

 (19) 대한민국특허청(KR) (12) 공개특허공보(A)	(11) 공개번호 10-2012-0070215 (43) 공개일자 2012년06월29일
(51) 국제특허분류(Int. Cl.) G06F 9/44 (2006.01) G06F 9/46 (2006.01) (21) 출원번호 10-2010-0131674 (22) 출원일자 2010년12월21일 심사청구일자 없음	(71) 출원인 삼성전자주식회사 경기도 수원시 영통구 삼성로 129 (매탄동) 서강대학교산학협력단 서울특별시 마포구 백범로 35 (신수동, 서강대학교) (72) 발명자 박수용 서울특별시 마포구 백범로 35, Ricci관 911호 (신수동, 서강대학교) 이승원 경기도 화성시 영통로27번길 53, 신영통현대2차 아파트 212동 901호 (반월동) (뒷면에 계속) (74) 대리인 특허법인무한

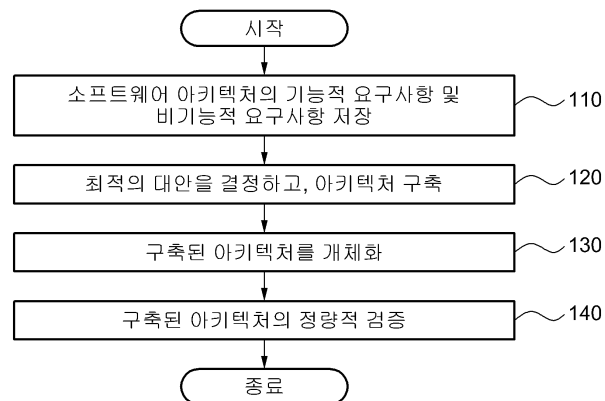
전체 청구항 수 : 총 23 항

(54) 발명의 명칭 **멀티미디어 아키텍처 패턴 결정 방법, 싱글코어에서 멀티코어 기반으로의 아키텍처 전환 장치 및 방법**

(57) 요약

싱글코어 기반의 소프트웨어 어플리케이션을 멀티코어 기반으로 전환하는 아키텍처를 구축하는 방법 및 장치 및 멀티미디어 시스템에서 아키텍처 패턴을 결정하는 방법을 제공한다. 아키텍처 결정 지원도구(Architectural Decision Supporter)를 이용함으로써, 소프트웨어 아키텍처에 대한 사전 지식 및 축적된 지식 없이도 아키텍처 구축 작업을 수행할 수 있다. 이미 검증된 동시성 관련 소프트웨어 아키텍처 패턴을 이용함으로써, 아키텍처를 구축하는 과정에서 발생하는 오류를 최소화할 수 있고, 따라서 전반적인 소프트웨어의 품질을 향상시키고, 개발시간을 단축시킬 수 있다.

대표도 - 도1



(72) 발명자

이시화

서울특별시 강남구 일원로 120, 샘터마을아파트
109동 304호 (일원동)

박수진

서울특별시 마포구 상수동 396번지 두산위브 102
동 101호

특허청구의 범위

청구항 1

싱글 코어 기반의 레거시 소스 코드에 기초하여 도출된(Elicited), 멀티 코어 기반으로의 전환을 위한, 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항을 저장하는 단계;

상기 기능적 요구사항 및 비기능적 요구사항을 고려하여 생성된 아키텍처의 대안들을 저장하고, 상기 아키텍처의 대안들 중에서, 품질속성에 기초한 최적의 대안을 결정하고, 아키텍처를 구축(Authoring)하는 단계;

상기 구축된 아키텍처에 대응하는 컴포넌트를 결정하여, 상기 구축된 아키텍처를 개체화(Instantiation)하는 단계; 및

상기 구축된 아키텍처에 대하여, 멀티 코어 개수에 비례하여 시스템의 성능이 개선되는지를 정량적으로 검증(Evaluation)하는 단계

를 포함하는 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 2

제1항에 있어서,

상기 기능적 요구사항 및 비기능적 요구사항을 저장하는 단계는

싱글 코어 기반 레거시 시스템의 소스 코드를 분석하는 단계;

상기 분석된 내용에 대해 행위 다이어그램을 적용하여 상기 시스템의 기능을 가시적으로 분류하는 단계;

상기 분류된 시스템의 기능을 수행하는 동작과 관련하여, 이벤트의 순서를 고려하여 유스케이스(Use Case) 명세서를 생성하는 단계;

품질 속성 시나리오를 사용하여, 상기 아키텍처의 비기능적 요구사항을 추출하는 단계;

상기 분류된 시스템의 기능을 구현하는데 제약이 되는 제약사항을 입력 받는 단계;

상기 분류된 시스템의 기능, 상기 아키텍처의 비기능적 요구사항 및 상기 제약사항을 반영하여 소프트웨어 요구사항 명세서를 생성하는 단계; 및

상기 소프트웨어 요구사항 명세서를 저장하는 단계

를 포함하는 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 3

제2항에 있어서,

상기 싱글 코어 기반 레거시 시스템의 소스 코드를 분석하는 단계는

함수설명(Function Description), 자료구조(Data Structure) 및 호출관계(Call Relationship)를 상기 레거시 시스템의 소스 코드에서 추출하여 소스코드 분석서를 생성하는

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 4

제3항에 있어서,

상기 시스템의 기능을 분류하는 단계는

상기 소스코드 분석서의 내용을 UML(Unified Modeling Language)의 행위 다이어그램을 이용하여 가시적인 형태로 변환하고, 시스템 행위 다이어그램을 생성하는

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 5

제4항에 있어서,

상기 유스케이스(Use Case) 명세서를 생성하는 단계는

상기 시스템 행위 다이어그램에 기초하여, 유스케이스의 각 스텝에 병렬성(parallelism)-수행하는 행위(activity)가 동시에 여러 개의 프로세서에서 실행 가능한지 여부-을 반영한 유스케이스 명세서를 생성하는

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 6

제5항에 있어서,

상기 병렬성은

복수의 함수(Function)가 동시에 여러 개의 프로세서에서 실행(Parallel) 가능한 경우 및 도메인 데이터를 분할하여 동일한 함수(Funtion)가 동시에 여러 개의 프로세서에서 실행(Parallel) 가능한 경우

를 포함하는 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 7

제2항에 있어서,

상기 제약사항은

상기 아키텍처를 구축하는 과정에서 아키텍처 패턴의 선택 또는 각 컴포넌트의 기술적 구현을 제한하는 요소들

을 포함하는 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 8

제2항에 있어서,

상기 소프트웨어 요구사항 명세서를 생성하는 단계는

상기 유스케이스 명세서, 상기 품질 속성 시나리오 및 상기 제약사항에 기초하여 소프트웨어 요구사항 명세서를 생성하는

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 9

제1항에 있어서,

상기 아키텍처를 구축하는 단계는

시스템 행위 다이어그램에 기초하여 싱글 코어 기반의 참조 아키텍처를 생성하는 단계;

아키텍처 결정 지원도구(Architectural Decision Supporter)를 사용하여 적용 가능한 적어도 하나의 아키텍처 패턴을 획득하는 단계;

소프트웨어 요구사항 명세서, 상기 참조 아키텍처 및 상기 획득된 적어도 하나의 아키텍처 패턴에 기초하여 멀티코어 기반의 시스템을 위한 아키텍처의 대안을 생성하는 단계; 및

상기 아키텍처의 대안에 대하여, 트레이드-오프(Trade-off) 분석을 통하여 최적의 대안을 결정하는 단계

를 포함하는 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 10

제9항에 있어서,

상기 싱글 코어 기반의 참조 아키텍처를 생성하는 단계는

시스템 행위 다이어그램에 식별된 행위들을 기존 소스코드 구조와 비교하여, 하나의 컴포넌트에 함께 수록되어야 할 행위들을 식별하고, 상기 하나의 컴포넌트를 동시성 컴포넌트로 하여, 동시성 컴포넌트들과 대응되는 물리적 컴포넌트들을 매핑하고, 상기 동시성 컴포넌트들을 시스템의 동작 순서와 비교하여 정적 아키텍처 및 동적 아키텍처를 생성하는

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 11

제9항에 있어서,

상기 아키텍처 패턴을 획득하는 단계는

아키텍처 드라이버에 기초하여 상기 아키텍처 결정 지원도구에서 상기 적어도 하나의 아키텍처 패턴을 획득하고,

상기 아키텍처 결정 지원도구는 상기 아키텍처 드라이버와 상기 적어도 하나의 아키텍처 패턴을 포함하는 아키텍처 지식 관리 도구인

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 12

제9항에 있어서,

상기 멀티코어 기반의 시스템을 위한 아키텍처의 대안을 생성하는 단계는

아키텍처 드라이버와 상기 적어도 하나의 아키텍처 패턴간의 상관관계, 상기 적어도 하나의 아키텍처 패턴의 선택에 필요한 체크리스트 및 제약사항에 기초하여 상기 아키텍처의 대안을 생성하는

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 13

제12항에 있어서,

상기 트레이드-오프 분석을 통한 최적의 대안을 결정하는 단계는

품질속성 시나리오 중 성능(Performance)관련 시나리오에 기초하여 우선순위를 반영한 품질속성 트리를 생성하는 단계; 및

상기 품질속성 트리 상에 부여된 시나리오의 개발 우선순위에 기초하여 트레이드-오프(Trade-Off) 분석을 통해 최적의 대안을 결정하는 단계를 포함하고,

상기 품질속성 트리는 품질속성, 품질 고려사항 및 품질 속성 시나리오 순으로 계층이 구분되는

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 14

제1항에 있어서,

상기 아키텍처를 개체화하는 단계는

아키텍처 결정 지원도구(Architectural Decision Supporter)를 사용하여 컴포넌트를 검색하는 단계;

상기 검색된 컴포넌트들 중에서, 상기 구축된 아키텍처에 대응하는 컴포넌트를 결정하는 단계; 및

상기 결정된 컴포넌트를 멀티코어 기반의 아키텍처에 적합한 형태로 수정하는 단계

를 포함하는 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 15

제14항에 있어서,

상기 아키텍처 결정 지원도구(Architectural Decision Supporter)를 사용하여 컴포넌트를 검색하는 단계는
아키텍처 상의 컴포넌트 중에서 이미 개발된 컴포넌트가 있는지를, 컴포넌트 명칭 또는 컴포넌트의 기능을 키워드로 하여, 상기 아키텍처 결정 지원도구를 통해, 검색하는
싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 16

제14항에 있어서,

상기 컴포넌트를 결정하는 단계는

상기 아키텍처 결정 지원도구에서 지원되는 참고자료에 기초하여, 상기 검색된 컴포넌트 중에서, 상기 구축된 아키텍처 상에 식별된 컴포넌트의 역할(role)을 수행하는 물리적 컴포넌트를 결정하는

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 17

제1항에 있어서,

상기 검증하는 단계는

상기 구축된 아키텍처를 검증하기 위한 프로토타입을 생성하는 단계; 및

상기 프로토타입에 기초하여 성능 인자(factor)별로 상기 구축된 아키텍처를 검증하는 단계를 포함하는 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 18

제17항에 있어서,

상기 프로토타입을 생성하는 단계는

구축된 소프트웨어 아키텍처 및 싱글코어 기반으로 개발된 코드에 기초하여, 정량적으로 성능 측정이 가능한 형태의 시스템 프로토타입을 생성하는

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 19

제18항에 있어서,

상기 성능 인자(factor)별로 상기 구축된 아키텍처를 검증하는 단계는

상기 시스템 프로토타입에 대하여, 품질속성 시나리오 중 반응을 측정하는 기준(Response Measure)을 근거로, 검증 결과(Evaluation Results)를 측정하고, 수용범위(Acceptance Range)에 기초하여, 상기 구축된 소프트웨어 아키텍처를 정량적으로 검증하는

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법.

청구항 20

제1항 내지 제19항 중 어느 한 항의 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법을 수행하는 프로그램을 수록한 컴퓨터 판독 가능 기록 매체.

청구항 21

싱글 코어 기반의 레거시 소스 코드를 입력받는 입력부;

상기 싱글 코어 기반의 레거시 소스 코드에 기초하여, 멀티 코어 기반으로의 전환을 위한, 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항을 추출하는 추출부;

상기 기능적 요구사항 및 비기능적 요구사항을 고려하여 아키텍처의 대안들을 생성하고, 상기 아키텍처의 대

안들 중에서, 품질속성에 기초한 적어도 하나의 최적의 아키텍처를 결정하는 제1 처리부;

상기 적어도 하나의 최적의 아키텍처에 대응하는 컴포넌트를 결정하여, 상기 적어도 하나의 최적의 아키텍처를 개체화(Instantiation)하고, 상기 적어도 하나의 최적의 아키텍처에 대하여, 멀티 코어 개수에 비례하여 시스템의 성능이 개선되는지를 정량적으로 검증(Evaluation)하는 제2 처리부;

상기 정량적으로 검증된 상기 적어도 하나의 최적의 아키텍처를 출력하는 출력부; 및

상기 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항, 상기 품질속성, 상기 아키텍처의 대안들 및 아키텍처 결정 지원도구를 저장하는 저장부

를 포함하는 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치.

청구항 22

실행시간/속도(Execution Time/Speed), 메모리 요구(Memory Demand), 메모리 대역폭(Memory Bandwidth) 및 부하 밸런스 필요성(Load Balancing Needs)을 포함하는 아키텍처 드라이버들을 입력 받는 단계;

소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항에 기초하여 상기 아키텍처 드라이버들의 우선순위를 결정하는 단계; 및

상기 결정된 아키텍처 드라이버들의 우선순위에 기초하여, 매핑 테이블에서 적어도 하나의 아키텍처 패턴을 결정하는 단계를 포함하고,

상기 매핑 테이블은 동시성 관련 아키텍처 패턴들과 상기 아키텍처 드라이버들을 포함하고, 상기 동시성 관련 아키텍처 패턴과 상기 아키텍처 드라이버들의 우선순위는 체크 리스트에 기초하여 서로 매핑되는

싱글 코어에서 멀티 코어 기반으로의 멀티미디어 아키텍처 패턴 결정 방법.

청구항 23

제22항에 있어서,

상기 적어도 하나의 아키텍처 패턴을 결정하는 단계는

상기 결정된 아키텍처 드라이버들의 우선순위와 일치하는 아키텍처 패턴을 추천 패턴(Recommended Pattern)으로 결정하고, 상기 결정된 아키텍처 드라이버들의 우선순위와 부분적으로 일치하는 아키텍처 패턴을 부분적용 가능 패턴(Partially Adaptable Pattern)으로 결정하는

싱글 코어에서 멀티 코어 기반으로의 멀티미디어 아키텍처 패턴 결정 방법.

명세서

기술분야

[0001] 기술분야는 멀티미디어 시스템에서 아키텍처 패턴을 결정하는 방법, 싱글코어 기반의 임베디드 소프트웨어 어플리케이션을 멀티코어 기반으로 전환하는 아키텍처를 구축하는 방법 및 장치에 관한 것이다.

배경기술

[0002] 하드웨어 시스템에서, 합리적인 가격대의 멀티코어 아키텍처가 선보이고 있으며, 지원되는 코어의 개수는 증가하고 있다. 듀얼코어 랩탑이 발표된 이후로, 쿼드코어 PC를 거쳐, 현재는 8-코어 서버가 시장에 출시되고 있다. 그런데, 18개월마다 하드웨어의 성능이 두 배씩 증가하던 현상이 주춤하고 있다. 즉, 하드웨어에만 의존한 시스템의 성능 향상은 그 한계를 드러내고 있다.

[0003] 멀티코어 병렬화(Parallelization)는 단지 컴파일러나 운영체제(OS)의 문제가 아닌 소프트웨어 개발 전략의 문제이다. 즉, 아무리 많은 개수의 코어를 지원하는 하드웨어, 컴파일러 및 운영체제가 출시된다고 하더라도, 하드웨어에 탑재되어 동작하는 소프트웨어가 병렬성(Parallelism)을 고려하지 않은 상태로 개발된다면, 고사양의 하드웨어, 컴파일러 및 운영체제는 무용지물이 된다.

