데이타 플로우 명세로부터 직렬화된 멀티태스킹 코드 생성

(Serialized Multitasking Code Generation from Dataflow Specification)

권 성 남 † 하 순 회 ††
(Seongnam Kwon) (Soonhoi Ha)

요 약 갈수록 복잡해지는 임베디드 시스템을 개발하는데 있어서 소프트웨어 개발의 중요성이 점차 커지고 있다. 대부분의 임베디드 응용 소프트웨어는 멀티 태스크로 구성되어 있는 병렬 소프트웨어이며, 기존의 순차적인 프로그래밍 언어만으로 개발하는 것 보다는 알고리즘의 병렬성을 명세하기에 용이한 데이타 플로우 모델로부터 소프트웨어를 생성하는 것이 유망하다. 생성된 멀티태스킹 코드를 수행하기 위해선 태스크들을 병렬적으로 수행해 주고 태스크간 동기화를 담당해 줄 운영체제의 도움이 필요하다. 그러나 운영체제를 사용하기 어려운 환경이나 설계 공간 탐색 과정에서 운영체제를 매번 다양한 하드웨어 플랫폼에 포팅하기 어려운 경우에는 운영체제 없이 멀티 태스크 응용을 수행할 수 있는 방법이 필요하다. 이것을 위해서 이 연구에서는 데이타 플로우 명세로부터 직렬화된 멀티태스킹 코드를 생성하는 방법을 제안한다. 제안하는 방법에서 하나의 태스크는 데이타 플로우 모델로 명세되며, 하나의 C 코드로 생성된다. 코드 생성은 크게 두 단계로 이루어지는데, 먼저 태스크를 구성하는 블록들을 각각 함수 형태로 코드를 생성한 후에, 생성된 여러 태스크의 함수들을 모아서 직렬화하여 호출하는 스케줄러를 만든다. 이 때에 스케줄러를 효율적으로 만들 수 있는 자료구조 및 정보를 제공하여 사용자가 수동으로 스케줄러를 만드는 것도 가능하도록 하였다. DivX예제를 통하여 제안하는 방법으로 생성한 코드가 효율적으로 올바르게 동작함을 보였다.

키워드 : MPSoC, 직렬화 컴파일러, 코드 생성, 데이타 플로우, 멀티태스킹

Abstract As embedded system becomes more complex, software development becomes more important in the entire design process. Most embedded applications consist of multi-tasks, that are executed in parallel. So, dataflow model that expresses concurrency naturally is preferred than sequential programming language to develop multitask software. For the execution of multitasking codes, operating system is essential to schedule multi-tasks and to deal with the communication between tasks. But, it is needed to execute multitasking code without OS when the target hardware platform cannot execute OS or target platforms are candidates of design space exploration, because it is very costly to port OS for all candidate platforms of DSE. For this reason, we propose the serialized multitasking code generation technique from dataflow specification. In the proposed technique, a task is specified with dataflow model, and generated as a C code. Code generation consists of two steps: First, a block in a task is generated as a separate function. Second, generated functions are scheduled by a multitasking scheduler that is also generated automatically. To make it easy to

· 본 연구는 BK21 프로젝트와 교육과학기술부 도약연구 지원사업(R17-2007-086-01001-0)의 지원을 받아 진행 되었다. 또한 서울대학교 컴퓨터신기술연구소와 IDEC은 본 연구에 필요한 기자재들을 지원해 주었다. 본 연구는 한국전자통신연구원의 SoC 핵심설계인력양성사업의 부분적인 지원을 받았다.

† 학생회원 : 서울대학교 전기 컴퓨터공학부
ksn@iris.snu.ac.kr
†† 정 회 원 : 서울대학교 전기 컴퓨터공학부 교수
sha@iris.snu.ac.kr

논문접수 : 2008년 3월 12일
심사완료 : 2008년 6월 18일

정보과학회논문지: 시스템 및 이론 제35권 제9호(2008.10)

write customized scheduler manually, the data structure and information of each task are defined. With the preliminary experiment of DivX player, it is confirmed that the generated code from the proposed framework is efficiently and correctly executed on the target system.

Key words : MPSoC, Serializing Compiler, Code Generation, Dataflow, Multitasking

1. 서 론

다양한 사용자의 요구를 만족시키기 위해 시스템은 갈수록 복잡해지고 있으며, 개발주기 또한 짧아지고 있다. 이러한 상황을 타개하고자 다양한 시스템 설계 방법론이 제안되었다. 기존에 설계 시간의 대부분을 차지하던 하드웨어 설계는 플랫폼 기반 설계 방법에 기반한 하드웨어 플랫폼의 재사용을 통해 개발시간을 상당 부분 단축할 수 있었다. 그 결과 최근에는 시스템의 개발 시간을 주도하는 부분이 하드웨어 설계에서 소프트웨어 설계로 바뀌고 있다. 더욱이 다양한 사용자 요구를 충족시키기 위해 소프트웨어는 기본적으로 병렬적으로 동작하는 멀티 태스크 프로그램이 되었으며 이는 소프트웨어 개발을 더욱 어렵게 하고 있다.

일반적으로 소프트웨어를 개발하는 데는 C언어를 비롯한 순차적으로 수행되는 프로그래밍 언어를 많이 사용하고 있다. 그러나 순차적 명세에 초점을 둔 기존 프로그래밍 언어만으로 병렬적인 응용을 개발하는 데는 어려움이 따른다. 반면에 주로 신호처리 알고리즘을 명세하기 위해 사용되었던 데이타 플로우 모델과 같은 계산 모델(MoC: Model of Computation)은 병렬성을 자연스럽게 표현할 수 있기 때문에 병렬 알고리즘 명세에 보다 적합하다. 특히 데이타 플로우 모델을 이용한 명세에서 블록간 수행 순서는 블록간에 주고 받는 데이타 의존관계가 있는 경우에만 정해지게 되며 서로 무관한 블록들은 병렬적으로 수행이 가능하다. 이러한 특성 때문에 데이타 플로우 모델로 명세한 알고리즘은 내부의 병렬성이 분명히 드러나게 되며, 알고리즘의 병렬성 명세에 있어서 큰 장점을 보인다. COSSAP[1], SPW[2], Ptolemy[3] 등은 데이타 플로우 명세로부터 코드 생성을 지원하는 대표적인 도구들로서 디지털 신호 처리 예제를 명세 하는 것에 특화되어 있으며, 그래픽 사용자 환경을 통하여 명세한 데이타 플로우 모델로부터 코드를 자동 생성할 수 있는 기능을 제공한다.

데이타 플로우 명세로부터 생성한 소프트웨어 코드가 멀티태스킹 응용이라면, 각 태스크 코드가 병렬적으로 수행이 되어야 하며, 그것을 위해 일반적으로 멀티 태스크 스케줄링을 지원해주는 운영체제가 필요하다. 만약 생성된 코드가 잘 알려진 운영체제 위에서 수행될 것을 가정한다면, 그 운영체제를 가정하고 코드를 생성하면 큰 문제 없이 동작시키는 것이 가능하다. 그러나 타겟 시스템에서 특정 운영체제를 가정하기 어렵거나 운영체제 자체를 사용하기 어렵다면 생성한 멀티 태스킹 소프트웨어를 구동하는 것에 문제가 발생한다.

이 문제는 시스템 수준 설계 방법론에서 큰 연구 주제인 설계 공간 탐색(Design space exploration) 과정에서는 더욱 문제가 된다. 설계 공간 탐색 과정에서는 최적의 플랫폼을 찾기 위해 플랫폼의 후보(프로세서의 종류 및 개수, 메모리 구조 및 크기, 통신 구조, 하드웨어 가속기, 운영체제 등)를 수시로 변경하여야 하며, 해당 후보에서 소프트웨어의 성능 및 각종 분석 정보를 얻기 위해 개발하려는 소프트웨어를 수행시켜야 할 필요가 있다. 이 때 각 플랫폼 후보에 대해서 매번 운영체제를 포팅하는 것은 설계 시간 측면에서 불가능에 가깝다. 따라서 설계 생산성 측면에서 생성된 멀티 태스킹 소프트웨어를 운영체제의 도움 없이 수행할 수 있는 방법이 필요하다.

따라서 이 논문에서는 데이타 플로우 명세로부터 운영체제의 도움 없이 수행할 수 있도록 직렬화된 멀티 태스킹 코드를 생성하는 방법을 제안한다. 이 방법은 크게 다음과 같은 장점을 갖는다.

