과로 다음의 결과를 도출할 수 있다.

- 사전에 동일한 애플리케이션이 여러 노드에 설치되어 있는 경우에는 정적 오프로딩을 선정하고, 그렇지 않은 경우에만 동적 오프로딩을 수행한다.
- 작업량이 많은 애플리케이션의 경우, 부분 오프로딩보다는 전체 오프로딩을 수행한다.
- 애플리케이션의 오프로딩 가능한 컴포넌트 비율에 따라 전체 또는 부분 오프로딩을 수행한다.
- 전송되는 데이터량이 적거나 전송되는 프로그램의 크기가 작을수록 오프로딩의 이점을 높으므로, 사전에 이를 고려하여 설계된 애플리케이션일수록 오프로딩 이점을 극대화할 수 있다.

8. 결　론

모바일 디바이스의 제한된 자원을 해결하는 MCC에서는 주로 기능 컴포넌트를 다른 노드로 오프로딩하는 접근법을 주로 사용한다. 현재까지 진행된 오프로딩 기법은 동적으로 변화가 심한 MCC 환경에서 현재 상황을 고려하지 않고 오프로딩을 실행하는것에 대한 한계점을 가지고 있다. 위의 한계점을 해결하기 위해, 본 논문에서는 MCC 환경에서 실용적으로 오프로딩을 설계하는 기법을 제안하였다.

먼저 MCC의 다양한 문제를 해결하기 위해, 다양한 상황에 적용 가능한 4가지 종류의 오프로딩을 소개하였다. 그리고, 정적 및 동적 오프로딩에 대한 구조 및 런타임 실행 절차, 오프로딩 비용 모델을 제안하였다. 오프로딩은 런타임에 현재 QoS 상황에 맞게 사용자에게 투명한 방식으로 오프로딩이 실행되도록 설계하며, 이에 대한 시간 성능 측면의 이점을 증명하기 위해 비용 모델을 제시하였다. 마지막으로, 비용 모델에서 고려된 오프로딩에 영향을 미치는 주요 요소를 기반으로 실험을 수행하여 오프로딩 별 성능 이점을 비교하였다.

제시된 기법은 구현이 가능하도록 실용적인 수준으로 설계되며, 비용 모델을 제시하여 오프로딩을 통한 성능향상이 있음을 정량적으로 증명하였다.

참고문헌(Reference)

[1] M. Fernando, S.W. Loke, and W. Rahayu, “Mobile Cloud Computing: A Survey,” *Future Generation Computer Systems*, Vol. 29, pp. 84-106, 2013.

[2] L. Guan, X. Ke, M. Song, and J. Song, “A Survey of Research on Mobile Cloud Computing,” *In Proceedings of 10th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2011)*, pp. 387-392, 2011.

[3] Han Qi and Abdullah Gani, “Research on Mobile Cloud Computing: Review, Trend and Perspectives,” *In Proceedings of the 2nd International Conference on Digital Information and Communication Technology and It’s Applications (DICTAP 2012)*, pp. 195-202, 2012.

[4] Longzhao Zhong, Beizhan Wang, Haifeng Wei, “Cloud Computing Applied in the Mobile Internet,” *In Proceedings of the 7th International Conference on Computer Science & Education (ICCSE 2012)*, pp. 218-221, 2012.

[5] Saeid Abolfazli, Zohreh Sanaei, Muhammad Shiraz, Abdullah Gani, “MOMMC: Market-Oriented Architecture for Mobile Cloud Computing Based on Service Oriented Architecture,” *In Proceedings of the 1st IEEE International Conference on Communications in China Workshops (ICCC 2012)*, pp. 8-13, 2012.

[6] E.Cuervo, A. Balasubramanian, D.K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “MAUI: Making Smarphones Last Longer with Code Offloaded,” *In Proceedings of 8th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2010)*, pp. 49-62, 2010.

[7] Kanad Sinha and Milind Kulkarni, “Techniques for Fine-Grained, Multi-Site Computation Offloading,” *In Proceedings of IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011)*, pp. 184-194, 2011.

[8] Byung-Gon Chun and Petros Maniatis, “Dynamically Partitioning Applications Between Weak Devices and Clouds,” *In Proceeding of 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond (MCS 2010)*, Articla No. 7, 2010.

[9] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik and Ashwin Patti, “CloneCloud: Elastic Execution between Mobile Device and Cloud,” *In Proceedings of 6th European Conference on Computer Systems (EuroSys 2011)*, pp. 301-314, 2011.

[10] Karthik Kumar, Yung-Hsiang Lu, “Cloud Computing for Mobile Users: Can Offloading Computation Save Energy,” *IEEE Computer*, Vol. 43, pp. 51-56, 2010.

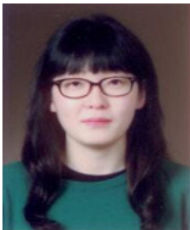
[11] Rich Wolski, Selim Gurun, Chandra Krintz, Dan Nurmi,“Using Bandwidth Data to Make Computation Offloading Decisions,” *In Proceedings of IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pp. 1-8, 2008.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

◐ 저 자 소 개 ◑

박 민 균 (Min Gyun Park)
2012년 경북대학교 소프트웨어공학과 (공학사)
2012년~현재 숭실대학교 컴퓨터학과 석사 과정
관심분야 : 클라우드 컴퓨팅(Cloud Computing), 모바일 서비스(Mobile Service)
e-mail: parkmingyun1@gmail.com

Piao Zhen Zhe
2011년 Changchun Normal University (공학사)
2012년~현재 숭실대학교 컴퓨터학과 석사 과정
관심분야 : 모바일 클라우드 컴퓨팅(Mobile Cloud Computing), 서비스 지향 아키텍처(SOA)
e-mail: zhenzhe826@gmail.com



라 현 정 (Hyun Jung La)
2003년 경희대학교 우주과학과 (이학사)
2006년 숭실대학교 컴퓨터학과 (공학석사)
2011년 숭실대학교 컴퓨터학과 (공학박사)
2011년~현재 숭실대학교 모바일 서비스 소프트웨어공학 센터 연구 교수
관심분야 : 서비스 지향 컴퓨팅 (Service Oriented Computing), 클라우드 컴퓨팅(Cloud Computing),
모바일 서비스(Mobile Service)
e-mail: hjla80@gmail.com

김 수 동 (Soo Dong Kim)
1984년 Northeast Missouri State University 전산학 (학사)
1988년/1991년 The University of Iowa 전산학 (석사/박사)
1991년~1993년 한국통신 연구개발단 선임연구원.
1994년~1995년 현대전자 소프트웨어연구소 책임연구원.
1995년 9월~현재 숭실대학교 컴퓨터학부 교수.
관심분야 : 서비스 지향 아키텍처(SOA), 클라우드 컴퓨팅(Cloud Computing),
모바일 서비스(Mobile Service), 객체지향 S/W공학, 컴포넌트 기반 개발 (CBD),
소프트웨어 아키텍처(Software Architecture)
e-mail: sdkim777@gmail.com