[0004] 따라서, 멀티코어 기반의 시스템이 성능을 최대한 발휘하기 위해서는 소프트웨어의 개발 전략을 결정하는 소프트웨어 아키텍처 구축 방법 및 새로운 프레임워크가 필요하다.

- [0005] 현재까지 발표된 일반적인 아키텍처 구축 방법에는 CMU SEI(Carnegie Mellon University Software Engineering Institute)에서 2000년대 초반부터 연구 발표하고 있는 QAW(Quality Attribute Workshop), ADD(Attribute-Driven Design) Method, ATAM(Architecture Tradeoff Analysis Method), ARID(Active Reviews for Intermediate Designs) 등이 있다.
- [0006] 위의 4가지 방법들은 소프트웨어의 아키텍처를 구축하는 데 있어서 소프트웨어가 만족시켜야 할 품질 속성을 근거로 작성된 시나리오 기반의 요구사항들을 시작점으로 한다.
- [0007] 또한, 4가지 방법들은 각 품질 속성과 관련된 요구사항들과 아키텍처 구축의 결과물 사이의 연관관계를 유지하는 데 필요한 작업의 스텝을 정의한다. 그러나, 4가지 방법들은 스텝에 있어서 상당 부분 중복되어 있으며, 특정 도메인에 대한 고려 없이 범용적으로 정의되었다.
- [0008] 따라서, 각 스텝에서 수행해야 할 작업이 무엇(What)인지만 명시되어 있을 뿐, 어떻게(How) 수행해야 할 지에 대한 세부적인 가이드라인은 제공되고 있지 않다.

발명의 내용

과제의 해결 수단

- [0009] 일 측면에 있어서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법은 싱글 코어 기반의 레거시 소스 코드에 기초하여 도출된(Elicited), 멀티 코어 기반으로의 전환을 위한, 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항을 저장하는 단계, 상기 기능적 요구사항 및 비기능적 요구사항을 고려하여 생성된 아키텍처의 대안들을 저장하고, 상기 아키텍처의 대안들 중에서, 품질속성에 기초한 최적의 대안을 결정하고, 아키텍처를 구축(Authoring)하는 단계, 상기 구축된 아키텍처에 대응하는 컴포넌트를 결정하여, 상기 구축된 아키텍처를 개체화(Instantiation)하는 단계 및 상기 구축된 아키텍처에 대하여, 멀티 코어 개수에 비례하여 시스템의 성능이 개선되는지를 정량적으로 검증(Evaluation)하는 단계를 포함한다.
- [0010] 상기 기능적 요구사항 및 비기능적 요구사항을 저장하는 단계는 싱글 코어 기반 레거시 시스템의 소스 코드를 분석하는 단계, 상기 분석된 내용에 대해 행위 다이어그램을 적용하여 상기 시스템의 기능을 가시적으로 분류하는 단계, 상기 분류된 시스템의 기능을 수행하는 동작과 관련하여, 이벤트의 순서를 고려하여 유스케이스(Use Case) 명세서를 생성하는 단계, 품질 속성 시나리오를 사용하여, 상기 아키텍처의 비기능적 요구사항을 추출하는 단계, 상기 분류된 시스템의 기능을 구현하는데 제약이 되는 제약사항을 입력 받는 단계, 상기 분류된 시스템의 기능, 상기 아키텍처의 비기능적 요구사항 및 상기 제약사항을 반영하여 소프트웨어 요구사항 명세서를 생성하는 단계 및 상기 소프트웨어 요구사항 명세서를 저장하는 단계를 포함할 수 있다.
- [0011] 상기 싱글 코어 기반 레거시 시스템의 소스 코드를 분석하는 단계는 함수설명(Function Description), 자료구조(Data Structure) 및 호출관계(Call Relationship)를 상기 레거시 시스템의 소스 코드에서 추출하여 소스코드 분석서를 생성할 수 있다.
- [0012] 상기 시스템의 기능을 분류하는 단계는 상기 소스코드 분석서의 내용을 UML(Unified Modeling Language)의 행위 다이어그램을 이용하여 가시적인 형태로 변환하고, 시스템 행위 다이어그램을 생성할 수 있다.
- [0013] 상기 유스케이스(Use Case) 명세서를 생성하는 단계는 상기 시스템 행위 다이어그램에 기초하여, 유스케이스의 각 스텝에 병렬성(parallelism)-수행하는 행위(activity)가 동시에 여러 개의 프로세서에서 실행 가능한지 여부-을 반영한 유스케이스 명세서를 생성할 수 있다.
- [0014] 상기 병렬성은 복수의 함수(Function)가 동시에 여러 개의 프로세서에서 실행(Parallel) 가능한 경우 및 도메인 데이터를 분할하여 동일한 함수(Funtion)가 동시에 여러 개의 프로세서에서 실행(Parallel) 가능한 경우를 포함할 수 있다.
- [0015] 상기 품질 속성은 싱글 코어에서 멀티 코어로의 아키텍처 전환 과정에서 요구되는 사항이, 소프트웨어의 품질 속성 범주 중에서 어디에 속하는지를 나타낸 카테고리(Category), 시스템에 가해지는 자극에 대한 근원을 반영한 소스(Source), 상기 시스템에 가해지는 자극(Stimulus), 상기 자극에 대하여 반응이 일어나도록 하는 환경(Environment), 상기 반응의 대상이 되는 것이 무엇인지를 반영한 아티팩트들(Artifacts), 상기 자극에 대한 시스템의 반응양식(Response) 및 상기 반응양식을 측정하는 기준(Response Measure)을 포함할 수 있다.
- [0016] 상기 제약사항은 상기 아키텍처를 구축하는 과정에서 아키텍처 패턴의 선택 또는 각 컴포넌트의 기술적 구현

을 제한하는 요소들을 포함할 수 있다.

- [0017] 상기 소프트웨어 요구사항 명세서를 생성하는 단계는 상기 유스케이스 명세서, 상기 품질 속성 시나리오 및 상기 제약사항에 기초하여 소프트웨어 요구사항 명세서를 생성할 수 있다.
- [0018] 상기 아키텍처를 구축하는 단계는 시스템 행위 다이어그램에 기초하여 싱글 코어 기반의 참조 아키텍처를 생성하는 단계, 아키텍처 결정 지원도구(Architectural Decision Supporter)를 사용하여 적용 가능한 적어도 하나의 아키텍처 패턴을 획득하는 단계, 소프트웨어 요구사항 명세서, 상기 참조 아키텍처 및 상기 획득된 적어도 하나의 아키텍처 패턴에 기초하여 멀티코어 기반의 시스템을 위한 아키텍처의 대안을 생성하는 단계 및 상기 아키텍처의 대안에 대하여, 트레이드-오프(Trade-off) 분석을 통하여 최적의 대안을 결정하는 단계를 포함할 수 있다.
- [0019] 상기 싱글 코어 기반의 참조 아키텍처를 생성하는 단계는 시스템 행위 다이어그램에 식별된 행위들을 기존 소스코드 구조와 비교하여, 하나의 컴포넌트에 함께 수록되어야 할 행위들을 식별하고, 상기 하나의 컴포넌트를 동시성 컴포넌트로 하여, 동시성 컴포넌트들과 대응되는 물리적 컴포넌트들을 매핑하고, 상기 동시성 컴포넌트들을 시스템의 동작 순서와 비교하여 정적 아키텍처 및 동적 아키텍처를 생성할 수 있다.
- [0020] 상기 아키텍처 패턴을 획득하는 단계는 아키텍처 드라이버에 기초하여 상기 아키텍처 결정 지원도구에서 상기 적어도 하나의 아키텍처 패턴을 획득하고, 상기 아키텍처 결정 지원도구는 상기 아키텍처 드라이버와 상기 적어도 하나의 아키텍처 패턴을 포함하는 아키텍처 지식 관리 도구일 수 있다.
- [0021] 상기 멀티코어 기반의 시스템을 위한 아키텍처의 대안을 생성하는 단계는 아키텍처 드라이버와 상기 적어도 하나의 아키텍처 패턴간의 상관관계, 상기 적어도 하나의 아키텍처 패턴의 선택에 필요한 체크리스트 및 제약사항에 기초하여 상기 아키텍처의 대안을 생성할 수 있다.
- [0022] 상기 트레이드-오프 분석을 통한 최적의 대안을 결정하는 단계는 품질속성 시나리오 중 성능(Performance)관련 시나리오에 기초하여 우선순위를 반영한 품질속성 트리를 생성하는 단계 및 상기 품질속성 트리 상에 부여된 시나리오의 개발 우선순위에 기초하여 트레이드-오프(Trade-Off) 분석을 통해 최적의 대안을 결정하는 단계를 포함하고, 상기 품질속성 트리는 품질속성, 품질 고려사항 및 품질 속성 시나리오 순으로 계층이 구분될 수 있다.
- [0023] 상기 아키텍처를 개체화하는 단계는 아키텍처 결정 지원도구(Architectural Decision Supporter)를 사용하여 컴포넌트를 검색하는 단계, 상기 검색된 컴포넌트들 중에서, 상기 구축된 아키텍처에 대응하는 컴포넌트를 결정하는 단계 및 상기 결정된 컴포넌트를 멀티코어 기반의 아키텍처에 적합한 형태로 수정하는 단계를 포함할 수 있다.
- [0024] 상기 아키텍처 결정 지원도구(Architectural Decision Supporter)를 사용하여 컴포넌트를 검색하는 단계는 아키텍처 상의 컴포넌트 중에서 이미 개발된 컴포넌트가 있는지를, 컴포넌트 명칭 또는 컴포넌트의 기능을 키워드로 하여, 상기 아키텍처 결정 지원도구를 통해, 검색할 수 있다.
- [0025] 상기 컴포넌트를 결정하는 단계는 상기 아키텍처 결정 지원도구에서 지원되는 참고자료에 기초하여, 상기 검색된 컴포넌트 중에서, 상기 구축된 아키텍처 상에 식별된 컴포넌트의 역할(role)을 수행하는 물리적 컴포넌트를 결정할 수 있다.
- [0026] 상기 검증하는 단계는 상기 구축된 아키텍처를 검증하기 위한 프로토타입을 생성하는 단계 및 상기 프로토타입에 기초하여 성능 인자(factor)별로 상기 구축된 아키텍처를 검증하는 단계를 포함할 수 있다.
- [0027] 상기 프로토타입을 생성하는 단계는 구축된 소프트웨어 아키텍처 및 싱글코어 기반으로 개발된 코드에 기초하여, 정량적으로 성능 측정이 가능한 형태의 시스템 프로토타입을 생성할 수 있다.
- [0028] 상기 성능 인자(factor)별로 상기 구축된 아키텍처를 검증하는 단계는 상기 시스템 프로토타입에 대하여, 품질속성 시나리오 중 반응을 측정하는 기준(Response Measure)을 근거로, 검증 결과(Evaluation Results)를 측정하고, 수용범위(Acceptance Range)에 기초하여 상기 구축된 소프트웨어 아키텍처를 정량적으로 검증할 수 있다.
- [0029] 다른 일 측면에 있어서, 싱글 코어에서 멀티 코어 기반으로의 멀티미디어 아키텍처 패턴 결정 방법은 실행시간/속도(Execution Time/Speed), 메모리 요구(Memory Demand), 메모리 대역폭(Memory Bandwidth) 및 부하 밸런싱 필요성(Load Balancing Needs)을 포함하는 아키텍처 드라이버들을 입력 받는 단계, 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항에 기초하여 상기 아키텍처 드라이버들의 우선순위를

결정하는 단계 및 상기 결정된 아키텍처 드라이버들의 우선순위에 기초하여, 매핑 테이블에서 적어도 하나의 아키텍처 패턴을 결정하는 단계를 포함하고, 상기 매핑 테이블은 동시성 관련 아키텍처 패턴들과 상기 아키텍처 드라이버들을 포함하고, 상기 동시성 관련 아키텍처 패턴과 상기 아키텍처 드라이버들의 우선순위는 체크 리스트에 기초하여 서로 매핑될 수 있다.

[0030] 상기 적어도 하나의 아키텍처 패턴을 결정하는 단계는 상기 결정된 아키텍처 드라이버들의 우선순위와 일치하는 아키텍처 패턴을 추천 패턴(Recommended Pattern)으로 결정하고, 상기 결정된 아키텍처 드라이버들의 우선순위와 부분적으로 일치하는 아키텍처 패턴을 부분적용 가능 패턴(Partially Adaptable Pattern)으로 결정할 수 있다.

[0031] 일 측면에 있어서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 싱글 코어 기반의 레거시 소스 코드를 입력받는 입력부, 상기 싱글 코어 기반의 레거시 소스 코드에 기초하여, 멀티 코어 기반으로의 전환을 위한, 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항을 추출하는 추출부, 상기 기능적 요구사항 및 비기능적 요구사항을 고려하여 아키텍처의 대안들을 생성하고, 상기 아키텍처의 대안들 중에서, 품질속성에 기초한 적어도 하나의 최적의 아키텍처를 결정하는 제1 처리부, 상기 적어도 하나의 최적의 아키텍처에 대응하는 컴포넌트를 결정하여, 상기 적어도 하나의 최적의 아키텍처를 개체화(Instantiation)하고, 상기 적어도 하나의 최적의 아키텍처에 대하여, 멀티 코어 개수에 비례하여 시스템의 성능이 개선되는지를 정량적으로 검증(Evaluation)하는 제2 처리부, 상기 정량적으로 검증된 상기 적어도 하나의 최적의 아키텍처를 출력하는 출력부 및 상기 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항, 상기 품질속성, 상기 아키텍처의 대안들 및 아키텍처 결정 지원도구를 저장하는 저장부를 포함한다.

발명의 효과

[0032] 아키텍처 결정 지원도구(Architectural Decision Supporter)를 이용함으로써, 소프트웨어 아키텍처에 대한 사전 지식 및 축적된 지식 없이도 아키텍처 구축 작업을 수행할 수 있다.

[0033] 또한, 이미 검증된 동시성 관련 소프트웨어 아키텍처 패턴을 이용함으로써, 아키텍처를 구축하는 과정에서 발생하는 오류를 최소화할 수 있고, 따라서 전반적인 소프트웨어의 품질을 향상시키고, 개발시간을 단축시킬 수 있다.

[0034] 또한, 아키텍처와 관련된 지식을 저장 공간에 저장하고, 접근 가능하게 함으로써, 지속적으로 아키텍처 구축에 관련된 지식을 재사용할 수 있다.

[0035] 또한, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법을 이용함으로써, 임베디드 소프트웨어 어플리케이션의 종류와 무관하게 싱글 코어 기반에서 멀티코어 기반으로 전환할 수 있다.

도면의 간단한 설명

[0036] 도 1은 일 실시예에 따른 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법의 흐름도이다.

도 2는 일 실시예에 따른 소프트웨어 아키텍처의 요구사항을 저장하는 단계를 구체화한 흐름도이다.

도 3은 일 실시예에 따른 소스 코드 분석의 일 예를 나타낸 도면이다.

도 4는 일 실시예에 따른 행위 다이어그램을 적용한 일 예를 나타낸 도면이다.

도 5는 일 실시예에 따른 아키텍처를 구축하는 단계를 구체화한 흐름도이다.

도 6은 일 실시예에 따른 식별된 컴포넌트와 시스템의 동작을 매핑시킨 도면이다.

도 7은 일 실시예에 따른 참조 아키텍처 중 정적 아키텍처의 일 예를 나타낸 도면이다.

도 8은 일 실시예에 따른 참조 아키텍처 중 동적 아키텍처의 일 예를 나타낸 도면이다.

도 9는 일 실시예에 따른 패턴 선택 지원자(Pattern Selection Supporter)를 나타낸 도면이다.

도 10은 일 실시예에 따른 시스템 아키텍처의 대안의 일 예를 나타낸 도면이다.

도 11은 도 10의 시스템 아키텍처의 대안을 상호협력 다이어그램을 통해 나타낸 도면이다.

도 12A 및 도 12B는 일 실시예에 따른 시스템 아키텍처 대안의 또 다른 일 예를 나타낸 도면이다.