1. 타겟 플랫폼에 운영체제의 포팅 없이 소프트웨어를 수행할 수 있어 개발 효율을 높일 수 있다.
2. 운영체제가 존재하는 경우 운영체제 위에서 코드를 수행할 수도 있으며, 이 경우 멀티 태스크 스케줄링을 위한 운영체제의 스케줄러와 태스크간 통신에 따른 동기화에 의한 실행 시간 부담을 줄일 수 있다.
3. 직렬화되어 생성된 코드는 기존 방법으로 생성된 코드와 동일하게 디버깅 정보를 포함한 데이타 플로우 모델의 상위수준 명세 정보가 유지되어, 추후 분석 및 디버깅에 용이하다.

본 논문의 나머지 부분은 다음과 같이 구성되어 있다. 먼저 2장에서는 제안하는 설계 환경의 설계 흐름도를 살펴본 뒤, 3장에서는 이 연구의 동기를 부여한 예제를 설명하겠다. 또한 4장에서는 관련 연구를 살펴본 뒤 효율적인 설계 공간 탐색을 위해 보완되어야 할 부분들을 설명한다. 5장에서는 제안하는 직렬화된 멀티 태스킹 코드 생성 방법을 설명하고 6장에서는 직렬화된 코드의 성능을 향상시키기 위한 블록 클러스터링 방법을 설명한다. 7장에선 실험을 통해 직렬화된 코드의 실행 부담을 살펴보겠으며, 8장에서는 제안 하는 방법을 여러 측면에서 평가한다. 최종적으로 9장에서 논문을 결론지으

며 추가 연구방향에 대해 설명하겠다.

2. 제안하는 설계 환경

그림 1은 제안하는 소프트웨어 설계 흐름도를 개괄적으로 보여준다[4]. 먼저 시스템 명세단계에서는 시스템의 동작과 아키텍쳐를 분리하여 기술하는 설계 방법론[5]에 기반하여 알고리즘과 아키텍쳐를 별도로 명세한다. 이 때 알고리즘은 그림 2와 같이 크게 두 가지 모델을 사용하여 명세 한다. 가장 상위에서는 태스크 모델[6]을 사용하여 태스크들을 명세 하는데 이 모델에서 각각의 블록은 하나의 태스크를 나타낸다. 각 태스크는 병렬적으로 동작하며 포트를 통해 채널에 접근하여 다른 태스크와 데이타를 주고받을 수 있다. 태스크는 각각 설정된 수행 조건에 따라 깨어나서 자신의 일을 수행하며 수행 조건으로는 정해진 시간 주기나 태스크에 도착하는 이벤트 등이 될 수 있다.

하나의 태스크 내부의 동작은 데이타 플로우 모델의 일종인 SDF(Synchronous Dataflow[7])를 이용해 명세한다. 제안하는 개발환경에서는 데이타 플로우 모델의 블록을 크게 통신 블록과 계산 블록으로 분류하였다. 통신 블록은 다른 태스크와 데이타를 주고 받는 기능을 수행하며 계산 블록은 입력 데이타를 받아 어떤 기능을 수행한 뒤 출력 데이타를 생성하는 기능을 한다. 그림 2에 예시된 H.263 디코더 태스크의 경우, 다른 태스크로부터 통신 블록을 통해 데이타를 받은 뒤 계산을 거쳐 화면으로 출력한다.

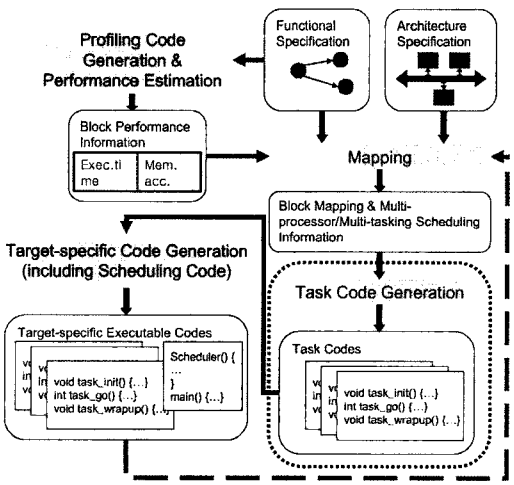

그림 1 제안하는 소프트웨어 설계 흐름도

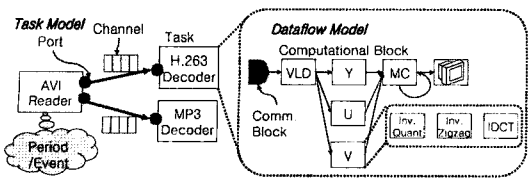

그림 2 알고리즘 명세의 예 (DivX 동영상 플레이어)

알고리즘을 명세한 후에는 알고리즘 명세의 각 블록을 아키텍쳐 후보에 매핑하게 된다. 각 데이타 플로우 블록은 매핑 과정 및 성능 평가에 사용될 수 있는 성능 정보를 갖고 있으며 이 성능 정보는 알고리즘 명세로부터 블록의 성능 분석용 코드를 생성하여 시뮬레이션을 통해 얻는다. 매핑과 블록의 성능 정보를 얻는 것과 관련된 자세한 내용은 각각 [8]과 [9]에 설명되어 있다.

매핑이 완료되면 매핑 결과를 가지고 각 태스크의 코드를 생성하게 된다. 하나의 태스크는 하나의 C코드 파일로 생성이 되는데, 태스크 내부의 블록이 복수의 프로세싱 유닛에 매핑 된 경우 하나의 태스크가 복수의 태스크로 분할되어 코드가 생성될 수 있다. 그 뒤에 태스크들을 묶어 스케쥴링하는 코드를 포함한 타겟 아키텍쳐 의존적인 코드를 생성한다. 최종적으로 타겟 의존적인 코드와 태스크 코드를 묶어 특정 타겟에서 수행할 수 있게 된다. 이 때 최종 코드가 실제 타겟에서 원하는 성능을 만족시키지 못하는 경우 성능 파라메터를 바꿔 매핑을 다시 수행하게 된다.

주어진 응용에 대한 최적의 시스템을 찾기 위해선 매핑 결과를 변화시키며 생성된 다양한 코드를 다양한 아키텍쳐에서 수행하는 것이 필연적이다. 이러한 설계 공간 탐색 과정에서, 설계 효율을 높이기 위해선 운영체제 없이 멀티 태스킹 소프트웨어를 수행할 수 있는 방법이 필요하다.

3. 연구의 동기

이 연구의 동기를 설명하기 위해선 먼저 기존의 데이타 플로우 명세로부터 코드를 생성하는 방법에 대해 설명할 필요가 있다. 데이타 플로우 모델에서 하나의 노드 혹은 블록은 입력 데이타 스트림을 변환하여 출력 스트림을 만들어내는 기능 블록을 의미한다. 각 단위 블록의 기능명세는 C 혹은 VHDL과 같은 상위수준 언어로 이루어진다. 블록들을 연결하는 간선(arc)은 원시 노드로부터 목적지 노드에 데이타 샘플의 스트림을 전달하는 채널을 나타낸다. 데이타 플로우 모델에서 블록이 수행되기 위해 필요한 입력 샘플의 개수와 실행 후 발생하는 출력 샘플의 개수를 각각 입력 혹은 출력 ***샘플 레이트***(input/output sample rate)라고 하며 샘플 레이트가 고정된 데이타 플로우를 특별히 SDF[7]라고 한다. 그림 3(a)의 Task2는 SDF로 명세 된 태스크의 예를 보여주는데, 각 간선은 블록이 수행될 때 소모되고 생성되는 샘플의 개수가 표시되어 있다. Task2는 내부 명세를 위

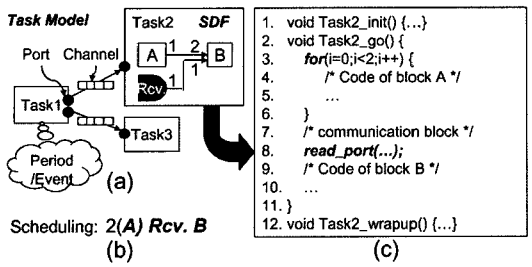

그림 3 SDF모델에서 태스크 코드 생성의 예: (a) 알고리즘 명세의 예, (b) 태스크 2의 스케줄링 예, (c) 스케줄링을 고려한 태스크 2의 자동 생성된 코드

해 계산 블록 A와 B, 그리고 데이타를 다른 태스크로부터 읽어오는 Rcv. 통신 블록이 사용되었다.

SDF의 특징 때문에 블록의 수행 순서나 필요한 메모리 요구량 등을 컴파일 시간에 정적으로 결정할 수 있다. 특히 블록의 수행 순서를 ***스케줄링***이라고 하며, 코드 생성을 위해 필요하다. 어떤 스케줄에서 한 노드가 호출되는 회수를 그 노드의 ***반복회수***(repetition count)라고 한다. 그림 3(b)는 Task2 명세의 가능한 여러 가지 스케줄 가운데 하나를 보여주고 있으며, 여기서 2(A)의 의미는 블록 A가 2번 수행되는 것을 의미한다.

하나의 SDF 명세는 하나의 태스크로서 C파일로 생성이 되며 각각 ***{task name}_init(), {task name}_go(), {task name}_wrapup()***의 3개 함수로 구성이 된다. {task name}_init()은 태스크의 초기화를 담당하며, {task name}_go()는 태스크의 메인 코드가 들어가고 {task name}_wrapup()은 태스크를 마무리 하는 코드가 들어가게 된다. 그림 3(c)는 (b)의 스케줄링 결과에 따른 태스크 2의 코드 생성 결과를 보여준다. 스케줄링에 따라 블록의 메인 코드가 {task_name}_go()안에 생성이 되는데 블록 B가 2번 수행되어야 하기 때문에 "***for***" 문으로 블록 B의 코드가 감싸져 2번 수행되는 형태로 코드가 생성된다. 생성된 코드에서 8번째 줄의 "***read_port()***"함수는 태스크간 통신을 위해 사용되는 통신 API로, 채널에서 데이타를 읽어오는데 사용되는 함수이다. 블록의 코드가 "go"함수 안에 생성될 때 블록간 통신을 위해 필요한 버퍼 변수들 또한 적절하게 생성된다.

이렇게 생성된 태스크 코드들은 각기 병렬적으로 동작하며 실행주기가 되거나 이벤트가 도착하면 깨어나 수행이 된다. 만약 수행도중 다른 태스크와의 통신 문제로 블록킹이 발생하면 그 태스크는 다른 태스크에 의해 통신 문제가 해결될 때까지 블록이 된다. 이러한 모든 멀티 태스크 스케줄링과 태스크간 통신은 운영체제의 스케줄러에 의존하여 처리가 된다. 예를 들면 그림 3의 Task2가 블록 A의 코드를 수행한 뒤 Task1으로부터의 데이타를 읽어야 하는데 데이타가 아직 도착하지 못해서 읽을 수가 없다면 이 태스크는 블록이 되며 Task1이 병렬적으로 수행하여 충분한 데이타가 생성되었을 때 Task2는 다시 깨어나 그곳부터 수행이 재개된다.

이러한 상황에서 운영체제가 없다면 Task2를 위한 데이타가 존재하지 않을 때 더 이상 수행이 불가능해진다. Task2의 수행을 진행시키기 위해선 Task1이 수행되어야 하는데 Task2의 수행을 마무리하여야만 Task1을 실행할 수 있다. 만약 통신 블록이 발생하였을 때 Task2의 수행을 포기하고 Task1을 실행하려 한다면, 다음 번 Task2를 수행할 때는 블록이 일어났던 태스크 코드의 중간부분(그림 3(c)의 8번째 줄)부터 수행이 되어야 한다. 따라서 운영체제가 없다면 이 경우 수행이 어려워지며 새로운 형태로 코드를 생성해야 한다.

4. 관련연구

멀티태스킹 응용을 효율적으로 수행하기 위한 접근방법의 하나는 운영체제를 아키텍처 응용에 맞게 특화 시키는 것이다. [10]에서는 SpecC라는 시스템 수준 명세 언어에 RTOS동작 특성을 명세할 수 있는 기능을 추가하였으며, 설계 초기 단계에서부터 RTOS의 동작을 고려하여 설계 할 수 있도록 하였다. 이렇게 명세 된 RTOS의 특징은 함께 명세 된 시스템과 함께 최종 구현 단계까지 점차 구체화(refine)된다. [11]에서는 타겟에 특화된 RTOS를 수작업으로 만드는 것이 아닌, 상위 수준 시스템 명세로부터 타겟에 특화된 RTOS를 생성해주는 방법을 제안하였다. RTOS생성을 위해서 OS가 제공해주는 각종 기능에 대한 OS 라이브러리를 구축하고 아키텍처에 맞게 라이브러리를 조합하여 최종적인 RTOS를 생성해낸다. 이 방법을 사용하여 설계 환경을 구축하면 다양한 멀티 태스킹 응용을 타겟에서 효율적으로 수행할 수 있을 뿐 아니라, 설계 생산성 또한 향상시킬 수 있다. 그러나 RTOS생성을 자동화하기 위해선 타겟 아키텍처에 대한 OS 라이브러리의 구축이 필수적이며 또 다른 아키텍처에 대해선 새로운 OS 라이브러리를 구축해야 한다.

또 다른 접근 방법은 운영체제의 수행 시간 부담을 줄이기 위해 운영체제 없이 소프트웨어를 구동할 수 있도록 멀티 태스킹 코드를 직렬화시켜 단일 태스크 코드로 변환하는 것이다[12]. 직렬화 컴파일러로 불리는 이 연구에서는 C/C++로 기술된 멀티태스킹 코드를 입력으로 받아서 직렬화 된 단일 태스크의 코드로 변환한다. RTOS 자체를 자동 생성하는 연구와는 다르게 기존 C/C++ 컴파일러를 사용해 컴파일 가능한 코드를 생성하기 때문에 하드웨어 아키텍처가 바뀐다 하더라도 큰

어려움 없이 멀티태스킹 코드를 새로운 아키텍쳐에서 동작시키는 것이 가능하다. 또한 [13]에서는 직렬화된 코드의 수행시간 부담을 줄이기 위해 코드를 재구성하는 연구를 수행하였다.

직렬화 컴파일러의 기본 아이디어는 순차적으로 동작하는 하나의 태스크 코드 내부를 여러 조각으로 자르는 것이다. 잘린 코드 조각들은 생성된 멀티 태스크 스케줄러에 의해 호출되어 수행되는 스케줄의 단위가 된다. 즉, 코드 조각 내부를 수행하는 도중에는 다른 태스크로의 문맥 전환이 발생하지 않으며 코드 조각이 완전히 수행된 뒤에 다른 태스크의 코드 조각을 수행할 수 있게 된다.

직렬화 컴파일러에서 가장 중요한 것 중 하나가 태스크의 코드 조각 크기이다. 코드 조각에서 스케줄링을 하기 위해선 코드 조각 전후에 태스크 내부에서 사용되는 변수의 값을 저장/복원 하는 과정이 필요하다. 또한 스케줄러를 호출하여 다음 스케줄 할 태스크의 코드 조각을 찾는 부담이 존재한다. 따라서 코드 조각을 작게 하여 스케줄을 자주 할 경우 멀티 태스크 스케줄링은 보다 원활하게 수행하는 것이 가능하여 시간 제약 조건을 만족시키기 보다 용이해지지만 추가 되는 부담은 커지게 된다. [14]에서는 이 코드 분할 문제의 중요성을 언급하고 몇 가지 기법을 소개하였다. 성능의 문제를 떠나 [14]에서는 직렬화된 코드가 올바르게 동작하기 위해선 각 태스크의 동기화 지점(Synchronization Point)에서는 반드시 코드가 분할되어야 함을 언급하고 있다.

이 논문에서 제안하는 방법은 멀티 태스킹 코드를 OS없이 수행하기 위해 기본적으로 직렬화 컴파일러 기법에서 제안한 멀티 태스크 코드의 직렬화 기법을 이용하고 있다. 그러나 기존 직렬화 기법과 달리 코드를 분할 할 때 블록의 경계에서 분할하여 생성을 한다. 기존의 직렬화 컴파일러 기법은 컴파일러 기법을 이용하여 코드 자체를 분석하여 코드를 분할하며, 따라서 코드가 지니는 상위 수준 정보 및 의미를 고려하여 분할하는 것은 아니다. 그 결과 직렬화된 코드의 형태가 유지되지 않으며, 프로그래머가 이해할 수 있는 상위수준 정보 또한 사라지게 된다. 이는 제안하는 설계 환경과 같이 상위 수준 명세 정보를 이용하여 분석 및 최적화를 수행하는 경우 문제가 된다. 이것을 해결하기 위하여 이 연구에서는 생성된 코드를 분석하는 것이 아닌 상위수준 정보를 유지한 채 블록 단위로 직렬화 시키는 방법을 제안한다.