도 13은 도 12B의 시스템 아키텍처의 대안을 상호협력 다이어그램을 통해 나타낸 도면이다.

도 14는 일 실시 예에 따른 비디오 디코더 시스템의 품질 속성 트리를 나타낸 도면이다.

도 15는 일 실시 예에 따른 최적의 대안을 결정하는 방법의 개념을 나타낸 도면이다.

도 16은 일 실시 예에 따른 구축된 아키텍처를 개체화하는 단계를 구체화한 흐름도이다.

도 17 내지 도 19는 일 실시 예에 따른 소프트웨어 자산을 나타낸 도면이다.

도 20은 일 실시 예에 따른 구축된 아키텍처를 정량적으로 검증하는 단계를 구체화한 흐름도이다.

도 21은 일 실시 예에 따른 싱글 코어에서 멀티 코어 기반으로의 멀티미디어 아키텍처 패턴 결정 방법의 흐름도이다.

도 22는 일 실시 예에 따른 멀티미디어 아키텍처 패턴을 결정하는 일 예를 나타낸 도면이다.

도 23은 일 실시 예에 따른 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치의 블록도이다.

발명을 실시하기 위한 구체적인 내용

[0037] 이하, 일측에 따른 실시예를 첨부된 도면을 참조하여 상세하게 설명한다.

[0038] 일 측에 따른 싱글코어에서 멀티코어 기반으로의 아키텍처 전환 방법은 싱글코어 기반의 임베디드 소프트웨어 어플리케이션을 멀티코어 기반으로 전환하는 데 있어서 필요한 아키텍처를 구축하는 데 필요한 방법을 제안한다. 일 측에 따른 싱글코어에서 멀티코어 기반으로의 아키텍처 전환 방법은 임베디드 소프트웨어 어플리케이션의 종류와 무관하게 적용 가능하나, 주로 핸드폰이나 MP3, PDA 등과 같은 휴대 가능한 소형 모바일 디바이스에 탑재되는 어플리케이션을 적용 대상으로 한다.

[0039] 또한, 일 측에 따른 싱글코어에서 멀티코어 기반으로의 아키텍처 전환 방법은 소프트웨어의 전체 개발 주기의 관점에서, 멀티코어 기반의 소프트웨어의 개발에 필요한 요건을 파악하는 요구사항 분석 단계에서부터 아키텍처의 수립 및 수립된 아키텍처의 검증에 이르기까지 과정을 포함하고 있다.

[0040] 대부분의 임베디드 소프트웨어 개발 조직들은 모델 기반의 소프트웨어 개발 방식보다는 구현 코드 기반의 개발 방식을 택하고 있다. 그러나, QAW나 ADD와 같은 범용 프로세스들은 비즈니스 드라이버들로부터 아키텍처 구축을 위한 최초 데이터를 수집하는 형태의 방법을 제시하고 있기 때문에, 별도의 커스터마이징(customizing) 작업 없이 QAW나 ADD의 스텝에 따라 아키텍처를 진행할 경우, 많은 오버헤드가 발생하게 된다. 따라서, 일 측에 따른 싱글코어에서 멀티코어 기반으로의 아키텍처 전환 방법은 싱글코어에서 멀티코어로의 전환을 위해 커스터마이징된 프로세스로, 아키텍처 전환 방법의 시작점을 이미 존재하는 싱글코어의 레거시 소스코드로부터 잡고 있다.

[0041] 기존의 아키텍처 검증 방법인 ATAM이나 ARID 등은 작성된 아키텍처가, 주어진 품질 요구사항에 비추어 적합한지의 여부를, 요구사항으로부터의 추론 작업을 통해 정성적으로 분석하였다. 그러나, 싱글코어에서 멀티코어 아키텍처로의 전환은 '코어 개수에 비례하는 제품의 성능 향상'이라는 명확하고 단일한 목적을 가지고 있다. 따라서, 아키텍처 검증 단계에서 반드시 제품의 성능을 정량적으로 측정할 수 있는 검증 방안이 필요하다.

[0042] 따라서, 일 측에 따른 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법은, 아키텍처의 최종 검증이 완료되기 이전에, 제안된 아키텍처에 따라 프로세스를 각 코어에 할당하는 프로토타이핑 작업을 진행한다. 이때, 정량적인 성능을 측정하고, 그 결과가 아키텍처 구축 시작 시점에 목표로 했던 성능 요구사항의 범위를 만족시키는지를 검증한다.

[0043] 도 1은 일 실시예에 따른 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 방법의 흐름도이다.

[0044] 110단계에서, 일 실시예에 따른 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항을 추출하고, 저장한다. 여기서, 소프트웨어 아키텍처의 기능적 요구사항 및 비기능적 요구사항은 싱글 코어 기반의 레거시 소스 코드에 기초하여 도출(Elicited)된다. 또한, 소프트웨어 아키텍처의 기능적 요구사항 및 비기능적 요구사항은 멀티 코어 기반으로의 전환을 위해 요구되는 사항이다. 소프트웨어 아키텍처의 기능적 요구사항 및 비기능적 요구사항을 도출하는 방법과 도출된 결과를 저장하는 방법은 도 2 내지 도 4에서 좀 더 상세하게 설명한다.

[0045] 120단계에서, 일 실시예에 따른 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처의 대안

들을 생성하고, 저장한다. 또한, 아키텍처의 대안들 중에서 최적의 대안을 결정하고, 상기 최적의 대안에 기초하여 아키텍처를 구축(Authoring)한다. 아키텍처의 대안들은 상기 기능적 요구사항 및 비기능적 요구사항을 고려하여 생성된다. 최적의 대안은 품질속성에 기초하여 결정된다. 아키텍처의 대안들을 생성하는 방법 및 최적의 대안을 결정하는 방법에 대해서는 도 5 내지 도 15에서 좀 더 상세하게 설명한다.

[0046] 130단계에서, 일 실시예에 따른 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 구축된 아키텍처를 개체화(Instantiation)한다. 상기 구축된 아키텍처를 개체화하는 과정은 상기 구축된 아키텍처에 대응하는 컴포넌트를 결정하는 과정으로부터 시작된다. 개체화 과정에 대해서는 도 16 내지 도 19에서 좀 더 상세하게 설명한다.

[0047] 140단계에서, 일 실시예에 따른 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 구축된 아키텍처에 대하여, 정량적으로 검증(Evaluation)한다. 정량적 검증은 상기 구축된 아키텍처에 대하여 멀티 코어 개수에 비례하여 시스템의 성능이 개선되는지를 측정함으로써 이루어진다.

[0048] 도 2는 일 실시예에 따른 소프트웨어 아키텍처의 요구사항을 저장하는 단계(110)를 구체화한 흐름도이다.

[0049] 210단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 싱글코어 기반 레거시 시스템의 소스코드를 분석한다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 사용자로부터 입력된 기준에 기초하여 싱글코어 기반 레거시 시스템의 소스코드를 분석할 수 있다. 싱글코어 기반 레거시 시스템은 싱글 코어를 위해 생성된 일반적인 시스템을 의미할 수 있다. 예를 들면, H.264 비디오 디코더(decoder) 시스템을 의미한다. 소스코드의 분석은 레거시 시스템의 소스코드를 함수설명(Function Description), 자료구조(Data Structure) 및 호출관계(Call Relationship)에 따라 분류함으로써, 이루어질 수 있다.

[0050] 또한, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 소스 코드를 함수설명(Function Description), 자료구조(Data Structure) 및 호출관계(Call Relationship)에 따라 분류함으로써, 소스코드 분석서를 생성할 수 있다. 소스코드의 분석은 소스코드 분석서에 기초하여 이루어질 수 있다. 또한, 소스코드의 분석과정은 특정 시스템의 프로세서에서 수행되고, 다양한 컴퓨터 수단을 통하여 수행될 수 있는 프로그램 명령 형태로 구현되어, 컴퓨터 판독 가능 매체에 기록될 수 있다.

[0051] 220단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 분석된 내용에 대해 행위 다이어그램을 적용하여 상기 시스템의 기능을 가시적으로 분류할 수 있다. 행위 다이어그램은 시스템에서 행위와 행위간의 제어의 흐름을 보여주는 워크플로우(workflow)를 나타내는 흐름도이다. 여기서 행위는 시스템의 기능을 수행하는 여러 동작들을 의미할 수 있다.

[0052] 상기 소스코드 분석서의 내용은 UML(Unified Modeling Language)의 행위 다이어그램을 이용하여 가시적인 형태로 변환될 수 있다. 상기 가시적인 형태로 변환된 행위 다이어그램을 시스템 행위 다이어그램이라고 할 수 있다. 시스템 행위 다이어그램을 통해 시스템의 행위를 몇 개의 의미있는 단계(Phase)로 식별할 수 있다.

[0053] 230단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 가시적으로 분류된, 시스템의 기능을 수행하는 동작과 관련하여, 이벤트의 순서를 고려하여 유스케이스(Use Case) 명세서를 생성할 수 있다. 이벤트의 순서는 시스템의 동작을 시간의 흐름에 따라 고려한 것일 수 있다. 또한, 유스케이스 명세서는 사용자로부터 입력된 기준에 기초하여 생성될 수도 있다.

[0054] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 멀티코어로의 전환을 위한 추가적인 요건으로서, 유스케이스의 각 스텝에 병렬성(Parallelism) 적용 여부를 표시할 수 있다. 병렬성은 각 스텝에서 수행되는 행위(activity)가 동시에 여러 개의 프로세서에서 실행 가능한지 여부를 의미한다. [표 1]은 예시 시스템인 비디오 디코더의 디코딩 유스케이스에서 병렬성의 적용 여부가 표시된 경우를 나타낸다.

표 1

[0055]	BF3	Entropy Decoding을 시작한다.	FP_01
		-----중략-----	
	BF4	Transform 을 시작한다.(Transform Module)	FP_01
		-----중략-----	
	BF5	Prediction을 시작한다.(Prediction Module)	FP_01
		-----중략-----	

BF6	Deblocking 을 수행한다. (Deblocking Filter Module)	FP_01
BF6.1	DeblockAreaFetch - SrcY, SrcU, SrcV로 부터 DeblockUnit으로 data 복사한다. (FN05-CN52)	DP_03
BF6.2	Deblock의 자료를 입력으로 deblocking 진행한다. (FN08-CN53)	DP_03

[0056] 여기서, FP_01은 여러 개의 행위(Activity), 그 중에서도 함수(Function)가 동시에 병렬적으로(Parallel) 실행이 가능한 경우를 나타낸다. 'FP_' 다음에 표기된 숫자는 동일한 단계에서, 행위(Activity)들끼리 함수의 병렬성(Functional Parallelism)을 구현함을 나타낸다. 기본 플로우 스텝(Basic Flow Step) 3,4,5,6은 함수의 병렬성(Functional Parallelism)으로 구현될 부분으로, 동시에 수행될 수 있는 행위(Activity)임을 의미한다.

[0057] DP_03은 도메인 데이터(Domain Data)를 적절히 분할하여 동시에 동일한 함수(Function)을 병렬적으로(Parallel) 실행이 가능한 경우를 나타낸다. 'DP_' 다음에 표기된 숫자는 동일한 단계에서, 행위(Activity)들끼리 데이터 병렬성(Data Parallelism)을 구현함을 나타낸다.

[0058] [표 1]을 참조하면, 시스템의 기능을 수행하는 동작은 Entropy Decoding, Transform, Prediction, Deblocking으로 구별되고, 각 동작은 병렬적으로 동시에 여러 개의 프로세서에서 실행 가능하다. 또한, Deblocking을 수행하는 구체적인 동작들(BF6.1, BF6.2)도 동시에 여러 개의 프로세서에서 실행 가능하다.

[0059] 아래 [표 2]는 예시 시스템인 비디오 디코더에 대하여, 디코딩 유스케이스에 대한 명세서의 전체 구성 및 각 구성에 대응하는 내용을 나타낸다.

표 2

Requirement ID	H.264에 따른 비디오Decoding		
설명	H.264 규약에 따라 동작하는 Video Decoder가 입력되는 Video Data의 압축을 풀어 화면상에 출력하기까지 시스템의 동작을 기술함		
Pre-condition	입력 Video Data가 DRAM상에 load되어 있음		Parallelism
Post-condition	입력된 Video Data가 출력 Buffer상에 모두 출력된 상태		
개발우선순위	1		
기본동작	BF1	비디오 디코더에게 디코딩 메시지가 도착한다.	N
	BF2	Input Data Processing 을 시작한다.	N
	BF2.1	remained_file_size, bitstream_pos 초기화한다. (FN01-CN01)	N
	BF2.2	bitstream_pos[]에서 FileBuffer[HALF_BUFFER_SIZE]로 HALF_BUFFER_SIZE 만큼 복사한다.	N
	BF2.3	stream_byte_start_pos(FileBuffer[]의 byte 단위 data 시작 index)를 설정한다. (FN01-CN02)	N
	BF2.4	FileBuffer[HALF_BUFFER_SIZE]로 HALF_BUFFER_SIZE 만큼 복사 후 stream_byte_start_pos 변경한다.	N
	BF2.5	next_start_pos(FileBuffer[]에서 다음번 시작 data index)를 설정한다. (FN01-CN03)	N
	BF2.6	LongStreamBitStartPos, BitstreamBufferPtr 설정한다. (FN01-CN04)	N
	BF2.7	BitstreamBufferLong[]을 parsing 한다. NAL Header Type이 유효하지 않으면, EF3으로 분기한다.	N
	BF2.7.1	[NAL Header Type == Slice Data, Decoding Reset] slice 관련 정보 추출하여 저장한다. (FN02-CN08,CN09)	N
	BF2.7.1.1	[iEndofFrame == 1] BF2.4 스텝으	N

[0060]

		로 돌아감	
	BF2.7.1.2	[iEndofFrame != 1] 영상복원 시작하여 BF2.8로 간다. (FN01-CN08)	N
	BF2.7.2	[NAL Header Type == SPS] sequence 관련 정보 seq_parameter_set 에 저장한다. (CN06)	N
	BF2.7.3	[NAL Header Type == PPS] picture 관련 정보 pic_parameter_set 에 저장한다. (CN07)	N
	BF2.8	Syntax 분석한다. Syntax에 맞지 않는 입력일 경우 BF1으로 분기한다.	N
	BF3	Entropy Decoding을 시작한다.	FP_01
	BF3.1	MacroBlock(MB) Type Decoding (FN03-CN10) 을 시작한다.	DP_01
	BF3.1.1	[MB type == Intra-Picture 4x4] Intra Prediction 을 시작한다.	DP_01
	BF3.1.2	L4x4 화소 parsing한다. (FN04-CN11)	DP_01
	BF3.1.3	CodedBlockPatternChroma, currMB->c_ipred_mode, currMB->i16mode 값을 설정한다. (FN04-CN11)	DP_01
	BF3.1.4	IntraPredDataFetch를 수행한다. (FN05-CN12)	DP_01
	BF3.2.1	[MB type == Predictive-Picture] Inter-Predction 을 시작한다.	DP_01
	BF3.2.1.1	[MB Type == p_16x16] p_16x16 화소 Processing : psMv0X, Y에 값 생성한다. (FN03-CN13)	DP_01
	BF3.2.1.2	motion vector를 추출한 후 gp_ref_idx[], g_current_pic_mv[]에 저장한다. (FN03-CN14)	DP_01
	BF3.2.1.3	currMB->RefArea에 listX의 image data를 복사한다. (FN05-CN15)	DP_01
	BF3.2.2.1	[MB type == P_8x8 or P_8x8ref] P_8x8 or P_8x8ref 화소 처리 : psMv0X, Y에 값을 생성한다. (FN04-CN16)	DP_01
	BF3.2.2.1.	[Avail] MV8x8_Avail : 계산된	DP_01

[0061]