5. 직렬화된 멀티태스킹 코드 생성

직렬화된 멀티태스킹 코드를 생성하는 것은 크게 두 가지 단계를 거친다. 하나는 태스크 코드 자체를 직렬화 할 수 있는 형태로 생성하는 것이고 다른 하나는 그 태스크 코드를 잘 엮어 효율적으로 스케줄링 하는 코드를 생성하는 것이다. 일반적인 직렬화 컴파일러 관련 연구에서는 컴파일러 내부에서 이 두 단계가 모두 이루어지는 반면, 이 연구에서는 프로그래머가 타겟 아키텍쳐나 응용에 특화된 스케줄러를 설계, 수정하기 용이하도록, 스케줄링 코드를 생성하는 단계를 태스크 코드를 생성하는 것과 분리하였다.

5.1 직렬화 가능한 태스크 코드 생성

[14]에서 언급한 것과 같이 운영체제의 도움 없이 직렬화 하여 멀티 태스킹 응용을 동작시키기 위해선 태스크 코드가 작은 코드 조각으로 분할되어야 한다. 제안하는 설계 환경에서 태스크 코드는 어떤 기능 블록들로 구성된 데이타 플로우 명세로부터 생성이 된다. 따라서 직관적으로 기능 블록을 분할 된 태스크의 코드 조각으로 삼는 것을 먼저 생각해 볼 수 있다. 태스크간 통신은 특정 통신 블록을 통해 이루어지기 때문에 기능 블록 단위로 코드를 분할하는 것만으로도 멀티 태스크 응용을 직렬화하는 것이 가능하다.

그림 4는 직렬화 가능하게 생성된 태스크 코드의 예를 보여준다. init과 wrapup함수는 기존의 코드와 유사하게 생성되며 go함수만 생성된 형태가 달라진다. 이 코드의 기본 아이디어는 직렬화 컴파일러와 유사하게 태스크 코드를 스케줄링 할 수 있는 여러 개의 코드 조각으로 자르는 것이다. 그러나 그 코드 조각의 크기를 데이타 플로우 블록으로 고정하여 함수 형태로 생성한 것에 방법적인 차이가 있다. 1-3번 줄을 살펴보면 태스크 2에 사용된 블록 Rcv, A, B가 각각 함수형태로 생성된 것을 볼 수 있다.

```
1.  int Task2_Rcv() {...}
2.  int Task2_A() {...}
3.  int Task2_B() {...}

4.  int (*Task2_func[])() = { Task2_Rcv, ...};
5.  static schedInfo *currSchedInfo = NULL;

6.  int Task2_go() {
7.     int (**func_list)() = Task2_func;

8.     while(1) {
9.        if(func_list[currSchedInfo->blockId]()==-1)
10.          return -2;     // sync. blocking
11.       currSchedInfo = nextBlock(currSchedInfo);
12.       if (currSchedInfo == NULL) break;
13.       if (checkScheduler()==1) return -1;
14.    }

15.    // initialize port variables for next execution
16.    ...
17.    return 0;
18. }
```

그림 4 직렬화 가능하게 생성된 태스크 코드의 예

블록을 함수 형태로 생성하기 위한 추가적인 작업은 단지 블록간 통신에 사용되는 포트 변수를 전역 변수로 선언하는 것이다. 데이타 플로우 블록은 그 자체로 완전하며 단지 데이타를 입력 포트를 통해 읽은 뒤 자신의 기능을 수행하고 그 결과를 출력 포트를 통해 전달한다. 따라서 어떤 블록이 수행된 뒤에는 블록의 내부 상태를 별도로 저장할 필요 없이 출력 결과만을 보존하면 데이타 플로우 모델로부터 생성한 태스크 코드는 동작하게 된다. 이러한 특징 때문에 기존의 직렬화 컴파일러 기법과는 달리 코드 분할 지점에서 포트 변수를 제외한 블록의 모든 상태가 유지될 필요가 없어 별다른 Live 변수의 처리가 필요 없으며, 포트 변수만을 전역 변수로 생성하면 된다.

생성된 블록 함수들은 4번 줄과 같이 함수에 대한 포인터의 배열로 정리가 되어 go함수 내부에서 SDF 스케줄에 따라 호출이 된다. 블록의 SDF 스케줄은 그림 5(a) 에서 보여주는 것과 같은 연결 리스트(linked list) 자료구조인 ***schedInfo***를 이용하여 저장된다. 스케줄 내부의 루프 구조는 계층적인 방법을 통해 표현되며, ***schedInfo***의 ***blockID***가 -1인 경우 반복 회수만큼 수행되는 ***child*** 서브 리스트를 갖는다. 그림 5(b)는 schedInfo를 이용하여 저장된 SDF스케줄의 예를 보여주는데 2(A)는 ***child*** 서브 리스트를 이용하여 계층적으로 표현이 되어 있다.

그림 4의 생성된 ***Task2_go()*** 함수 내부에서 ***curr-SchedInfo***는 SDF 스케줄에 따라 현재 실행되어야 하는 블록을 가르키고 있으며, ***nextBlock()***함수에 의해 다음 블록으로 이동하게 된다. 스케줄 내 블록의 수행을 전부 마치게 되면 ***nextBlock()***은 ***NULL***을 반환하며, go함수의 1회 수행을 마치게 된다.

계산 블록은 일단 수행되면 항상 결과를 내고 수행을 종료하는 반면에 통신 블록은 블록이 발생할 수 있다. 통신 블록의 블록 여부를 구분하기 위해 블록의 수행 결과를 성공(0)과 실패(-1)의 두 가지 경우로 구분하였으며, 블록 함수가 실행되었을 때 반환 값을 체크하도록 하였다(9번 줄). 블록이 수행된 뒤에는 ***checkScheduler()*** 함수를 호출하여 이 태스크의 수행을 다른 태스크에게 양보해야 하는지 체크한다(13번 줄). 이 함수가 1을 반환하면 태스크 스케줄러 코드는 다음에 수행할 태스크를 선택하여 수행시킨다.

{task name}_go()함수는 태스크의 수행 결과에 따라 3 종류의 값을 반환한다: 태스크간 통신으로 인해 블록됨(-2), 다른 태스크에 의해 선점됨(-1), 그리고 수행이 종료됨(0). 태스크 스케줄링 코드는 이 반환 값에 따라 다음 수행할 태스크를 선택한다.

5.2 스케줄링 코드 생성

멀티 태스크를 효율적으로 스케줄링 하는 연구는 많은 실시간 관련 그룹에서 오랜 기간 해온 연구로 이 연구의 범위를 벗어난다. 그러나 스케줄링 코드를 작성하거나 생성하는데 필요한 정보를 그림 6과 같은 자료구조에 정리하여 생성하도록 하였다. 이 자료구조를 기반으로 작성된 스케줄링 코드는 제안하는 개발환경에서 라이브러리 형태로 정리되어 생성된 태스크 코드와 함께 묶여 동작하게 된다. 이 서브 섹션에서는 이 자료구조를 이용하여 어떻게 스케줄링 코드를 작성할 수 있는지 몇 가지 구현 방법을 설명하겠다.

그림 6은 스케줄러 코드에서 태스크의 정보를 접근하는데 이용되는 ***taskInfo*** 자료구조의 정의를 보여준다. 이 자료구조는 각 태스크의 go 함수(go), 태스크 수행 주기(period), 예상 수행 시간(taskTime), 태스크의 상태(status), 태스크의 남은 수행시간(remainedTime), 태스크 수행 회수(runCount)의 6개 변수로 구성되어 있다. 이 가운데 앞의 3개는 태스크 코드 생성시에 자동으로 값이 할당되며, 다음 3개는 생성된 코드가 동작하며 동적으로 변하게 된다.

그림 7은 이 자료구조를 이용하여 작성된 스케줄링 코드의 동작을 보여주는데, 스케줄링 정책에 따라 숫자가 표기된 ***①checkScheduler()*** 함수의 내부와 ②스케줄 할 태스크를 선택하는 코드(스케줄러 코드를 변경해줘야 한다.