	1	Motion vector 값 설정한다.(FN04-CN17)	
	BF3.2.2.1.2	[notAvail] MV8x8_notAvail : 계산된 Motion vector 값을 설정한다.(CN18)	DP_01
	BF3.2.2.2	currMB->RefArea에 listX의 image data를 복사한다.(FN05-CN19)	DP_01
	BF3.2.3.1	[MB type == P_16x8 or P_8x1] P_16x8 or P_8x16 화소를 처리한다.(FN04-CN20)	DP_01
	BF3.2.3.1.1	[MotionVector(MV) == 8x16] P_8x16 처리한다.(FN04-CH23,FN05-CN24)	DP_01
	BF3.2.3.1.2	[MotionVector(MV) == 16x8] P_16x8 처리한다.(FN04-CN21,FN05-22)	DP_01
	BF3.3	currMB->QP, currMB->QPC 값을 설정한다.	N
	BF3.4	coefficient값을 parsing한다.(FN03-CN25,CN26)	N
	BF4	Transform 을 시작한다. (Transform Module)	FP_01
	BF4.1	currMB->QPC, currMB->QP, currMB->CBP값을 추출한다.	DP_02
	BF4.2	양자화(Dequantization) 및 역변환 작업을 수행한다.	DP_02
	BF4.3	currMB->MBBuffer에 저장한다.	DP_02
	BF5	Prediction을 시작한다. (Prediction Module)	FP_01
	BF5.1	currMB->iMB_type 값을 체크한다.	DP_03
	BF5.1.1	[currMB->iMB_type == P_16x16 or P_16x8 or P_8x16 or P_8x8 or P_8x8ref] 각 타입별로 Mv_x[], Mv_y[], currMB->RefAreaY의 값을 바탕으로 currMB->MBBuffer의 값을 계산한다.(FN09-CN56)	DP_03
	BF5.1.2	[currMB->iMB_type == I_4x4] cg_ipredmode의 값에 따라 currMB->MBBuffer의 다른 영역에서 현재 영역으로 복사한다.(FN07-CN30~CN41)	DP_03

[0062]

	BF5.1.3	[currMB->iMB_type == L16x16] currMB->i16mode에 따라 currMB->MBBuffer의 값을 다른 slice의 영역에서 복사한 다.(FN07-CN57~60)	DP_03
	BF5.2	FMO(Flexible Macroblock ordering) 여부를 체크한다.	N
	BF5.2.1	[Non-FMO인 경우] NonFMO 처리 한다. (FN02-CN50)	N
	BF5.2.2	[FMO인 경우] FMO처리한다. (FN02-CN51)	N
	BF5.3	currMB->MbInxX,Y를 설정한다.	N
	BF6	Deblocking 을 수행한다. (Deblocking Filter Module)	FP_01
	BF6.1	DeblockAreaFetch - SrcY, SrcU, SrcV로 부터 DeblockUnit으로 data 복사한다. (FN05-CN52)	DP_04
	BF6.2	Deblock의 자료를 입력으로 deblocking 진행한다. (FN08-CN53)	DP_04
	BF6.3	currMB->MbIdxX,Y 및 다양한 자 료 구조(g_current_pic_mv[], gp_ref_idx[]) 초기화한다. (FN02-CN54)	N
	BF6.4	출력 buffer list에 맨 뒤에 새로운 출력 buffer 추가한다. (FN020-CN55)	N
	BF6.5	해당 출력 buffer를 화면에 출력한 다. (FN020-CN55)	N
대체동작	AF1	해당사항 없음	
예외동작	EF1	BF2.8에서 Syntax에 맞지 않는 입력일 경우, 예 러 메시지를 출력하고 프로그램 종료	
	EF2	BF5.1.1에서 BF5.1.3사이에 currMB->MBBuffer 의 입력 데이터 주소가 잘못된 경우에는 예러 메 시지를 출력하고 프로그램 종료	
	EF3	BF2.7에서 NAL Header Type이 유효하지 않으면 버퍼 포인터를 next_start_pos까지 이동한 후 BF2.7에서 다시 실행	
품질요구사항	QR1	해당사항 없음	

[0063]

[0064]

[표 2]에는 [표 1]과 같이 병렬성(Parallelism) 적용 여부가 표시되어 있다. 여기서, N은 병렬적으로 (Parallel) 실행이 불가능한 행위를 나타낸다. FP, DP를 통하여 각 단계에서 병렬적으로 여러 개의 프로세서에서 실행되는 행위(Activity)들을 식별할 수 있다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 병렬성이 적용되는 행위들을 추출함으로써 시스템의 기능적인 요구사항을 추출할 수 있다.

[0065]

240단계에서, 싱글코어에서 멀티코어 기반으로의 아키텍처 전환 장치는 품질속성 시나리오를 사용하여, 상기 아키텍처의 비기능적 요구사항을 추출할 수 있다. 이때 사용되는 품질속성 시나리오의 양식은 QAW, ATAM, ARID 등에서 제안하는 양식과 동일하다.

[0066]

품질속성 시나리오의 각 항목에는 카테고리(Category), 소스(Source), 자극(Stimulus), 환경(Environment), 아티팩트(Artifacts), 반응(Response) 및 반응양식을 측정하는 기준(Response Measure)이 포함될 수 있다.

[0067]

카테고리(Category)는 싱글코어에서 멀티코어로의 아키텍처 전환 과정에서 요구되는 사항이, 소프트웨어의 품질속성 범주 중에서 어디에 속하는지를 나타낸다. 일반적으로 Performance, Security, Scalability, Usability 등의 다양한 카테고리가 적용되나, 싱글코어에서 멀티코어로의 전환이라는 Context에 대해 특화되어 있는 만큼, 카테고리는 성능(Performance)으로 좁혀져 있다.

[0068]

소스(Source)는 시스템에 가해지는 자극에 대한 근원을 나타낸다. 싱글코어에서 멀티코어로의 전환의 경우에, 소스는 하드웨어 구성으로 표현될 수 있다.

[0069]

자극(Stimulus)은 상기 시스템에 가해지는 이벤트가 무엇인지를 나타낸다. 싱글코어에서 멀티코어로의 전환의 경우에, 코어 수의 변경으로 표현될 수 있다.

[0070]

환경(Environment)은 입력된 자극에 대하여 반응이 일어나도록 하는 조건을 나타낸다. 싱글코어에서 멀티코

어로의 전환의 경우에는 특수한 조건이 필요하지 않고, 일반적인 환경에서 코어 수의 변경에 따라 반응이 일어날 수 있다.

[0071] 아티팩트(Artifacts)는 시스템의 반응의 대상이 되는 것이 무엇인지를 나타낸다. 싱글코어에서 멀티코어로의 전환의 경우에 시스템 자체가 반응의 대상이 된다.

[0072] 반응(Response)은 입력된 자극에 대한 시스템의 반응양식을 나타낸다. 싱글코어에서 멀티코어로의 전환의 경우에 시스템 전원의 반응을 통해 성능을 측정하고자 한다.

[0073] 반응양식을 측정하는 기준(Response Measure)은 입력된 자극에 대한 시스템의 반응을 측정하는 기준 또는 방법을 나타낸다. 싱글코어에서 멀티코어로의 전환의 경우에 시스템 전원의 동작 주파수(Hz)를 기준으로 성능을 평가할 수 있다.

[0074] [표 3]은 비디오 디코더의 경우에 품질 속성 시나리오 예제를 나타낸다.

표 3

[0075]

Requirement-ID	NFR_01
Category	Performance
Source	Hardware 구성
Stimulus	Core 수의 변경
Environment	Normal Operation
Artifacts	System
Response	시스템 전원
Response Measure	시스템 동작 Hz 수
개발우선순위	1
Description	1Core 시스템에서 4Core를 지원하는 시스템으로 변경될 경우, 정상적인 Operation을 수행하는 데 있어서 소모되는 전력이 1Core의 경우와 비교하여 35%의 Hz수를 출력하여야 한다.

[0076] 설명(Description)은 구체적인 자극에 대해 목표로 설정된, 반응 양식을 나타낸다. 즉, 싱글코어에서 멀티코어로의 전환의 경우에 비기능적인 요구사항으로서 성능 측정은 시스템의 동작 주파수(Hz)의 측정에 의하여 이루어질 수 있다.

[0077] 250단계에서, 싱글코어에서 멀티코어 기반으로의 아키텍처 전환 장치는 상기 분류된 시스템의 기능을 구현하는데 제약이 되는 제약사항을 입력 받는다.

[0078] 상기 제약사항은 상기 아키텍처를 구축하는 과정에서 아키텍처 패턴의 선택 또는 각 컴포넌트의 기술적 구현을 제한하는 요소들을 포함한다. 컴포넌트의 구현의 제한은 하드웨어 상으로 구현하는데 한계가 있는 경우 또는 법률적으로 구현이 제약되는 경우를 포함할 수 있다. 기술적, 법률적 제약사항은 데이터베이스에 미리 저장될 수 있다. 싱글코어에서 멀티코어 기반으로의 아키텍처 전환 장치는 기술적, 법률적 제약사항을 데이터베이스에서 읽을 수 있다.

[0079] 예를 들면, 예시 시스템이 하드웨어 측면에서 ARM-SRP 상에 탑재됨에 따라, 물리적 코어의 최대 개수는 4개라는 제약사항을 가진다. 또한 CGA/ Dual Mode가 지원됨에 따라 하드웨어 상에서 Concurrent Execution Category에 속하는 SIMD(Single Instruction Multiple Data)라는 Data Parallelism이 지원됨을 알 수 있다. 물리적 코어의 최대 개수는 4개라는 제약사항 하에서 아키텍처 상에 물리적 코어의 개수를 8개로 설정하는 것은 그 자체로 오류가 된다.

[0080] 260단계에서, 싱글코어에서 멀티코어 기반으로의 아키텍처 전환 장치는 상기 분류된 시스템의 기능, 상기 아키텍처의 비기능적 요구사항 및 상기 제약사항을 반영하여 소프트웨어 요구사항 명세서를 생성하고, 상기 소프트웨어 요구사항 명세서를 저장할 수 있다. 즉, 소프트웨어 요구사항 명세서는 싱글코어에서 멀티코어 기반으로의 아키텍처 전환에 필요한 기능적 요구사항 및 비기능적 요구사항을 반영한다. 기능적 요구사항은 유스케이스 명세서로부터 추출될 수 있으며, 비기능적 요구사항은 품질속성 시나리오에 기초하여 추출될 수 있다.

- [0081] 도 3은 일 실시예에 따른 소스 코드 분석의 일 예를 나타낸 도면이다.
- [0082] 도 3을 참조하면, H.264 비디오 디코더 시스템의 레거시 소스 코드를 분석하는 일 예를 살펴볼 수 있다.
- [0083] 싱글코어에서 멀티코어 기반으로의 아키텍처 전환 장치는 함수설명(Function Description)(301), 자료 구조(Data Structure)(303), 호출관계(Call Relationship)(305)등을 소스코드로부터 추출하여 소스코드 분석서를 생성할 수 있다. 예시 시스템인 비디오 디코더의 경우 C언어로 구현되어 있기 때문에 C 파일의 명칭(307)과 해당 파일에 대한 설명(Description)(309), 그리고 해당 파일에 구현된 함수명칭(311)과 함수 내부의 구현 로직(313)이 기술되어 있다. 소스코드 분석서의 내용은 구현 언어에 따라 변경될 수 있다. 이때, 반드시 기술되어야 하는 요소는 시스템의 행위를 파악하기 위한 내부 로직들이다.
- [0084] 도 4는 일 실시예에 따른 행위 다이어그램을 적용한 일 예를 나타낸 도면이다. 도 4는 H.264 비디오 디코더 시스템의 동작을 행위 다이어그램으로 표현한 것이다.
- [0085] 소스코드 분석서의 내용은 UML(Unified Modeling Language)의 행위 다이어그램을 이용하여 가시적인 형태로 바뀔 수 있다. 행위 다이어그램을 사용한 가시적인 형태를 통해 시스템의 행위를 몇 개의 의미 있는 단계(Phase)로 식별하기 용이하다. 이렇게 식별된 시스템 동작시의 Phase는 아키텍처 구축시 하나의 시스템을 여러 개의 컴포넌트로 구분해야 할 때 의미 있는 리소스로 활용될 수 있다. 또한, 각 Phase의 연결관계도 명확하게 파악할 수 있다.
- [0086] 도 4를 참조하면, H.264 비디오 디코더 시스템은 5개의 Phase로 식별이 될 수 있다. 5개의 Phase는 데이터 전처리 과정(Data Preprocessing)(410), 엔트로피 디코딩과정(Entropy Decoding)(420), 변환과정(Transform)(430), 예측과정(Prediction)(440) 및 디블로킹과정(Deblocking)(450)을 포함한다.
- [0087] 데이터 전처리 과정(Data Preprocessing)(410)은 동작1(411), 동작2(413) 및 동작3(415)의 세부 동작을 통해 수행될 수 있다. 예를 들면, 동작 1(411)은 매크로블록(Macro Block) 정보를 저장하는 자료구조 버퍼를 초기화하는 동작을 의미할 수 있다.
- [0088] 엔트로피 디코딩과정(Entropy Decoding)(420)은 동작 4(421) 및 동작 5(423)의 세부 동작을 통해 수행될 수 있다. 이때, 동작2(413)의 결과에 따라 동작 4(421)가 시작될 수 있다. 예를 들면, 동작 2(413)는 슬라이스(Slice)를 구성하는 매크로블록에 대한 영상 복원 시작을 의미할 수 있다. 동작 4(421)는 처리하고자 하는 매크로블록의 정보를 SPS(Sequence Parameter Set),PPS(Picture Parameter Set),IDR(Instantaneous Decoding Refresh) 버퍼에서 로딩하는 동작을 의미할 수 있다.
- [0089] 변환과정(Transform)(430)은 동작 6(431)의 세부 동작을 통해 수행될 수 있다. 예를 들면, 동작 6(431)은 참조하는 매크로블록의 정보와 이미지 데이터를 로딩하고, 양자화 및 역변환을 통해 복원 매크로블록과 참조 매크로블록간의 오차를 계산하는 동작을 의미할 수 있다.
- [0090] 예측과정(Prediction)(440)은 동작 7(441), 동작 8(443)및 동작 9(445)의 세부 동작을 통해 수행될 수 있다. 예를 들면, 동작 7(441)은 복원할 매크로블록의 타입(type) 정보를 로딩하여 파싱(parsing)하는 동작을 의미할 수 있다.
- [0091] 디블로킹과정(Deblocking)(450)은 동작 10(451)의 세부 동작을 통해 수행될 수 있다. 예를 들면, 동작 10(451)은 데이터를 복원하는 과정에서 발생한 오류, 즉 블록화 현상을 체크하는 동작을 의미할 수 있다.
- [0092] 도 5는 일 실시예에 따른 아키텍처를 구축하는 단계(120)를 구체화한 흐름도이다.
- [0093] 510단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 시스템 행위 다이어그램에 기초하여 싱글 코어 기반의 참조 아키텍처를 생성할 수 있다.
- [0094] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 시스템 행위 다이어그램에서 식별된 행위들을 기존 소스코드 구조와 비교할 수 있다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 비교결과, 하나의 컴포넌트에 함께 수록되어야 할 행위들을 식별하고, 상기 하나의 컴포넌트를 동시성 컴포넌트로서 하여, 동시성 컴포넌트들과 대응되는 물리적 컴포넌트들을 매핑할 수 있다.
- [0095] 여기서, 동시성 컴포넌트는 시스템에서 식별된 phase를 의미하고, 물리적 컴포넌트는 소스코드 파일을 의미할

수 있다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 동시성 컴포넌트들을 시스템의 동작 순서에 따라 배치하여, 참조 아키텍처를 생성할 수 있다. 참조 아키텍처는 정적 아키텍처 및 동적 아키텍처를 포함한다. 정적 아키텍처는 동시성 컴포넌트들 간의 연결관계 및 각 동시성 컴포넌트들이 수행해야 할 동작들을 나타낸다. 동적 아키텍처는 정적 아키텍처의 동시성 컴포넌트들이 어떻게 상호협력 하는지를 나타낸다.

[0096] 520단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처 결정 지원도구(Architectural Decision Supporter)를 사용하여, 적용 가능한 적어도 하나의 아키텍처 패턴을 획득할 수 있다.