5.2.1 비선점형(Non-preemptive) 스케줄러

비선점형 스케줄러에서는 어떤 태스크가 수행 중일 때 스케줄러에 보다 높은 우선순위의 태스크가 들어온다고 해서 기존 태스크가 중단되지 않는다. 태스크가 선

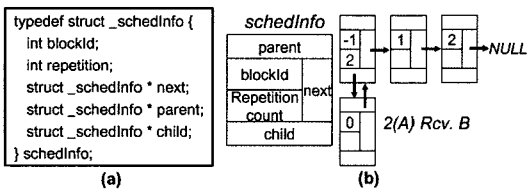

그림 5 (a) schedInfo자료구조의 정의와 (b) schedInfo를 이용하여 저장된 SDF 스케줄의 예

```
1. typedef struct {
2.    int (*go)();
3.    unsigned int period;
4.    unsigned int taskTime;
5.    int status;
6.    unsigned int remainedTime;
7.    unsigned int runCount;
8. } taskInfo;
...
9. taskInfo tInfo[TASK_NUM] = {...};
```

그림 6 태스크 정보를 위한 자료구조

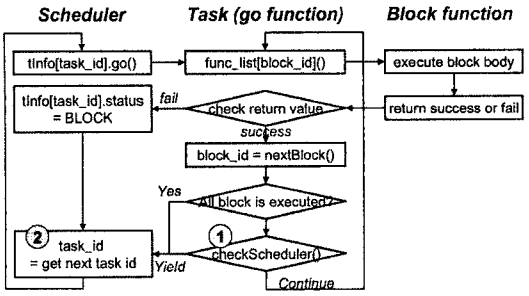

그림 7 생성된 코드의 동작 구조

```
1. for (i=0; i<runCount;i++) {
2.   task_id = 0;  // task id initialize
3.   while(1) {
4.     int all_task_done = 1;
5.     if (tInfo[task_id].runCount =< i) {
6.       status = tInfo[task_id].go();
7.       // if complete, runCount increase
8.       if (status == 0)
9.         tInfo[task_id].runCount+=tInfo[task_id].period;
10.    }
11.    // Run count check for all tasks
12.    for (j=0;j<TASK_NUM;j++) {
13.      if(tInfo[j].runCount=<i) {
14.        all_task_done = 0;
15.        break;
16.      }
17.    }
18.    if (all_task_done) break;
19.    // Round robin task selection
20.    task_id++;
21.    if (task_id==TASK_NUM)
22.      task_id=0;
23.  }
24.}
```

그림 8 순환 순서형 스케줄러의 태스크 선택 코드의 예

점되는 것을 막기 위해 ① checkScheduler()함수는 항상 0을 리턴하도록 하고 ② 태스크 선택 코드에는 태스크의 우선순위에 따라 선택하도록 하면 된다. 태스크의 우선 순위를 모두 같게 설정한다면 순환순서로 태스크가 선택 될 것이다.

5.2.2 순환순서형(Round-Robin) 스케줄러

순환순서형 스케줄러는 태스크들을 한 차례씩 반복하여 수행하는 스케줄러이며, 수행 도중 태스크에 할당된 시간이 초과되면 다른 태스크에 의해 선점이 발생된다. 이것을 위해서 ① ***checkScheduler()*** 함수 내부에 태스크가 할당된 시간을 초과 했으면 1을 리턴하도록 설정한다. 이 논문의 실험에서는 생성된 코드의 문맥 전환에 의해 발생할 수 있는 가장 큰 실행 부담을 측정하기 위해, 태스크 문맥 전환의 태스크 별 시간 할당을 모든 블록의 수행시간보다 작게 설정하였으며, 이 경우 블록 함수가 수행된 뒤에 무조건 문맥전환이 발생하게 된다.

② 태스크 선택 코드내부에는 전체 태스크의 한 주기 동안 수행되어야 하는 태스크 별로 정해진 수행 회수만큼 태스크가 수행되도록 설정한다. 이것을 위해 각 태스크의 실행 주기를 설정하며, 한 태스크의 실행 주기가 다른 태스크보다 더 짧다면 더 자주 실행이 된다. 예를 들면, 태스크 1이 태스크 2보다 3배 자주 수행되어야 한다면 태스크 1과 2의 실행주기를 각각 1과 3으로 설정해야 한다. 이 경우 각 태스크의 실행 주기는 실시간 스케줄링의 실행 주기와 달리 모든 태스크와의 상대적인 실행 주기를 의미하게 된다.

그림 8은 순환 순서형 스케줄러의 태스크 선택 코드의 예를 보여준다. 이 코드에서 스케줄러는 ***runCount*** 변수에 저장된 수행 회수만큼 반복해서 태스크를 수행하며 그것을 위해 전체 시간을 ***i*** 라는 변수에 저장하여 ***runCount***와 비교한다. 각 실행 때마다, 스케줄러는 순환순서로 각 태스크를 조사한다(2번, 20-22번 줄). 각 태스크의 ***runCount*** 가 전체 시간 i보다 작거나 같다면 그 태스크를 수행하게 되며(5-10번 줄), 그 태스크의 수행이 완전히 종료되면 그 태스크의 ***runCount*** 를 ***period*** 만큼 증가시킨다(8-9번 줄).

5.2.3 실시간 스케줄러

실시간 스케줄러를 구현하기 위해선 태스크의 시간 관련 정보가 필요하다. 필요한 태스크의 시간 정보로는 먼저 태스크의 수행 주기와 태스크가 소모한 시간, 또한 태스크가 수행을 완료하는데 남은 시간 등이 될 수 있다. ① ***checkScheduler()*** 함수에서는 시간 정보를 이용하여 각 블록이 수행한 뒤 그 태스크가 계속 수행을 할지를 결정하는 코드를 작성한다. ② 태스크 선택 코드 내부에는 스케줄링 정책에 맞게 다음 수행할 태스크를 선택하는 코드가 들어가야 한다.

만약 타겟 플랫폼에 시간 관련 라이브러리가 준비되어 있다면 그것을 이용하여 실시간 스케줄러를 구성할 수 있다. 그러나 운영체제의 도움 없이 시간관련 라이브러리를 이용하기는 쉽지 않으며 그러한 경우 대략적인 시간을 예측할 수 있는 방법이 필요하다.

태스크의 대략적인 시간 예측을 위해서 2장에서 언급한 블록의 성능 정보를 이용할 수도 있다. [9]에서는 블록의 성능 정보를 이용하여 전체 소프트웨어의 대략적인 시간을 예측할 수 있는 방법이 언급되어 있는데 이 방법을 이용해 태스크 별 전체 수행 시간을 예측할 수 있다. 먼저 태스크를 수행하는데 실행되는 블록 B_k에 대해서 $P(k,i)$, $m(k)$, $c(k,l)$을 각각 프로세서 P_i에서의 계산 시간, 메모리 접근 회수, 다음에 수행될 블록 B_l과의 통신량이라고 하자. 그리고 $n(k)$는 블록 B_k가 한번 태스크를 수행하는데 수행되는 회수를 나타낸다고 하자. 또한 n_m과 n_c를 각각 메모리 접근 오버헤드와 채널 통신 오버헤드라고 하자. 그러면 태스크를 수행하는데 필요한 시간은 대략 식 (1)과 같이 표현할 수 있다.

$$P_{est} = \sum_{B_k onCP} n(k) \times \{P(k,i) + m(k) \times n_m + c(k,l) \times n_c\} \quad (1)$$

이 식에 얻어진 값은 섹션 5.2에서 언급한 것과 같이 그림 6에 나타난 자료구조의 ***taskTime*** 이라는 변수에 초기화 되어 정리된다. 또한 블록 별 수행시간 정보는 각 태스크 별 배열 형태로 저장이 된다. 이 정보를 바탕으로 자료구조의 ***remainedTime*** 이라는 변수를 이용하여 태스크의 남은 시간을 관리할 수 있다. 즉 태스크가 처음 수행되었을 때 ***remainedTime*** 을 ***taskTime*** 으로 초기화 한 뒤에 블록이 수행될 때마다 블록의 수행 시간을 ***remainedTime*** 에서 줄여주는 방법을 통해 시간을 관리할 수 있다. 또한 전체 시스템 시간을 저장하는 변수를 하나 두고 블록의 수행 시간을 더해가는 방법을 통해서 전체 시간도 관리할 수 있다.

주의할 점은 이 값에 사용된 블록의 정보는 대략적인 시간으로 실제 시간과 차이가 있을 수 있다. 특히 블록 내부에 복잡한 수행 분기가 존재할 경우 수행 시간의 변화가 매우 클 수 있으며, 이는 실시간 태스크 스케줄링이 실패하는 원인이 될 수 있다. 이 경우는 성능 분석에 사용된 값에 여유를 둬서 사용하는 것을 통해 어느 정보 보완이 가능하나 엄격한 실시간 시스템에는 적합하지 않은 방법이며, 시스템의 시간 정보를 직접 이용할 수 있는 라이브러리가 반드시 필요하다.

6. 블록 클러스터링

직렬화된 멀티태스킹 코드에서 태스크코드가 작게 분할될수록 스케줄을 체크하고 문맥전환이 일어나는 빈도수가 증가하게 되어 실행 부담이 증가하게 된다. [12]에서는 코드를 자르는 적당한 위치를 찾기 위해서 응답시간(response time)이란 개념을 정의하고 있다. 응답시간은 태스크 코드가 수행되다가 스케줄러가 불리는데 걸릴 수 있는 최대 시간을 의미하며, 어떤 이벤트가 외부에서 들어왔을 때 이벤트가 시스템에서 처리되는데 걸리는 최대 시간을 가늠하는데 지표가 된다. 특히 제안하는 방법에서는 블록 단위로 코드를 분할하여 생성하는데, 이 때 수행시간이 작은 블록이 존재한다면 불필요한 스케줄링 부담이 추가된다. 따라서 이러한 블록을 묶어 스케줄링 부담을 줄이는 방법이 필요하다.

[12]에서는 원하는 응답시간을 직렬화된 코드에서 얻기 위하여 소프트웨어를 직접 수행하여 응답시간을 맞추는 방법을 제안하고 있다. 이 방법에서는 동기화 지점에서만 최소한으로 분할한 코드로 시작하여 그 소프트웨어를 직접 수행시켜 각 코드 조각이 원하는 응답시간 안에 종료되지 못할 경우 그 조각을 계속 잘라 나간다. 이 과정을 모든 코드 조각이 원하는 응답시간 안에 종료할 때까지 반복하게 된다.

이 방법은 실제 제안하는 설계 환경에서 사용하기에는 분석 시간 측면에서 어려움이 따른다. 설계 공간 탐색이 진행되는 과정에서 다양한 아키텍처 후보가 생성이 되는데, 각 후보마다 실제 코드를 수행해 볼 수 있는 빠른 환경이 존재하기는 어려우며, 느린 시뮬레이션 환경 등에서 최적의 코드 조각 크기를 찾기 위해 반복적으로 소프트웨어 코드를 실행해 보는 것은 매우 시간이 많이 걸린다. 따라서 이 연구에서는 기존의 방법 대신에 블록 별 수행 시간 정보를 이용하여 블록을 묶는 방법을 사용하였다.

그림 9는 정적 스케줄링 결과에 따라 태스크 내부의 블록을 돌며 주어진 응답시간 내에서 묶는 알고리즘의 의사 코드를 보여준다. 알고리즘의 기본은 스케줄에 따라서 블록의 수행시간을 누적해 나가다가 주어진 응답시간이 넘어갈 경우 묶은 블록의 코드를 생성하고, 다시 묶기 시작하는 것이다. 이 때 블록이 통신 블록일 경우에는 동기화 지점이기 때문에 별도의 묶음으로 빼주게 된다. 또한 블록이 여러 번 수행되는 루프의 경계에 있는 경우에도 코드를 자르게 된다. 이는 루프가 포함된 묶음이 루프 안에서 잘릴 경우 수행에 문제가 발생하기 때문이며, 루프 안에서 별도로 묶이는 것은 상관이 없다.

실험을 위해서 DivX 동영상 플레이어 예제를 태스크 모델과 SDF코델을 이용하여 그림 2와 유사하게 명세를 하였다. 타겟 시뮬레이션 환경으로는 RealView SoC Designer[15]를 이용하여 그림 10과 같이 하드웨어 플랫폼을 명세하였다. 컴파일러는 RealView Development Suite 2.2[16]에 포함되어있는 C컴파일러를 사용하였으며 최적화 옵션은 -O2를 주었다. 실험에 사용한 플랫폼은 프로세서로 ARM926ej-s를 사용하였으며, 충분한 양의 메모리를 AHB 버스를 통해 연결하였다. 또한 운영체제를 동작시키기 위한 타이머를 APB 버스를 통해 연결하였으며, 타이머는 인터럽트 컨트롤러를 통해 프로세서에 인터럽트를 주기적으로 걸어주게 된다. 이 플랫폼에 DivX 예제를 포팅 하였으며, 176*144 크기의 QCIF 포멧 5 프레임을 디코딩 하며 수행 시간을 실행 싸이클 단위로 측정하였다.

```
1.clustTime ← 0
2.while all blocks are not clustered do
3.  if clustTime+blockTime < responceTime then
4.    If block is not communication block, and there
          is no loop before/after block then
5.        clustTime ← clustTime + blockTime
6.        continue
7.    end if
8.  end if
9.  generates clustered block code
10. clustTime ← 0
11.end while
```

그림 9 블록 클러스터링 알고리즘의 의사 코드 실험

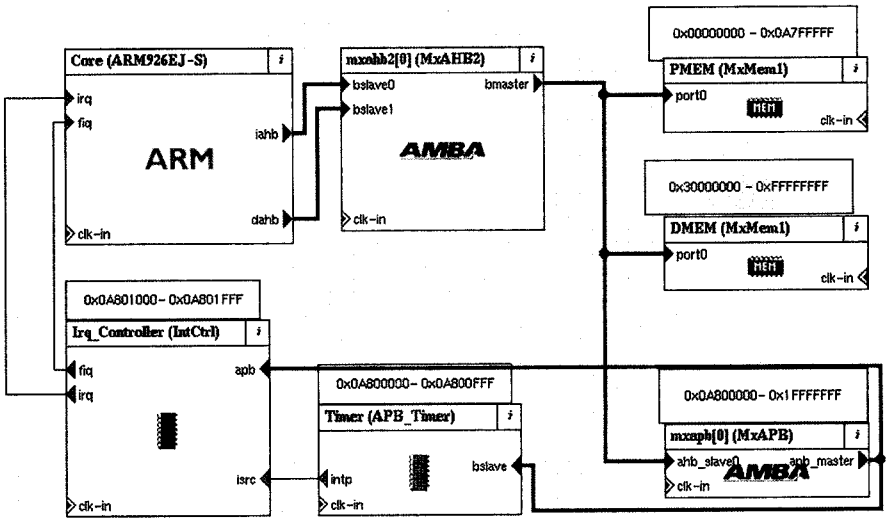

그림 10 SoC Designer를 이용한 실험 플랫폼 명세

직렬화된 코드의 응답시간 변화에 따른 실행 부의 변화를 알아보기 위해 DivX예제의 응답시간을 다양하게 변경하며 코드를 생성하였다. 또한 직렬화된 코드의 성능 비교 및 추가 실행 부담을 측정하기 위한 비교 자료로서 직렬화 되지 않은 코드를 비선점형으로 순차적으로 실행시킨 코드와 가장 간단한 실시간 운영체제 가운데 하나인 uCos[17]를 이용한 두 개의 실험군을 만들어 추가로 성능 실험을 하였다.

표 1은 응답시간 변화에 따른 직렬화된 코드의 수행시간 부담의 변화를 보여준다. 여기서 측정한 수행 시간은 공정한 비교를 위해 전체 시간에서 초기화 시간을 뺀, 프로그램이 수행되며 반복적으로 호출되는 메인 알고리즘의 수행 시간만을 측정하였다. 또한 각 경우에 대해서 추가되는 실행시간 부담을 비교하기 위한 기준 시간을 측정하기 위해 직렬화되지 않은 태스크 코드의 {task name}_go()함수를 비선점형, 순환형으로 호출하는 코드를 수행하였다. 응답시간은 싸이클 단위로 0(블록 클러스터링을 하지 않음), 3000, 10000, 100000, 그리고 무한대(최대한으로 클러스터링)의 5가지 경우에 대해 실험하였다. 표의 값은 수행시간(싸이클)과 기준시간대비 시간 증가량(%)을 기록하였다.

또한 각 응답시간 별로 함수 호출 부담(*케이스 1*), 문맥 전환 부담(*케이스 2*), 순환형태의 스케줄러 부담(*케이스 3*)에 대해서 따로 실험을 하였다. 함수 호출 부담을 측정하기 위해선 직렬화 된 코드를 섹션 5.2.1의 비선점형 스케줄러에 의해 수행하였다. 또한 문맥전환 부담을 위해선 checkScheduler()함수가 매번 1을 반환하도록 하여 스케줄러로 빠져나가게 하되, 스케줄러 내부에서는 태스크 코드가 종료될 때까지 실행이 다른 태스크로 넘어가지 않도록 반복하여 태스크 코드를 다시 호출하도록 하여 측정을 하였다. 마지막으로 순환형태 스케줄러 부담을 측정하기 위해선 문맥전환 부담을 측정하기 위한 코드의 스케줄링 코드에 섹션 5.2.2에서 설명한 스케줄러를 작성하여 수행하였다.