[0097] 아키텍처 결정 지원도구는 아키텍처 드라이버 및 사용자의 요구에 따른 아키텍처 드라이버의 우선순위에 따라, 적용 가능한 아키텍처 패턴을 지원한다. 이때, 아키텍처 드라이버의 우선순위와 아키텍처 패턴이 매핑된 결과는 데이터베이스에 저장될 수 있다. 아키텍처 드라이버의 우선순위와 아키텍처 패턴을 매핑시키는 매핑 규칙에 대해서는 도 21에서 상세하게 설명한다.

[0098] 아키텍처 드라이버는 사용자가 요구하는 아키텍처를 구축하는데 결정적인 파라미터를 의미한다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환하는 과정에서 아키텍처를 결정짓는 4가지 파라미터가 설정되었다. 4가지 파라미터는 실행시간/속도(Execution Time/Speed), 메모리 요구(Memory Demand), 메모리 대역폭(Memory Bandwidth) 및 부하 밸런스 필요성(Load Balancing Needs)를 포함한다.

[0099] 사용자로부터 4가지 파라미터에 대한 우선순위가 결정되면, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 우선순위에 매칭되는 적어도 하나의 아키텍처 패턴을 획득할 수 있다. 우선순위에 매칭되는 아키텍처 패턴은 복수 개일 수 있다.

[0100] 530단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 소프트웨어 요구사항 명세서, 상기 참조 아키텍처 및 상기 획득된 적어도 하나의 아키텍처 패턴에 기초하여 멀티코어 기반의 시스템을 위한 아키텍처의 대안을 생성할 수 있다.

[0101] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처 드라이버와 적어도 하나의 아키텍처 패턴간의 상관관계, 상기 적어도 하나의 아키텍처 패턴의 선택에 필요한 체크리스트 및 제약사항에 기초하여 아키텍처의 대안을 생성할 수 있다. 아키텍처 드라이버와 적어도 하나의 아키텍처 패턴간의 상관관계 및 아키텍처 패턴의 선택에 필요한 체크리스트는 아키텍처 결정 지원도구에서 지원될 수 있다. 제약사항은 소프트웨어 요구사항 명세서에서 추출될 수 있다. 아키텍처의 대안은 참조 아키텍처를 구성하는 동시성 컴포넌트들을 중심으로 생성될 수 있다. 아키텍처의 대안은 휴리스틱(heuristic)한 방법으로, 다양한 형태로 생성될 수 있다. 또한, 아키텍처의 대안은 참조 아키텍처와 같이 정적 아키텍처와 동적 아키텍처의 형태로 생성될 수 있다.

[0102] 540단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처의 대안들에 대하여, 트레이드-오프(Trade-off) 분석을 통하여 최적의 아키텍처 대안을 결정할 수 있다.

[0103] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 품질속성 시나리오 중 성능(Performance)관련 시나리오에 기초하여, 우선순위를 반영한 품질속성 트리를 생성할 수 있다. 이때, 품질속성 트리는 품질속성, 품질 고려사항, 품질 속성 시나리오 및 품질 속성 시나리오의 개발 우선순위 순으로 계층이 구분될 수 있다. 품질 고려사항에는 4가지의 아키텍처 드라이버가 반영될 수 있다. 4가지의 아키텍처 드라이버는 실행시간/속도(Execution Time/Speed), 메모리 요구(Memory Demand), 메모리 대역폭(Memory Bandwidth) 및 부하 밸런스 필요성(Load Balancing Needs)를 의미할 수 있다.

[0104] 품질속성 시나리오에 대한 개발 우선순위는 실현난이도(Difficulty)와 중요도(Importance)를 기준으로 결정된다. 중요도는 시스템 성공에 해당 시나리오가 미치는 영향을 의미한다. 우선순위는 실현난이도와 중요도에 대해 High(H), Medium(M), Low(L) 값을 매겨서, 결정된다. 품질속성 시나리오에 대한 개발 우선순위는 미리 설정될 수 있다. 개발 우선순위는 중요도는 높으면서 실현난이도는 낮을수록 높으며, 반대로 중요도는 낮으면서 실현난이도는 높을수록 낮다. [표 4]를 참조하면, 중요도와 실현난이도에 따른 우선순위를 알 수 있다.

표 4

Importance	Difficulty	Priority
H	L	1

H	M	2
H	H	3
M	L	4
M	M	5
M	H	6
L	L	7
L	M	8
L	H	9

- [0106] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 품질속성 트리 상에 부여된 시나리오의 개발 우선순위에 기초하여 트레이드-오프(Trade-Off) 분석을 수행하고, 분석 결과 최적의 아키텍처 대안을 결정할 수 있다.
- [0107] 아키텍처의 트레이드-오프(Trade-Off) 분석은 품질속성 트리 상에 부여된 시나리오의 개발 우선순위를 근거로 이루어진다. 트레이드-오프(Trade-Off) 분석은 아키텍처 대안들의 장점과 단점을 비교하여 수행된다. 이때, 단점에는 해당 아키텍처 대안의 적용시 예상되는 잠재적인 리스크도 포함된다. 품질속성 시나리오에 대한 아키텍처 대안들의 장점과 단점은 미리 설정되어 데이터베이스에 저장될 수 있다.
- [0108] 아키텍처 대안의 장, 단점은 각각 관련된 품질속성 시나리오와 연결될 수 있다. 장점의 경우, 연결된 시나리오를 시스템에 구현하는데 있어서 해당 아키텍처 대안이 (+)요인으로 작용함을 의미한다. 반대로 단점의 경우, 연결된 시나리오를 시스템에 구현하는 데 있어서 (-)요인으로 작용함을 의미한다. 일반적으로, 어떤 아키텍처 대안도 모든 시나리오에 (+)요인으로 작용할 수는 없다. 최적의 아키텍처 대안을 결정하는 과정은 각각 식별된 아키텍처 대안이 가지고 있는 트레이드-오프(Trade-Off) 요인을 분석하여, 결과적으로 (+)요인이 큰 방향으로 작용할 수 있는 한가지를 선택하는 과정이다.
- [0109] 통상적으로 아키텍처 지식 관리 도구가 관리하는 아키텍처 지식(AK: Architectural Knowledge)은 다음의 식과 같이 표현될 수 있다.
- [0110] 아키텍처 지식 = 설계측면의 결정사항(Design Decision) + 설계물(Design)
- [0111] 설계측면의 결정사항은 특정한 산출물을 생성한 근거가 무엇인지를 말하며, 대부분의 근거는 소프트웨어의 개발 요구사항을 출발점으로 삼고 있다. 따라서 아키텍처 지식으로 설계 측면의 결정사항을 관리한다는 것은 특정 아키텍처가 선택되기까지 아키텍처가 참조한 모든 자료와 아키텍처의 머리속에서 연결된 개념들이 무엇인지를 추적한 자료를 관리함을 의미한다. 아키텍처가 참조한 모든 자료와 아키텍처의 머리속에서 연결된 개념들이 무엇인지를 추적한 자료는 아키텍처의 문맥적인 지식(Contextual Knowledge)라고 표현될 수 있다.
- [0112] 학계에서는 2000년대 중반 이후 ADDSS(Architecture Design Decision Support System), ARchium, AREL(Architecture Rationale and Element Linkage), The Knowledge architect, PAKME 등의 실험적인 도구들이 발표되고 있다.
- [0113] 이러한 도구들이 관리하는 지식은 Trade-off, Design Rule, Quality Attribute, architectural design pattern, design tactic 등으로, 다양한 소프트웨어 개발의 부산물 중에서 아키텍처와 관련되어 있는 몇 개의 산출물에 국한되어 있다.
- [0114] 그러나, 실제 아키텍처와 관련된 결정을 내릴 때, 고려하고 참조해야 할 산출물에는 그 범위와 종류를 결정하기 어려운 경우가 많다. 예를 들면, 아키텍처 구축에 필요한 지식은 이미 개발된 레거시 소스 코드와도 관련되어 있을 수 있으며, 이미 개발된 컴포넌트의 테스트와 관련될 수도 있다. 또한, 소프트웨어 시스템이 탑재된 하드웨어의 API 사양서도 아키텍처를 결정하는 데 참고자료가 될 수 있다. 따라서 아키텍처의 문맥적인 지식은 유한개의 템플릿이나 몇 가지의 산출물 형태만으로 한정 지어질 수 없는 성격의 지식이다.
- [0115] 이러한 아키텍처의 문맥적인 지식의 표현 및 관리의 한계를 극복하기 위해, 일 실시예에 따른 아키텍처 지식 관리 도구가 제안된다. 일 실시예에 따른 아키텍처 지식 관리 도구는, 지식의 조각을 담을 수 있는 템플릿으로 OMG(Object Management Group)에서 소프트웨어 자산 명세의 표준으로 제정한 RAS(Reusable Asset Specification)를 지원한다.
- [0116] RAS를 아키텍처 지식 표현의 도구로 이용함으로써, 메타모델을 사용한 아키텍처 지식 표현 방식을 통해 관리 가능한 지식의 유한성 극복할 수 있다. 즉, 소프트웨어 개발과 관련된 모든 형태의 산출물들을 도구 상에서

입력, 수정, 삭제 및 조회가 가능하다.

- [0117] 또한, RAS를 아키텍처 지식 표현의 도구로 이용함으로써, 아키텍처 지식의 이식성(Portability)을 지원할 수 있다. 예를 들면, 아키텍처 관련 산출물 중 UML 모델 형태로 기술된 부분에 대하여, .ras 파일로 Export하여 IBM Rational Software Architect 같은 모델링 도구상에 Import함으로써, 중복 작업 없이 모델링으로 연계할 수 있다.
- [0118] 도 6은 일 실시예에 따른 식별된 컴포넌트와 시스템의 동작을 매핑시킨 도면이다.
- [0119] 도 6을 참조하면, 시스템 행위 다이어그램에서 식별된 동작들은 식별된 컴포넌트 별로 매핑될 수 있다. 여기서, 식별된 컴포넌트는 동시성(Concurrency) 관점에서의 컴포넌트로, 프로세스에 대응되는 개념이다. 즉, 소스코드 상에 함수(Function) 또는 연산(Operation) 형태로 이미 구현된 시스템의 동작들이 하나의 프로세스로 구성된다.
- [0120] 데이터 전처리(Data Preprocessing)(610) 컴포넌트는 동작 1과 매핑된다. 비디오 디코딩 시스템을 예를 들면, 동작 1은 NAL(Network Abstraction Layer) 데이터를 입력 받는 동작, 헤더 타입을 파싱하는 동작을 포함할 수 있다.
- [0121] 엔트로피 디코딩(Entropy Decoding)(620) 컴포넌트는 동작 2 및 동작 3과 매핑된다. 예를 들면, 동작 2는 픽처 타입이 인트라 모드인 경우, 매크로블록의 데이터를 로딩하여 복원에 유리한 화소값을 계산하는 동작을 포함할 수 있다. 동작 3은 픽처 타입이 예측 모드인 경우, 매크로블록의 데이터를 로딩하여 복원에 유리한 화소값을 계산하는 동작을 포함할 수 있다.
- [0122] 변환(Transform)(630) 컴포넌트는 동작 4와 매핑된다. 예를 들면, 동작 4는 원본 매크로블록과 복원을 위해 참조할 매크로블록 간의 오차를 계산하는 동작을 포함할 수 있다.
- [0123] 예측(Prediction)(640) 컴포넌트는 동작 5 및 동작 6과 매핑된다. 예를 들면, 동작 5는 오차값에 대한 인트라 모드 4x4 영상 복원 동작을 포함할 수 있다. 동작 6은 오차값에 대한 예측모드 16x16, 16x8, 8x16, 8x8 영상 복원 동작을 포함할 수 있다.
- [0124] 디블로킹(Deblocking)(650) 컴포넌트는 동작 7 및 동작 8과 매핑된다. 예를 들면, 동작 7은 복원한 매크로블록 영상을 버퍼에 저장하는 동작을 포함할 수 있다. 동작 8은 해당 버퍼가 모두 사용되었는지 및 마지막 프레임인지를 확인하는 동작을 포함할 수 있다.
- [0125] 멀티코어 기반의 시스템은 통상 싱글코어 기반에서의 시스템이 제공하던 기능을 그대로 수행하게 된다. 따라서, 대부분의 싱글코어 기반에서 돌아가던 소스코드들을 멀티코어 기반의 아키텍처에서도 재사용 가능하므로 동시성 컴포넌트(Concurrent Component)와 물리적 컴포넌트(Physical Component)들 간의 매핑 관계를 데이터 베이스에 저장한다. 물리적 컴포넌트는 소스코드 파일(Source Code File)을 의미할 수 있다. 이때, 하나의 소스코드 파일(Source Code File)은 둘 이상의 동시성 컴포넌트의 구성에 참여할 수 있다.
- [0126] 아래 [표 5]는 예시 시스템에서 식별된 동시성 컴포넌트(Concurrent Component)와 물리적 컴포넌트(Physical Component)간의 매핑 관계를 나타낸다. 동시성 컴포넌트는 프로세스로 표현될 수 있다. PredictionFetch.c 파일의 경우 매크로블록 파서(Macro Block Parser)와 디블로커(Deblocker), 두 개의 프로세스에 모두 참여하고 있음을 확인할 수 있다. 예시 시스템의 경우 전반적으로 대량의 데이터가 하나의 파이프라인을 따라 흘러가는 형태의 행위를 보이고 있기 때문에 행위 다이어그램에서 식별되는 데이터 처리 단계(Process Phase)에 따라 프로세스가 구성됨을 확인할 수 있다.

표 5

Process Phase	Process (Filter)	Physical Component (File)
Data Preprocessing	Data Preprocessor	Video_main.c
		DecoderTop.c
Entropy Decoding	Macro Block Parser	EntropyDecMB.c
		EntropyDecMB2.c
		PredictionFetch.c
Transform	Dequantizer	IDCTQ.c
Prediction	Macro Block Predictor	Prediction.c
		PredictionIntra.c
		DecoderTop.c
Deblocking	Deblocker	PredictionFetch.c
		Deblock.c
		DecoderTop.c
Picture Management	Picture Manager	Picmanagement.c

[0127]

도 7은 일 실시예에 따른 참조 아키텍처 중 정적 아키텍처의 일 예를 나타낸 도면이다.

[0128]

싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 동시성 컴포넌트들을 시스템의 동작 순서에 따라 배치하여, 참조 아키텍처를 생성할 수 있다. 정적 아키텍처는 동시성 컴포넌트들 간의 연결관계 및 각 동시성 컴포넌트들이 수행해야할 동작들을 나타낸다.

[0129]

본 예시 시스템은 입력버퍼에 쌓여 있는 프레임들을 순차적으로 프로세싱하여 디코딩한 후 출력버퍼에 쓰는 일련의 작업을 수행하는 비디오 디코더 시스템이다. 따라서, 비디오 디코더 시스템은 기본적으로 파이프 앤 필터(Pipe&Filter) 패턴과 같은 형태의 프로세스 모델을 가진다고 할 수 있다. 다만, 기본적인 파이프 앤 필터(Pipe&Filter) 패턴과 다른 점은 반양자화기(Dequantizer)와 매크로블록 예측기(Macro Block Predictor)의 경우, 서로 수행 순서와 상관없이 디블로커(Deblocker)의 구동 이전에 수행된다는 점이다. 따라서, 반양자화기(Dequantizer)와 매크로블록 예측기(Macro Block Predictor)는 서로 컨커런트(Concurrent)하게 동작될 수 있다.

[0130]

도 7을 참조하면, 정적 아키텍처에는 각 컴포넌트와 컴포넌트 간의 커넥터 및 각 컴포넌트가 수행하는 기능들이 명시되어 있다. 입력 데이터는 데이터 전처리기(Data Preprocessor)(710) 컴포넌트, 매크로블록 파서(Macro Block Parser)(720) 컴포넌트, 반양자화기(Dequantizer)(730) 컴포넌트, 매크로블록 예측기(Macro Block Predictor)(740) 컴포넌트, 디블로커(Deblocker)(750) 컴포넌트 및 픽처 매니저(Picture Manager)(760) 컴포넌트에서 처리된 후, 출력 버퍼에 저장된다.

[0131]

데이터 전처리기(Data Preprocessor)(710) 컴포넌트가 수행하는 동작(711)은 동작 1, 동작 2 및 동작 3을 포함할 수 있다. 동작 1은 NAL 데이터의 헤더 타입 값을 읽어서, 해당 데이터가 SPS, PPS, IDR, Slice 중 어떤 타입에 해당되는지 체크하는 동작을 포함할 수 있다. 동작 2는 타입이 SPS, PPS, IDR인 경우, 해당 데이터를 각각의 자료 구조에 저장하고, 다음 NAL 데이터를 읽는 동작을 포함할 수 있다. 동작 3은 타입이 Slice인 경우, 복원할 데이터를 매크로블록 파서에게 보내는 동작을 포함할 수 있다.

[0132]

매크로블록 파서(Macro Block Parser)(720) 컴포넌트가 수행하는 동작(721)은 동작 4, 동작 5 및 동작 6을 포함할 수 있다. 동작 4는 복원할 영상의 매크로블록을 확인하는 동작을 포함할 수 있다. 동작 5는 복원할 영상이 예측 영상으로, 참조할 영상이 있으면 참조 영상의 위치정보를 모션 벡터로 저장하고, 해당 화소로 분할하는 동작을 포함할 수 있다. 동작 6은 복원할 영상이 인트라 영상으로, 독립적으로 복원이 가능한 데이터라면, 바로 해당 화소로 분할하는 동작을 포함할 수 있다.

[0133]

반양자화기(Dequantizer)(730) 컴포넌트가 수행하는 동작(731)은 동작 7 및 동작 8을 포함할 수 있다. 동작 7은 원본 영상과 참조 영상 간의 오차 값을 계산하는 동작을 포함할 수 있다. 동작 8은 오차를 줄이고 쉽게 압축할 수 있도록 해당 데이터의 표현 범위를 정확한 값으로 변환하는 동작을 포함할 수 있다.

[0134]

매크로블록 예측기(Macro Block Predictor)(740) 컴포넌트가 수행하는 동작(741)은 동작 9를 포함할 수 있다. 동작 9는 원본 영상과 참조 영상간의 오차를 적용하여, 복원할 데이터를 예측하고, 복원하는 동작을 포함할 수 있다.

[0135]

- [0136] 디블로커(Deblocker)(750) 컴포넌트가 수행하는 동작(751)은 동작 10을 포함할 수 있다. 동작 10은 복원한 영상에 더 나은 품질을 제공하기 위해, 해당 영상에 발생한 에러 및 블록 왜곡 현상을 보정하는 동작을 포함할 수 있다.
- [0137] 영상 매니저(Picture Manager)(760) 컴포넌트가 수행하는 동작(761)은 동작 11 및 동작 12를 포함할 수 있다. 동작 11은 디블로킹이 끝난 영상을, 원본 영상의 순서대로 동기화시켜, 버퍼에 저장하는 동작을 포함할 수 있다. 동작 12는 영상 버퍼에 16개의 영상이 모두 차면, 가장 먼저 저장된 영상부터 화면에 출력하는 동작을 포함할 수 있다.
- [0138] 도 8은 일 실시예에 따른 참조 아키텍처 중 동적 아키텍처의 일 예를 나타낸 도면이다.
- [0139] 동적 아키텍처는 시스템의 정적 아키텍처 상에 식별된 각 컴포넌트들이 동적인 관점에서 어떻게 상호협력 하는지를 나타낸다. 표현하는 방식에는 제약이 없으나, 컴포넌트 간의 협력행위가 정확히 표현되어야 한다.
- [0140] 도 8은 도 7에서 식별된 프로세스에 해당되는 필터(Filter)들 간의 상호협력의 모습을 보여준다. 파이프 앤 필터(Pipe&Filter) 패턴의 운영방식 중에서 푸쉬(Push) 방식을 사용하여 필터(Filter)들간의 데이터 전송이 이뤄짐을 확인할 수 있다.
- [0141] 데이터 소스(810)는 푸쉬 방식을 이용하여, NAL 데이터를 데이터 전처리기(820)에게 전송한다. 데이터 전처리기(820)는 푸쉬 방식을 이용하여, 헤더 타입이 슬라이스(Slice)인 경우 복원할 데이터를 매크로블록 파서(830)에게 전송한다. 매크로블록 파서(830)는 푸쉬 방식을 이용하여, 복원할 데이터를 엔트로피 디코딩 한 후, 그 결과 데이터를 다음 단계의 필터에 전송한다. 최종적으로 픽처 관리부(840)는 디블로킹이 끝난 영상을, 원본 영상의 순서대로 동기화할 수 있다.
- [0142] 도 9는 일 실시 예에 따른 패턴 선택 지원자(Pattern Selection Supporter)를 나타낸 도면이다.
- [0143] 아키텍처 결정 지원도구는 아키텍처 패턴의 목록을 출력하는 유저 인터페이스 형태의 패턴 선택 지원자(910)를 포함한다. 패턴 선택 지원자(910)는 데이터베이스에 저장된 아키텍처 드라이버(920) 및 아키텍처 드라이버의 우선순위(930)에 따른 아키텍처 패턴이 매핑된 결과에 기초하여, 아키텍처 패턴의 목록(940)을 출력할 수 있다. 패턴 선택 지원자는 아키텍처 드라이버의 우선순위가 입력되면, 아키텍처 패턴의 목록을 출력할 수 있다.
- [0144] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환과정에서 품질 속성은 성능(Performance)에 맞춰지고, 이에 대응하는 아키텍처 드라이버는 실행시간/속도(Execution Time/Speed), 메모리 요구(Memory Demand), 메모리 대역폭(Memory Bandwidth) 및 부하 밸런스 필요성(Load Balancing Needs)를 포함한다. 각 아키텍처 드라이버에 우선 순위가 High, Medium, Low 중 어느 하나로 설정되면, 적용 가능한 아키텍처 패턴의 목록이 출력될 수 있다.
- [0145] 데이터베이스에는 아키텍처 패턴에 대응하는 패턴 명세서가 저장되어 있다. 패턴 명세서는 패턴의 운영 방식 및 패턴 적용을 위한 체크리스트가 반영될 수 있다.
- [0146] 도 10은 일 실시 예에 따른 시스템 아키텍처의 대안의 일 예를 나타낸 도면이다.
- [0147] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 제약사항, 싱글코어 기반의 참조 아키텍처 및 획득한 아키텍처 패턴에 기초하여 멀티코어 기반의 시스템을 위한 아키텍처의 대안을 생성할 수 있다.
- [0148] 4개의 코어 기반의 비디오 디코딩 시스템을 예로 들면, 제1 아키텍처 대안은 역동적 마스터 앤 워커(Dynamic Master & Worker)로 칭할 수 있다. 역동적 마스터 앤 워커(Dynamic Master & Worker)에서 프로세스 스레드는 마스터(Master) 스레드와 워커(Worker) 스레드의 두 가지 종류로 나뉜다.
- [0149] 마스터 스레드(1010)는 NAL 데이터 파서(NAL Data Parser)를 통하여 입력버퍼에 저장된 NAL 데이터를 읽어와, 각 코어에서 처리할 데이터로 분할하는 프로세스를 수행한다.
- [0150] 워커 스레드(1020, 1030, 1040, 1050)는 프레임 데이터를 싱글코어 시스템을 기반으로 하여 생성된 참조 아키텍처와 같이 디코딩하는 프로세스를 수행한다. 워커 스레드(1020, 1030, 1040, 1050)의 각 코어는 싱글코어

시스템과는 달리 4개의 코어가 동시에 4개의 프레임을 디코딩 한다. 또한, 각 코어는 할당된 비디오 데이터에 대하여 동일한 프로세스 과정을 거쳐 각각 서로 다른 프레임을 디코딩한다.

- [0151] 마스터 스레드(1060)는 동기화기(Synchronizer)를 통하여, 디코딩이 완료된 4개의 프레임들 간의 선후관계를 정리하고, 프레임 버퍼에 저장하는 프로세스를 수행한다. 또한, 픽처 매니저(Picture Manager)를 통하여 프레임 버퍼에 저장된 16개의 프레임을 출력버퍼로 출력하는 프로세스를 수행한다.
- [0152] 도 11은 도 10의 시스템 아키텍처의 대안을 상호협력 다이어그램을 통해 나타낸 도면이다.
- [0153] 도 11을 참조하면, 워커 스레드 중 하나는 마스터 스레드와 동일한 코어에서 동작한다. 획득한 아키텍처 패턴이 리더 앤 팔로어(Leader & Follower) 패턴인 경우에, 상기 워커 스레드와 마스터 스레드가 동작하는 코어를 리더(Leader) 코어라고 할 수 있다.
- [0154] 리더(Leader)코어에 탑재된 마스터 스레드는 각 코어의 워커 스레드 간의 동기화를 담당하는 동기화기(Synchronizer)를 내재하고 있다. 따라서, 원칙적으로 각 코어의 처리가 모두 종료될 때까지 유휴 시간(Idle Time)이 발생하게 된다.
- [0155] 제1 아키텍처 대안은 리더코어의 유휴 시간을 최소화 하기 위해, 동시에 프레임을 처리하고 있는 팔로어코어(Follower)들 중에서 먼저 종료된 코어가 다음 번 리더(Leader)의 기능을 수행하도록 리더코어를 변경시킨다.
- [0156] 코어 1(1110)은 마스터 스레드(1111)와 워커 스레드(1113)가 탑재된 리더코어이다. 코어 2(1120), 코어 3(1130) 및 코어 4(1140)는 워커 스레드가 탑재된 팔로어코어이다. 그러나 코어 2(1120)는 첫 번째 4 프레임 처리시에는 팔로어코어로 동작하지만, 워커 스레드(1113)의 동작이 종료함에 따라, 리더코어인 코어 1(1110)로부터 리더의 역할을 할당(1121) 받는다. 리더코어가 된 코어 2(1120)는 마스터 스레드 중에서 NAL 데이터 파서(NAL Data Parser) 프로세스의 구동을 시작함으로써, 다른 팔로어코어들의 프로세싱이 완료될 때까지 기다리는 유휴시간을 최소화할 수 있다.
- [0157] 도 12A 및 도 12B는 일 실시 예에 따른 시스템 아키텍처 대안의 또 다른 일 예를 나타낸 도면이다.
- [0158] 제2 아키텍처 대안은 싱글코어 기반의 참조 아키텍처와 같이 파이프라인 구조이다. 그러나, 멀티코어 기반으로 전환되는 만큼, 싱글코어에서 동작하던 독립적인 프로세스들은 컴퓨팅 로드(Computing Load)에 따라 4개의 코어에 할당된다. 또한, 각 코어들 간에는 큐 버퍼(Queue Buffer)를 위치하여 파이프 역할을 수행하도록 하였다.
- [0159] 각 코어는 서로 다른 프로세스들을 탑재함에 따라 함수의 병렬성(Functional Parallelism)을 구현할 수 있다. 또한, 파이프라인 구조에서 디코딩의 대상이 되는 프레임 데이터는 순차적으로 처리되므로 별도의 동기화(Synchronize) 과정은 필요하지 않다.
- [0160] 도 12A를 참조하면, 코어 1(1210)에는 데이터 전처리 프로세스 및 매크로 블록 파서 프로세스가 할당된다. 코어 2(1220)에는 반양자화 프로세스가 할당된다. 코어 3(1230)에는 매크로블록 예측 프로세스가 할당된다. 코어 4(1240)에는 디블로킹 프로세스 및 픽처 매니저 프로세스가 할당된다.
- [0161] 도 12B의 경우는, 제2 아키텍처 대안을 각 코어가 수행하는 프로세스 간의 컴퓨팅 로드(Computing Load)의 차이를 고려하여, 변형한 일 예이다.
- [0162] 파이프라인 구조에서는 하나의 프로세스의 성능이 현저하게 떨어질 경우, 멀티코어 시스템으로의 전환에 따른 성능개선 효과가 작을 수 있다. 싱글코어 시스템의 각 프로세스 단계를 분석한 결과, 가장 많은 시간을 소요하는 프로세스는 매크로블록 예측 프로세스이다. 따라서, 매크로블록 예측 프로세스를 두 개의 코어에서 병렬로 수행시킴으로써, 매크로블록 예측 프로세스에 집중되는 컴퓨팅 로드를 분산시킬 수 있다.
- [0163] 코어 1(1250)에는 데이터 전처리 프로세스, 매크로 블록 파서 프로세스 및 반양자화 프로세스가 할당된다. 코어 2(1260) 및 코어 3(1270)에는 매크로블록 예측 프로세스가 할당된다. 코어 4(1280)에는 디블로킹 프로세스 및 픽처 매니저 프로세스가 할당된다.
- [0164] 도 13은 도 12B의 시스템 아키텍처의 대안을 상호협력 다이어그램을 통해 나타낸 도면이다.

- [0165] 도 13은 도 12B의 시스템 아키텍처 대안에서, 각 코어와 코어 내부의 프로세스들 간의 상호협력 관계를 보여주는 시퀀스 다이어그램이다.
- [0166] 코어 1(1310)에는 데이터 전처리 프로세스(1311), 매크로 블록 파서 프로세스(1313) 및 반양자화 프로세스(1315)가 할당된다. 코어 2(1320) 및 코어 3(1330)에는 매크로블록 예측 프로세스가 할당된다. 코어 4(1340)에는 디블로킹 프로세스(1341) 및 픽처 매니저 프로세스(1343)가 할당된다.
- [0167] 코어 2(1320)와 코어 3(1330)의 매크로블록 예측 프로세스가 동시에 수행되기 시작한 후, 두 코어에서 모두 수행되어야만, 코어 4(1340)의 디블로킹 프로세스는 수행될 수 있다.
- [0168] 도 14는 일 실시예에 따른 비디오 디코더 시스템의 품질속성 트리를 나타낸 도면이다.
- [0169] 품질속성 트리는 소프트웨어의 전반적인 품질을 루트(Root) 노드로 삼았을 때, 품질을 측정하는 품질 속성(Quality Attribute)(1410), 품질 고려사항(Quality Concern)(1420) 및 품질 속성 시나리오(Quality Attribute Scenario)(1440) 순으로 계층을 이루어 생성될 수 있다. 또한, 품질 속성 시나리오의 우선순위를 결정하기 위한 중요도(Importance) 및 실현난이도(Difficulty)(1430), 품질 속성 시나리오의 우선순위(1450)가 추가적으로 반영될 수 있다.
- [0170] 이때, 품질 고려사항에 대해서 ATAM 상에서는 어떤 가이드라인도 제공되고 있지 않다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처 패턴을 결정할 때, 주요 입력 값으로 활용되었던 아키텍처 드라이버 4가지(실행시간/속도(Execution Time/Speed), 메모리 요구(Memory Demand), 메모리 대역폭(Memory Bandwidth) 및 부하 밸런싱 필요성(Load Balancing Needs))를 품질 고려사항에 반영하여, 품질 속성 시나리오를 분류하는 기준으로 삼을 수 있다.
- [0171] 품질 속성 시나리오의 우선순위를 결정하기 위한 중요도(Importance) 및 실현난이도(Difficulty)(1430)는 품질 속성 시나리오를 고려하여 미리 설정될 수 있다. 중요도는 시스템 성공에 해당 시나리오가 미치는 영향을 의미한다. 비디오 디코딩 시스템에서 제1 품질속성 시나리오(QAS1)는 '1 코어 시스템에서 4 코어 시스템을 지원하는 시스템으로 변경될 경우, 정상적인 오퍼레이션(Operation)을 수행하는데 소모되는 전력이 1 코어 시스템의 경우와 비교하여 35프로의 Hz 수를 출력하여야 한다'이다.
- [0172] 제2 품질속성 시나리오(QAS2)는 '정상적인 오퍼레이션 하에서 비디오 데이터 디코딩 성능은 기존 1 코어 시스템의 3배 정도인 2.5 데이터 프레임/초를 만족시켜야 한다'이다.
- [0173] 제3 품질속성 시나리오(QAS3)는 '정상적인 오퍼레이션 하에서 비디오 디코더가 지원 가능한 출력 화면의 크기는 기존 1 코어 시스템의 경우와 마찬가지로 1280x720 픽셀까지이다'이다.
- [0174] 제4 품질속성 시나리오(QAS4)는 '정상적인 오퍼레이션을 수행하는데 있어서, 코어 간 커뮤니케이션으로 인해 발생하는 오버헤드 증가분은 10프로 이내여야 한다'이다.
- [0175] 제5 품질속성 시나리오(QAS5)는 '정상적인 오퍼레이션 하에서 비디오 디코더가 출력하는 화면의 화질을 측정하는 비트 레이트는 기존 1 코어 시스템의 경우와 마찬가지로 20Mbps를 지원해야 한다'이다.
- [0176] 제6 품질속성 시나리오(QAS6)는 '정상적인 오퍼레이션을 수행하는데 있어서 메모리 사용량 증가 정도는 70프로 이내여야 한다'이다.
- [0177] 제7 품질속성 시나리오(QAS7)는 '정상적인 오퍼레이션을 수행하는데 있어서, 전체 시스템의 운영 시간 중 각 코어들이 유휴(Idle) 상태에 머무르는 시간은 15프로 이내여야 한다'이다.
- [0178] 품질 속성 시나리오의 우선순위(1450)를 고려해 볼 때, 성능(Performance) 관련 시나리오 중에서는 소모전력의 Hz 수치를 낮추는 것, 즉, 실행 속도 (Execution Speed)를 높이는 것이 가장 높은 우선순위를 가지는 것을 알 수 있다.
- [0179] 각 시나리오의 시스템 반영 시, 우선순위 외에도 품질 속성 시나리오의 순위로부터 아키텍처 드라이버가 미치는 영향 정도를 알 수 있다. 비디오 디코더 시스템의 경우, 실행시간/속도(Execution Time/Speed), 메모리 요구(Memory Demand), 메모리 대역폭(Memory Bandwidth), 부하 밸런싱 필요성(Load Balancing Needs)의 순서임을 알 수 있다.
- [0180] 아키텍처의 트레이드-오프(Trade-Off) 분석은 품질속성 트리 상에 부여된 시나리오의 개발 우선순위를 근거로