실행 시간 부담 증가에 직접적인 영향을 주는 요인은 응답시간 자체 보다 문맥전환이 몇 번 발생하는가와 문맥전환이 발생할 수 있는 지점이 얼마나 되는가에 달려있다. 표 1에서는 실험에 사용한 코드 내부에서 checkScheduler()함수가 항상 1을 리턴하도록 설정하였기 때문에 블록 함수의 경계에서 문맥전환이 매번 발생한다고 가정하여 응답 시간 별 문맥전환 숫자를 기록하였다.

표 1에서 태스크를 블록 단위로 잘라 함수 형태로 싸는 것은 0.12%~0.5%사이의 실행 부담만 증가하며 문맥전환 회수에 따른 가감이 존재하기는 하지만, 전반적으로 크게 부담이 되지 않음을 볼 수 있다. 그러나 문맥전환이 일어나거나 실제 순환순서형 스케줄러를 사용하면 수행시간 부담은 점차 증가하며, 문맥전환 회수에 따른 부담의 증가도 큰 차이를 보인다.

전반적으로 문맥전환 회수가 감소함에 따라 실행시간 부담도 감소하는 것을 보이나 몇 경우에서는 오히려 실행시간이 증가하고 있다. 이것의 원인을 살펴보기 위해 클러스터링 된 블록의 개수를 바꿔가면 태스크의 내부 수행시간 변화를 살펴 보았다. 그 결과 태스크의 수행시간 자체는 클러스터링을 많이 하여 함수 호출을 줄인다고 반드시 좋아지는 것이 아님을 관측하였다. H.263 디코더 태스크의 경우 문맥전환 회수가 3019에서 2509가 될 때(클러스터 된 블록의 개수가 7개에서 4개로 변함), 그리고 MP3 디코더 태스크의 경우 2509에서 2494가

표 1 응답시간 변화에 따른 직렬화된 코드의 수행시간 부담 변화

Execution Time(cycles) / Additional Overhead(%)					
Response time	0	3,000	10,000	100,000	∞
Context switch	10444	5989	3019	2509	2494
The number of clustered blocks (AVI/H263/MP3)	1 / 16 / 5	1 / 10 / 5	1 / 7 / 5	1 / 4 / 4	1 / 4 / 1
Case 1: Function call	19,396,050 / 0.4633	19,369,013 / 0.3232	19,337,643 / 0.1608	19,349,112 / 0.2202	19,338,210 / 0.1637
Case 2: Context switch	20,645,122 / 6.9010	20,083,339 / 4.0231	19,738,388 / 2.2364	19,684,060 / 1.9550	19,736,108 / 2.2210
Case 3: RR scheduler	21,759,357 / 11.8805	20,737,519 / 7.4115	20,109,650 / 4.1594	19,995,112 / 3.5662	20,029,233 / 3.6614

Non-preemptive Schedule: 19,306,607 uCos(Context switch: 15): 20,377,330 cycles / 4.9207%

될 때(클러스터 된 블록의 개수가 4개에서 1개로 변함) 오히려 태스크의 수행시간이 증가함을 보였다. 이 경우 하나의 함수로 묶인 블록들의 코드 크기가 매우 거대하였으며, 따라서 컴파일러 최적화 등을 수행하기 더 어려웠을 것으로 추측된다.

표 1에서는 케이스1의 경우 문맥전환 회수가 3019에서 2509가 될 때, 케이스2와 3의 경우는 2509에서 2494가 될 때 문맥전환 회수가 감소 했음에도 전체 수행 시간이 오히려 증가하는 것을 보인다. 3019에서 2509가 될 때는 함수로 묶인 코드의 크기로 인해 태스크 내부의 수행 시간이 증가 했음에도 문맥전환 회수에 따른 실행 시간 부담의 감소가 그것을 상쇄하여 전체 수행시간은 감소하고 있다. 반면에 2509에서 2494가 될 때는 문맥전환 회수의 감소가 미미하여 컴파일러 최적화의 감소로 인한 수행시간 증가가 실험 결과에 드러나게 되었다.

그림 11은 표 1에서 각 문맥전환 회수의 변화에 따른 순환순서형 스케줄링 부담의 변화를 그래프 형태로 정리하였다. 여기서 문맥전환 회수가 많아지면 실시간 운

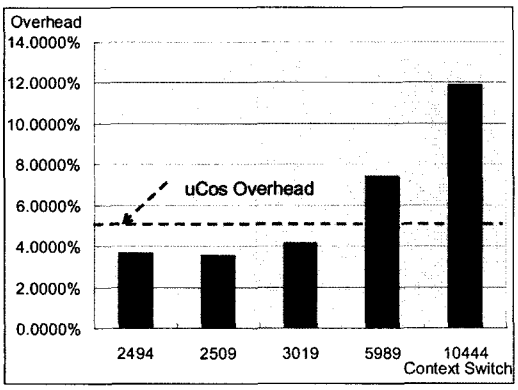

그림 11 생성된 코드의 문맥 변환 회수와 수행 부담의 변화

영체제의 실행 부담보다 오히려 추가적인 부담이 많아질 수도 있음을 알 수 있다. 따라서 효율적인 직렬화 코드를 얻기 위해선 응답시간을 적절하게 잘 설정해 주는 것이 매우 중요함을 알 수 있다. 그러나 비교에 사용된 실시간 운영체제의 경우 문맥전환이 15번 밖에 안 일어났음을 고려할 때 대부분의 경우에 있어서 직렬화된 코드가 운영체제를 이용한 코드에 비해서 추가적인 부담이 작을 것으로 예상된다. 또한 운영체제를 타겟 아키텍쳐에 포팅하는데는 많은 시간과 노력을 필요로 하는 반면, 직렬화된 멀티 태스크 코드는 일반 C 컴파일러로 컴파일 가능한 C 코드이기 때문에 별도의 포팅 부담이 적다. 따라서 제안하는 방법이 설계 효율 측면에서는 매우 큰 효과를 기대할 수 있다.

8. 평가

제안하는 방법은 기존의 직렬화 컴파일러 연구와 다르게 컴파일러 적인 기법을 사용하지 않고 데이타 플로우 모델의 특성을 이용하여 멀티 태스크 응용을 직렬화 한다. 방법이 다르기 때문에 기존 방법에 비해 장점과 단점이 존재한다.

8.1 장점

데이타 플로우 명세로부터 직렬화된 멀티 태스킹 코드를 생성하는 방법의 장점은 다음과 같이 요약할 수 있다.

- 방법이 간단하며, 데이타플로우 형태의 명세로부터 시스템을 개발하는 환경에 쉽게 적용이 가능하다.
- 데이타 플로우 블록은 그 자체로 완전하며, 수행 후에 입/출력을 위한 포트 변수를 제외한 블록의 모든 상태가 유지될 필요가 없다. 따라서 블록 단위로 코드를 자를 경우 Live 변수를 별도로 처리하는 등의 추가적인 부담이 필요 없다.
- 블록 별 코드 형태가 유지된다. 이것을 통해서 블록별 분석 방법 등이 가능하며, 블록 정보를 이용한 코

드 수행(실시간 스케줄링 등)이 가능하다. 또한 디버깅 정보가 유지된다.

- 태스크 코드와 스케줄링 코드의 분리가 가능하며, 태스크 코드 생성 이후에 고유의 스케줄링 코드 작성 및 수정이 용이하다. 이는 생성된 코드의 스케줄링의 단위가 데이타플로우 블록이기 때문에 가능한 점으로, 컴파일러 기법에 의해 분할 된 코드라면 개발자가 수동으로 스케줄링 코드를 작성하기가 어려울 것이다. 기존의 직렬화 컴파일러 기법에서는 개발자 고유의 스케줄러를 이용하기 위해선 컴파일러 내부를 수정해야 했지만, 제안하는 방법에서는 분리되어 있기 때문에 고유의 스케줄러 생성을 추가하거나 별도로 작성하는 것이 용이하다.
- 스케줄링 오버헤드와 응답시간을 고려한 코드 조각의 크기를 결정할 때, 코드를 반복적으로 수행할 필요가 없다. 블록은 재사용 가능한 단위이며, 성능 정보 또한 재사용이 가능하다. 따라서 블록의 성능 정보를 이용하여 코드를 직접 수행하지 않고 분석에 의한 크기 결정이 가능하다.