이루어진다. [표 6]과 같이, 아키텍처의 트레이드-오프(Trade-Off) 분석은 각 아키텍처 대안에 대하여 생성된 아키텍처 접근법 분석서를 이용할 수 있다. 아키텍처 접근법 분석서에는 제안된 아키텍처 대안의 장점과 단점이 반영되어 있다. 제안된 아키텍처 대안의 장점과 단점이 반영된 아키텍처 접근법 분석서는 데이터베이스에 저장될 수 있다. 또한, 사용자에게 의해 기 설정될 수 있다. 아키텍처 대안의 장, 단점은 각각 관련된 품질속성 시나리오와 연결된다. 장점의 경우는 연결된 시나리오를 시스템에 구현하는데 있어서 해당 아키텍처 대안이 (+)요인으로 작용함을 의미한다. 단점의 경우는 연결된 시나리오를 구현하는 데 있어서 (-)요인으로 작용함을 의미한다. 최적의 아키텍처 결정 과정은 각각 식별된 아키텍처 대안이 가지고 있는 장점 및 단점을 분석하여 결과적으로 (+)요인이 큰 방향으로 작용할 수 있는 한가지를 선택하는 과정이다.

표 6

Architectural Approach 1	Dynamic Master&Worker																								
장점(Pros.)	4개의 독립된 Core에 Decoding Thread를 각각 하나씩 할당함으로써, 싱글 코어와 비교하여 일정한 수준 이상의 데이터 처리 속도 증가를 보장(++QAS1, ++QAS3, +QAS4)																								
단점(Cons.)	1. Dynamic한 Leader&Worker 운영으로 인해 모든 Core상에 거의 대부분의 소스코드가 탑재되어야 하므로, Memory Size가 커짐(- - QAS6) 2. 각 Core에서 처리된 Frame을 출력 Buffer에 쓰기 전에 별도의 Synchronization 작업이 필요함(- QAS1) 3. Master와 Worker 프로세스간의 Data 전송량이 적어도 4Frame 이상(--QAS6)																								
Architectural Approach 2	Pipeline																								
장점(Pros.)	Pipeline 을 따라 흐르는 데이터 전송량은 특정 시점에서 1Frame에 해당하는 량으로 Core간 Data 전송량 감소 효과, Queue Buffer Size 감소효과, 각 Core 상에 싱글코어에서 동작하던 Filter들이 부분적으로 산재되어 있으므로, 하나의 Core에 탑재되는 소스코드 Size 감소(++QAS6)																								
단점(Cons.)	1. 특정 Core의 Computation Time에 전체 시스템의 Execution Time이 의존적일 수 있으며, 특정 Core에서 과부하가 발생할 경우 싱글코어와 비교하여 데이터 처리 속도향상을 기대할 수 없음(--QAS1) 2. Core간의 Communication 횟수가 Approach1에 비해 많음(-QAS4)																								
결론	Approach 1과 Approach2의 장단점과 품질 속성 시나리오, 그리고 각 품질속성에 부여된 개발 우선순위와의 관계를 도식화하면 아래와 같다. <table><tr><th>Quality Attribute Scenario No.</th><th>Architecture Approach1</th><th>Architecture Approach2</th><th>Priority</th></tr><tr><td>QAS1</td><td>+++</td><td>--</td><td>1</td></tr><tr><td>QAS3</td><td>++</td><td>/</td><td>3</td></tr><tr><td>QAS4</td><td>+</td><td>-</td><td>2</td></tr><tr><td>QAS6</td><td>----</td><td>++</td><td>6</td></tr><tr><td>QAS7</td><td>/</td><td>++</td><td>7</td></tr></table> 1. Approach1은 각 Core의 Idle Time을 최소화한 구조로, QAS1, QAS3, QAS4를 지원하는 아키텍처이다. 반면 우선순위 6이 부여된 QAS6에 대해서는 품질을 상당히 저하시키는 경향이 있다. 2. Approach2는 우선순위 6이 부여된 QAS6은 비교적 잘 지원하고 있으나 상대적으로 높은 우선순위를 가지는 QAS1과 AQS4의 품질은 감소시키는 접근법이다. 따라서, 두 가지 아키텍처 접근방법 중에서, 우선순위 1이 부여된 QAS1에서 좋은 성능을 나타내는 Approach 1안을 최적의 대안으로 결정.	Quality Attribute Scenario No.	Architecture Approach1	Architecture Approach2	Priority	QAS1	+++	--	1	QAS3	++	/	3	QAS4	+	-	2	QAS6	----	++	6	QAS7	/	++	7
Quality Attribute Scenario No.	Architecture Approach1	Architecture Approach2	Priority																						
QAS1	+++	--	1																						
QAS3	++	/	3																						
QAS4	+	-	2																						
QAS6	----	++	6																						
QAS7	/	++	7																						

도 15는 일 실시 예에 따른 최적의 아키텍처 대안을 결정하는 방법의 개념을 나타낸 도면이다.

도 15를 참조하면, 임의의 아키텍처 대안 1(1510)과 대안 2(1540)가 있다고 가정한다. 각 대안은 각각 몇 가지의 품질속성 시나리오(QAS)에 플러스 요인으로 작용하는 장점(Pros.)과 마이너스 요인(점선연결)으로 작용하는 단점(Cons.)들을 포함한다. 플러스 요인은 실선 연결로 표시되고, 마이너스 요인은 점선 연결로 표시된다. 최적의 아키텍처 대안을 결정하는 기준은 각 품질속성 시나리오에 부여된 우선순위들이다.

대안 1(1510)의 경우 비교적 높은 우선순위 1, 3, 2를 가지는 QAS 1, 3, 4에는 플러스 요인으로 작용하는 장점(1520)을 가진다. 반면, 비교적 낮은 우선순위 8, 6를 가지는 QAS 2, 7에는 마이너스 요인으로 작용하는 단점(1530)을 가진다.

대안 2(1540)의 경우 낮은 우선순위 8, 7를 가지는 QAS 2, 5에 플러스 요인으로 작용하는 장점(1550)을 가진

다. 반면, 높은 우선순위 1,2를 가지는 QAS 1, 4에는 마이너스 요인으로 작용하는 단점(1560)을 가진다.

- [0186] 두 아키텍처 대안들 중에서 보다 높은 우선순위의 QAS에 대해 장점을 가진 대안 1(1510)이 최종 아키텍처 대안으로 결정된다.
- [0187] 도 16은 일 실시 예에 따른 구축된 아키텍처를 개체화하는 단계(130)를 구체화한 흐름도이다. 개체화는 구축된 아키텍처를 실제로 구현할 수 있도록 컴포넌트를 검색, 결정 및 수정하는 과정을 통해 이루어진다.
- [0188] 1610단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처 결정 지원도구(Architectural Decision Supporter)를 사용하여 컴포넌트를 검색할 수 있다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처 결정 지원도구를 사용하여 소프트웨어 자산 저장소에서, 아키텍처 상의 컴포넌트를 검색할 수 있다. 소프트웨어 자산 저장소에는 소프트웨어 자산이 저장되어 있다. 소프트웨어 자산에는 컴포넌트의 기능, 컴포넌트의 명칭, 컴포넌트의 기능 및 명칭과 매핑된 컴포넌트들이 저장되어 있다. 컴포넌트는 동시성 컴포넌트 및 물리적 컴포넌트를 포함할 수 있다. 물리적 컴포넌트는 소스코드 컴포넌트를 포함할 수 있다. 소프트웨어 자산 저장소는 기 설정되어 데이터베이스에 저장될 수 있다.
- [0189] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처 상의 컴포넌트 중에서 이미 개발된 컴포넌트가 있는지를, 컴포넌트 명칭 또는 컴포넌트의 기능을 키워드로 하여, 상기 아키텍처 결정 지원도구를 통해, 검색할 수 있다.
- [0190] 1620단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 검색된 컴포넌트들 중에서, 상기 구축된 아키텍처에 대응하는 컴포넌트를 결정할 수 있다.
- [0191] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처 결정 지원도구에서 지원되는 참고자료에 기초하여, 상기 검색된 컴포넌트 중에서, 상기 구축된 아키텍처 상에 식별된 컴포넌트의 역할(role)을 수행하는 물리적 컴포넌트를 결정할 수 있다. 참고자료는 소프트웨어 자산 저장소에 저장된 소프트웨어 자산을 의미할 수 있다.
- [0192] 1630단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 결정된 컴포넌트를 멀티코어 기반의 아키텍처에 적합한 형태로 수정할 수 있다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 결정된 소스코드 컴포넌트를 멀티코어 아키텍처에 적합한 형태로 수정할 수 있다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 이미 기존에 개발되어, 소프트웨어 자산 저장소에 저장되어 있는 소스코드 컴포넌트들을 재사용함으로써, 전반적인 개발 공정의 비용을 줄일 수 있다.
- [0193] 도 17 내지 도 19는 일 실시 예에 따른 소프트웨어 자산을 나타낸 도면이다.
- [0194] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처 결정 지원도구를 통하여 아키텍처 상의 컴포넌트 중에서 이미 개발된 것이 있는지를 소프트웨어 자산 저장소에 접근하여 검색할 수 있다.
- [0195] 검색 방법은 찾고자 하는 기능이나 컴포넌트 명칭 등을 키워드로 하는 키워드 검색 방식을 따른다. 도 17은 Prediction과 관련된 컴포넌트의 검색을 위해 아키텍처 결정 지원도구를 이용한 소프트웨어 자산 검색 화면의 모습을 보여준다.
- [0196] 도 18은 검색된 컴포넌트 중 하나의 컴포넌트에 대한 상세정보를 나타낸다. 도 19는 검색된 컴포넌트 중 하나의 컴포넌트가 속한 소프트웨어 자산(Asset)의 전체 구조를 나타낸다. 컴포넌트에 대한 상세정보 및 컴포넌트가 속한 소프트웨어 자산(Asset)의 전체 구조는 구축된 아키텍처 상에서, 식별된 컴포넌트의 역할(role)을 수행하는, 컴포넌트 결정시의 참고자료들이다.
- [0197] 도 20은 일 실시 예에 따른 구축된 아키텍처를 정량적으로 검증하는 단계(140)를 구체화한 흐름도이다.
- [0198] 2010단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 구축된 아키텍처를 검증하기 위한 프로토타입을 생성할 수 있다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 구축된 소프트웨어 아키텍처 및 싱글코어 기반으로 개발된 코드에 기초하여, 정량적으로 성능 측정이 가능한 형태의 시스템 프로토타입을 생성할 수 있다.

- [0199] 구축된 아키텍처의 정량적인 검증을 위해, 실제 시스템의 일부 기능이 반영된 프로토타입을 이용한다. 멀티코어의 개수에 비례하여 시스템의 성능이 증가하는 것을 지원하는 아키텍처가 구축되었는지를 검증하는 것이 프로토타입 생성의 목적이다. 프로토타입 생성시에 싱글코어에서는 없던 기능이 추가적으로 반영되는 것은 지양한다.
- [0200] 예시 시스템은 비디오 디코더 시스템으로, 압축된 비디오 데이터를 디코딩하는 단일한 기능만을 가지고 있다. 따라서, 비디오 디코딩 기능을 수행하는 프로토타입을 이용한다. 여러 가지 기능과 서비스를 가지고 있는 소프트웨어 제품의 경우에도, 아키텍처 검증을 위한 프로토타입의 범위는 최소화하여 주요 기능 1-2가지에 국한시키는 것이 좋다. 프로토타입을 통한 정량적인 검증은 주로 멀티코어로의 전환 이전 대비 이후의 성능 향상 정도를 측정하는 것이 목적이므로, 대상이 되는 기능은 성능(performance) 관점에서 큰 오버헤드가 예상되는 기능을 반영하는 것이 의미 있다.
- [0201] 2020단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 프로토타입에 기초하여 성능 인자(factor)별로 상기 구축된 아키텍처를 검증할 수 있다.
- [0202] 성능 인자는 반응을 측정하는 기준(Response Measure), 검증 결과(Evaluation Results) 및 수용범위(Acceptance Range)를 포함할 수 있다.
- [0203] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 시스템 프로토타입에 대하여, 품질속성 시나리오 중 반응을 측정하는 기준(Response Measure)을 근거로, 검증 결과(Evaluation Results)를 측정할 수 있다. 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 수용범위(Acceptance Range)에 기초하여, 상기 구축된 소프트웨어 아키텍처를 정량적으로 검증할 수 있다.
- [0204] 품질 속성 시나리오의 항목 중에서 반응양식을 측정하는 기준(Response Measure)으로 선택된 아이템들은, 아키텍처의 정량적 검증 시, 성능 측정 항목인 Metric이 된다. 시스템에 따라 반응양식을 측정하는 기준(Response Measure) 항목은 서로 상이할 수 있으나, 동일한 도메인에 속하는 소프트웨어 제품들은 서로 비슷한 반응양식을 측정하는 기준(Response Measure)을 가지게 된다. 멀티코어 시스템의 성능 목표치는 주로 싱글코어 성능에 대비한 비율로 정의하는 경우가 대부분이다. 성능 분석 결과에는 기존의 싱글코어 시스템의 성능치와 비교된 결과가 표시된다. 품질 속성 시나리오는 반응양식을 측정하는 기준(Response Measure)에 명시된 단위에 따라 시스템의 성능의 기대치가 수치로 명시되어 있다. 성능의 기대치는 수용범위(Acceptance Range) 값으로 표시될 수 있다. 프로토타입의 실행 결과 측정되는 성능 값들은 검증 결과(Evaluation Results) 항목에 해당 한다. 비교 분석 결과(Enhancement)에는 싱글코어 대비 개선된 성능이 비율로 표시될 수 있다. [표 7]은 예시 시스템의 성능 분석 결과를 보여주고 있다.