8.2 단점

제안하는 방법의 가장 큰 단점은 블록 내부를 세부적으로 잘라서 응답시간을 맞출 수 없다는 것과 그 결과 태스크의 데드라인을 어길 수도 있다는 것이다. 만약 블록의 수행시간이 응답시간 제약조건보다 크다면 제약조건을 맞추는 코드생성은 불가능하다. 반면에 기존의 컴파일러 기법을 사용한 방법이라면 블록 내부의 코드를 분할하여 제약조건을 만족시킬 수도 있을 것이다.

그러나 Y-chart 접근방법을 사용하는 개발 환경에서 알고리즘을 아키텍쳐에 매핑하는 과정은 블록 단위로 이루어지게 되며, 이 경우 매핑 결과는 블록단위 스케줄링으로 만족시킬 수 있는 결과를 보인다. 따라서 이러한 환경에서 얻어진 분할 결과로부터 블록 단위로 스케줄링 하는 직렬화 코드를 생성하였을 때도 태스크의 실행주기를 맞출 수 있다.

반면에 외부 이벤트가 발생하였을 때에 어떤 블록의 수행시간이 응답시간보다 크다면 그 블록을 수행하는 사이에 응답시간을 놓칠 수도 있다. 그러나 어떤 블록의 수행시간이 다른 블록에 비해 특별히 큰 경우, 명세한 알고리즘의 분석 및 최적화가 어렵게 되어 그 알고리즘 명세는 좋은 명세라고 보기 어렵다. 따라서 이 경우 큰 블록을 보다 작은 여러 블록으로 분할하여 명세 하는 것이 보다 바람직하다. 실제 실험에 사용한 DivX예제의 경우는 주어진 응답시간을 초과할 정도의 큰 블록은 존재하지 않았으며, 일반적으로 잘 명세 된 데이타 플로우 모델의 경우 응답시간을 초과하는 경우가 많이 발생하지 않을 것으로 사료된다.

그러나 만약 어떤 큰 블록을 작은 블록으로 나누어 명세하기 어려운 경우는 기존의 직렬화 컴파일러 기법을 블록 내부 코드에 적용하여 블록 내부의 코드를 직렬화 시키는 것을 통해 해결이 가능하다. 이 경우 응답시간도 블록 크기에 관계 없이 세밀하게 조절이 가능하며, 블록 단위의 정보 또한 어느 정도 유지할 수 있다.

9. 결론 및 추가 연구 방향

설계 과정에서 효율적인 설계 공간 탐색을 위해 타겟 아키텍쳐에 운영체제를 포팅하지 않고 멀티 태스킹 코드를 수행할 수 있는 방법이 필요하다. 기존의 직렬화 컴파일러 기법이 이것의 해결책이 될 수 있으나, 직렬화 컴파일러 기법은 상위 수준 명세 정보를 잃어버리는 단점을 보인다. 따라서 이 연구에서는 상위 수준 데이타플로우 명세로부터 직렬화된 코드를 생성하는 방법을 제안하였다.

제안하는 방법에서는 태스크의 스케줄링 가능한 지점을 태스크들을 구성하고 있는 블록의 경계점으로 설정하고 각 블록을 함수 형태로 코드를 생성하였다. 그 뒤에 스케줄러를 필요한 각종 자료구조를 생성하여 그 자료구조 및 정보를 이용하여 멀티 태스크 스케줄러를 작성할 수 있도록 하였다. 이렇게 함으로 직렬화 가능한 태스크 코드 생성과 스케줄러 작성을 분리하였으며, 블록 별 상위수준 정보를 최종 코드까지 유지하도록 하였다.

제안한 방법은 멀티 프로세서 환경까지 확장이 가능하나, 이 논문에서는 단일 프로세서 개발 환경으로 설계 환경을 구축하였다. 그러나 현재 시스템은 멀티 프로세서 환경으로 확장되고 있으며, 직렬화된 코드생성 기법 또한 멀티 프로세서 환경을 고려한 효율적인 코드 생성 방법에 대한 고려가 필요하다.

제안하는 방법을 멀티프로세서 환경에서 효율적으로 적용하기 위해서는 몇 가지 확장이 필요하다. 먼저 태스크간 통신 방법의 확장이다. 멀티 프로세서 환경에서는 태스크간 통신이 하드웨어 구조 및 태스크 구조에 따라 달라지게 된다. 이러한 다양한 요소를 고려하여 태스크간 통신이 이루어져야 한다. 또한 스케줄링 방법도 더욱 복잡해지게 된다. 단일 프로세서에서는 모든 태스크의 상태 변화를 프로그램 내부에서 알 수 있지만, 멀티 프로세서 환경이 되면 코드 내부에서 다른 프로세서에 매핑 된 태스크 정보를 알기가 어려워진다. 따라서 효율적인 스케줄링을 위해 필요한 자료구조 및 정보를 정리하고 스케줄러를 만들기 위한 가이드라인 정립이 필요하다. 더욱이 다양한 병렬성을 고려한 스케줄링이 가능하도록 확장이 필요하다.

참 고 문 헌

[1] Synopsys Inc., 700 E. Middlefield Rd., Mountain View, CA94043, USA, COSSAP User's Manual.

[2] Signal Processing Designer(SPW) product site of CoWare, http://www.coware.com/products/signalprocessingSPW.

[3] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," Intl. Journal of Computer Simulation, special issue on "Simulation Software Development," Vol.4, pp. 155-182, 1994.

[4] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. Joo, "PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems," in ACM Trans. on Design Automation of Electronic Systems(TODAES) Vol.12(3), Article 24, 2007.

[5] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization of Concerns and Platform-Based Design," in IEEE Trans. on Computer-Aided Design, 19(12), pp. 1523-1543, 2000.

[6] D. Kim, M. Kim, and S. Ha, "A Case Study of System Level Specification and Software Synthesis of Multi-mode Multimedia Terminal," in Proc. of Workshop on Embedded Systems for Real-Time Multimedia(ESTIMedia), 2003.

[7] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," in Proc. of IEEE, 1987.

[8] C. Lee and S. Ha, "Hardware-Software Cosynthesis of Multitask MPSoCs with Real-Time Constraints," in Proc. of IEEE The 6th Intl. Conf. on ASIC, Shanghai, China, Vol.2, pp. 919-924, 2005.

[9] S. Kwon, C. Lee, S. Kim, Y. Yi and S. Ha, "Fast Design Space Exploration Framework with an Efficient Performance Estimation Technique," in Proc. of Intl. Second Workshop on Embedded Systems for Real Time Multimedia(ESTIMedia), pp. 27-32, 2004.

[10] A. Gerstlauer, H. Yu, and D. Gajski, "RTOS Modling for System Level Design," in Proc. of Intl. Conf. on Design Automation & Test in Europe (DATE), 2003.

[11] L. Gauthier, S. Yoo, and A. A. Jerraya, "Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software," in IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol.20(11), pp. 1293-1301, 2001.

[12] A. C. Nacul and T. Givargis, "Phantom: a serializing compiler for multitasking embedded software," in Proc. of American Control Conference, 2006.

[13] Y. Cho, N. Zergainoh, K. Choi, and A. A. Jerraya, "Low runtime-overhead software synthesis for communicating concurrent processes," in Proc. of 18th IEEE/IFIP Intl. Workshop on Rapid System Prototyping, 2007.

[13] A. C. Nacul and T. Givargis, "Code Partitioning for Synthesis of Embedded Applications with Phantom," in Proc. of ICCAD, 2004.

[14] RealView SoC Designer official homepage, http://www.arm.com/products/DevTools/MaxSim.html

[15] RealView Development Suite official homepage, http://www.arm.com/products/DevTools/RealViewDevSuite.html

[16] uC/OS-II Kernel Overview site, http://www.micrium.com/products/rtos/kernel/rtos.html

권 성 남

2002년 2월 서울대학교 컴퓨터공학과 학사. 2002년 3월~현재 서울대학교 전기컴퓨터공학부 석박사 통합과정. 관심분야는 하드웨어-소프트웨어 통합설계, 소프트웨어 성능 예측, MPSoC 내장형 소프트웨어 설계

하 순 회

1985년 서울대학교 전자공학과 학사. 1987년 서울대학교 전자공학과 석사. 1992년 미국 UCB 전기컴퓨터공학과 박사. 1993년~1994년 현대전자 근무. 1994년~현재 서울대학교 전기컴퓨터공학부 교수 관심분야는 하드웨어-소프트웨어 통합설계, 내장형 시스템을 위한 설계 방법론, MPSoC 내장형 소프트웨어 설계