표 7

[0205]

Metric	Single Core	Acceptance Range	Evaluation Results	Enhancement
System 동작 Hz수	1148MHz	싱글코어 시스템의 35%	334MHz	싱글코어의 29%
사용된 Memory 용량	29827KB	증가분이 싱글코어 시스템의 70%	47548KB	싱글코어 대비 59% 증가
Core들의 Idle Time 비율	0%	전체 수행시간의 15%	13%	멀티코어 관리에 전체 성능의 13%의 오버헤드 발생
처리 Data Frames/Second	Second 0.87 fps	2.5 Data Frames/Second	2.99 fps	싱글코어의 3.4배의 처리속도 향상
출력 Pixel 수	1280 X 720	1280 X 720	1280 X 720	싱글코어와 동일한 Pixel 사이즈 지원
출력 Bit Rate	20M bps	20M bps	20M bps	싱글코어와 동일한 화질 유지

[0206]

표 7을 참조하면, 4개의 코어 기반으로 구축된 아키텍처를 구현한 비디오 디코더의 경우, 싱글코어 시스템의 출력화면의 사이즈와 화질을 그대로 유지하면서 싱글코어와 비교하여 약 3.4배 정도의 처리속도 향상이 있다. 각 코어의 유휴 시간(Idle Time)이 0일 경우, 처리속도의 향상은 4배를 기대할 수 있다. 그러나 싱글코어 시스템과 비교하여 코어간의 커뮤니케이션(Communication) 및 동기화(Synchronization) 등으로 인해 발생한 각 코어의 유휴 시간이 약 13% 정도 발생함에 따라, 이상적인 수치에서 0.6배 정도의 감소가 발생한 것으로 분석

된다. 품질 속성 시나리오 중 가장 우선순위가 높았던 시스템 동작의 Hz 수치는 싱글코어의 29% 수준에서 비디오 디코딩이 수행됨에 따라, 목표 수치로 잡았던 싱글코어 시스템의 35%보다 더 낮은 값으로 기대치보다 더 좋은 결과를 보였다. 따라서, 예시 시스템에서 아키텍처 대안 중 최적의 대안으로 선택되었던 역동적 리더 앤 팔로어(Dynamic Leader & Followers)를 적용한 아키텍처는 정량적인 검증 면에서도 타당함이 입증되었다.

- [0207] 도 21은 일 실시 예에 따른 싱글 코어에서 멀티 코어 기반으로의 멀티미디어 아키텍처 패턴 결정 방법의 흐름도이다.
- [0208] 싱글 코어에서 멀티 코어 기반으로의 멀티미디어 아키텍처 패턴 결정 방법은 동시성(Concurrency) 확보 문제와 관련된 패턴들 중에서 주어진 시스템에 적합한 최소한의 아키텍처 패턴 세트를 결정하는 방법을 제공한다. 현재까지 발표된 문헌상의 모든 동시성(Concurrency) 관련 패턴을 수집하여, 멀티미디어 어플리케이션의 경우 아키텍처 드라이버로 정의될 수 있는 각 요소들과의 직접적인 매핑 규칙을 정의하였다. 별도의 소프트웨어 아키텍처 관련 지식이 없는 경우에도 매핑 규칙에 의해, 시스템 상황과 관련된 아키텍처 패턴을 획득함으로써 아키텍처 구축에 소모되는 시간 및 비용을 줄일 수 있다.
- [0209] 2110단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 아키텍처 드라이버들을 입력 받는다. 아키텍처 드라이버들은 실행시간/속도(Execution Time/Speed), 메모리 요구(Memory Demand), 메모리 대역폭(Memory Bandwidth) 및 부하 밸런스 필요성(Load Balancing Needs)을 포함한다.
- [0210] 2120단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항에 기초하여 상기 아키텍처 드라이버들의 우선순위를 결정한다. 비기능적 요구사항에서 품질속성 중 성능에 초점을 둔다면, 실행시간/속도(Execution Time/Speed)의 우선순위는 High, 메모리 요구(Memory Demand)의 우선순위는 Low, 메모리 대역폭(Memory Bandwidth)의 우선순위는 Low 및 부하 밸런스 필요성(Load Balancing Needs)의 우선순위는 High로 결정될 수 있다. 또한, 아키텍처 드라이버의 우선순위는 사용자에 의해 임의로 결정될 수도 있다.
- [0211] 2130단계에서, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 결정된 아키텍처 드라이버들의 우선순위에 기초하여, 매핑 테이블에서 적어도 하나의 아키텍처 패턴을 결정할 수 있다. 매핑 테이블은 소프트웨어 자산 저장소 및 데이터베이스에 저장될 수 있다.
- [0212] 상기 매핑 테이블은 동시성 관련 아키텍처 패턴들과 상기 아키텍처 드라이버들을 포함하고, 상기 동시성 관련 아키텍처 패턴과 상기 아키텍처 드라이버들의 우선순위는 체크 리스트에 기초하여 서로 매핑될 수 있다.
- [0213] 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 결정된 아키텍처 드라이버들의 우선순위와 일치하는 아키텍처 패턴을 추천 패턴(Recommended Pattern)으로 결정할 수 있다. 또한, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 상기 결정된 아키텍처 드라이버들의 우선순위와 부분적으로 일치하는 아키텍처 패턴을 부분적용 가능 패턴(Partially Adaptable Pattern)으로 결정할 수 있다.

표 8

Category	Architecture Pattern	N-Core Multimedia Applicability (Y/N)	Execution Time/Speed			Memory Demand			Memory Bandwidth			Load Balancing Needs		
			H	N	L	H	N	L	H	N	L	H	N	L
Performance	Leader / Followers	Y	●					●			●	●		
	Map Reduce	Y	●			●			●			●		
	Pipes / Filters	Y		●			●		●					●
	Produce / Consumer	Y	●				●		●					●
	Scheduler	Y		●			●		●				●	
	Acceptor / Connector	N		●			●		●				●	
	Active Object	N	●				●		●				●	
	Asynchronous Completion Token	N	●				●		●				●	
	Asynchronous Processing	N	●				●		●			●		
	Double-Checked Locking Optimization	N	●				●		●				●	
	Event-based Asynchronous	N	●				●		●				●	
	Event-based, implicit invocation	N			●		●		●			●		
	Half-Sync / Half-Async	N	●			●			●			●		
	Iterative Refinement	N	●			●			●				●	
	Monitor Object	N			●	●					●		●	
	Proactor	N	●				●		●				●	
	Reactor	N			●		●		●				●	
	Read/Write Lock	N	●				●		●				●	
	Scoped Locking	N			●		●		●				●	
	Strategized Locking	N	●				●		●				●	
	Thread Pool	N	●					●	●			●		
	Thread-Safe Interface	N			●		●		●				●	
	Thread-Specific Storage	N	●					●	●				●	

[0214]

[0215]

[표 8]은 각 아키텍처 패턴들이 4개의 아키텍처 드라이버 측면에서 봤을 때 어떤 경우에 활용 가능한지를 표기한 매핑 테이블이다. 예를 들어, 예시 시스템의 아키텍처 구축시, 적용되었던 리더 앤 팔로어(Leader/Followers) 패턴은 실행 시간(Execution Time)에 대한 우선순위는 높고(High: H), 메모리 요구(Memory Demand)량과 메모리 대역폭(Memory Bandwidth)에 대한 우선순위는 낮으며(Low: L), 부하 밸런싱 필요(Load Balancing Need)에 대한 우선순위는 높은(H) 특성을 가지는 시스템에 적용 가능함을 [표 8]을 통해 파악할 수 있다. 모든 아키텍처 패턴이 4가지 아키텍처 드라이버와 모두 관련되는 것은 아니다. Acceptor/Connector 패턴은 4개의 아키텍처 드라이버와는 트레이드-오프(trade-off) 관계가 없기 때문에, 특정 패턴이 해당 멀티코어 아키텍처 드라이버의 값을 결정짓는 것과는 무관함을 의미하는 'Neutral(N)' 값을 적용하였다. [표 8]의 각 아키텍처 패턴과 4개의 아키텍처 드라이버 값의 매핑 근거는 [표 9]와 [표 10]에 기재되어 있다. [표 9]는 성능(performance) 품질속성과 관련된 아키텍처 패턴 중 멀티코어 시스템에 적용 가능한 패턴들의 매핑 근거 및 체크리스트를 나타낸다. [표 10]은 성능(performance) 품질속성과 관련된 일반적인 패턴에 대한 매핑 근거 및 체크리스트를 보여주고 있다.

표 9

[0216]

No	Pattern Name	Architectural Driver Value & Rationales	Check List
1	Leader / Followers	Execution Speed(H) Thread간의 Locking overhead 감소, Context switching이 필요 없어 latency 감소 Memory Demand(L) 각 프로세서 상에는 서로 다른 기능을 가지는 Event Handler들이 탑재되어 각자의 역할을 수행할 수 있음. 따라서 서로 다른 Handler가 각 프로세스에 구현 가능 Memory Bandwidth(L) 대량의 데이터 스트림을 처리하기 위해 고려된 부분은 없음 Load Balancing Needs(H) 패턴 구조에 정의된 Thread Pool은 정확하고 효율적인 thread의 조정을 보장하며, 구현되는 전략에 따라 Load를 분산시킬 수 있음	* 다양한 자원들로부터 발생하는 요청들을 처리하여야 하는가? * 요청된 이벤트를 처리하기 위하여 context switching 등의 performance overhead 가 빈번하게 발생하는가? * 요청된 이벤트를 처리하기 위한 thread들 간의 Locality 밀집도는 높은 편인가? * 비교적 Network I/O의 양은 적은 편인가?
2	Map Reduce	Execution Speed(H) 대용량의 데이터를 효율적으로 병렬 처리할 때 주로 사용되는 패턴 Memory Demand(H) 대용량의 데이터가 Map으로 분산된 후 Reduce를 거치면서 여러 번 복사되기 때문에 필요한 Memory Size는 커짐 Memory Bandwidth(H) 큰 사이즈의 데이터들이 입출력을 한번에 거침 Load Balancing Needs(H) Master의 구현 전략에 따라 Load Balancing을 고려한 구현이 가능함	* 대용량 데이터에 대한 분산 처리 및 병렬 처리 작업을 필요로 하는 도메인인가? * 해당 시스템이 처리할 데이터를 분산시키고 그에 따른 load balancing을 필요로 하는 도메인인가? * 어플리케이션이 처리하는 작업이 Batch 성격을 띠는 작업인가? * 처리할 데이터가 하나의 Key 와 Value로 표현 가능한 데이터인가? * 대량의 데이터를 처리할 프로세스의 추가 및 삭제가 빈번하게 발생할 것으로 예상되는 어플리케이션인가? (Salability) * 해결하고자 하는 문제가 병렬처리, fault-tolerance, load balancing, locality 최적화 등과 관련되어 있는가?
3	Pipes / Filters	Execution Speed (N) 대량의 데이터 흐름에 대해 각각의 독립적인 필터들이 병렬적으로 수행됨으로써 수행시간은 감소되나 특정 필터의 프로세싱 타임이 오래 걸리거나, 모든 입력을 사용하여야 출력 값을 계산할 수 있는 필터가 존재할 경우, 프로세싱 타임의 향상이 없을 수도 있다. Memory Demand (N) 각 필터에 배치되는 소스코드의 양에 따라 달라질 수 있으며, 상당한 크기의 전역 데이터를 필터간에 공유해야 할 경우 많은 메모리를 소요 Memory Bandwidth (H) Batch Sequential 구조와 같이 각 필터에 대량의 데이터가 한꺼번에 유입될 경우 Memory Bandwidth 증가 Load Balancing Needs (L) Pipeline으로 이미 구성된 흐름상에서 데이터들은 일괄적으로 흘러가는 형태	* 데이터 스트림 처리와 관련된 어플리케이션인가? * 물리적으로 서로 다른 프로세서에 서로 다른 processing 작업을 할당할 필요가 있는가? * 인접하지 않은 컴포넌트들 간의 정보 공유가 필요 없는 어플리케이션인가? * 해당 도메인 시스템을 서로 다른 processing 단계를 변경 및 재조합 할 수 있는가?
4	Produce / Consumer	Execution Speed(H) Task 생성자와 처리자의 분리로 처리속도 가속화 가능 Memory Demand(N) Queue 사이즈나 Producer와 Consumer로 실행되는 소스코드의 사이즈에 따라 달라짐 Memory Bandwidth(N) 패턴사용에 따라 좌우되지 않음 Load Balancing Needs(L) Load는 Producer와 Consumer의 비율에 따라 결정되는 것으로 패턴 사용만으로 Load Balancing 향상을 기대하기는 어려움	* 데이터를 만들고, 만들어진 데이터를 사용하는 형태인가? * 데이터를 사용하는 것보다 만드는 것이 빠르나? * 데이터를 만드는 작업과 데이터를 사용하는 작업이 비동기적인가? * 특정 Task 를 생성하는 주체와 처리하는 주체가 분리되어 있는가? * Task의 생성 요청에 대해 전체 프로세스가 비동기적으로 동작하는가? * Task 생성 요청 수가 Task 처리하는 건수보다 항상 많은 것이 문제가 되는가?

5	Scheduler	Execution Speed(N) Scheduler의 Scheduling Algorithm의 효율성에 달려 있음 Memory Demand(N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth(N) 패턴사용에 따라 좌우되지 않음 Load Balancing(N) 패턴사용에 따라 좌우되지 않음	* 동시에 많은 작업이 요청될 가능성이 있는 어플리케이션인가? * 요청된 작업은 순서대로 수행되어야 할 필요가 있는 어플리케이션인가?
---	-----------	--	--

표 10

No	Pattern Name	Architectural Driver Value & Rationales	Check List
1	Acceptor / Connector	Execution Speed(N) Connection 연결을 위한 추가적인 step이 발생하여 수행시간을 늘릴 수 있다. Memory Demand(N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth (N) 패턴사용에 따라 좌우되지 않음 Load Balancing(N) 패턴사용에 따라 좌우되지 않음	* 해당 시스템은 run-time behavior를 지원하기 위해 다양한 connection roles을 필요로 하는가? * 요청된 서비스에 대한 connection을 수립 및 초기화하는 작업(configuration code)과 해당 서비스를 processing하는 작업(service processing code)이 혼재되어 있는 시스템인가? * Connection 수립 및 초기화 작업으로 인한 latency 발생으로 성능 저하를 발생시킬 가능성이 있는가? * Asynchronous Connection이 요구되는 어플리케이션인가?
2	Active Object	Execution Speed(H) Processing-intensive Method의 호출과 실행 시점을 Async로 운영하면서 Client는 다른 작업을 수행 가능함에 따라 수행 시간 감소 Memory Demand(N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth(N) 패턴사용에 따라 좌우되지 않음 Load Balancing(N) 패턴사용에 따라 좌우되지 않음	* 여러 client 들이 공유 자원에 동시에 접근하는 정도가 빈번하게 발생하는가? * Invocation과 execution time이 명확하게 구분 가능하고, 각각의 실행 순서는 중요하게 다뤄지지 않는 시스템인가? * Processing-intensive 한 method가 많이 존재하는가? * 시스템이 제공할 서비스의 requester가 여러 개이고, 요청된 서비스를 해결할 thread 역시 여러 개 존재하는가? * 요청을 처리하는 스케줄링에 대한 필요성이 높은 어플리케이션인가?
3	Asynchronous Completion Token	Execution Speed(H) service에 의해 반환된 데이터에 대하여 복잡한 parsing 작업을 필요로 하지 않기 때문에, 시간면에서 감소효과 발생 Memory Demand(N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth(N) 패턴사용에 따라 좌우되지 않음 Load Balancing (N) 패턴사용에 따라 좌우되지 않음	* 요청된 operation에 대한 작업 결과와 client 어플리케이션에게 비동기적으로 출력되는가? * Client와 서비스 간에 completion event에 대한 demultiplexing 작업으로 인하여 communication overhead가 발생하는가? * Service Initiator가 service 결과값에 대한 Parsing에 많은 시간이 소요되는 어플리케이션인가?
4	Asynchronous Processing	Execution Speed(H) 일반적인 Synchronous Processing 보다는 High-Speed Memory Demand(N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth(N) 패턴사용에 따라 좌우되지 않음 Load Balancing(H) Asynchronous processing을 시작하는 시점을 조절할 수 있으므로 load balancing이 가능	* 연속된 두 개 이상의 작업의 호출 순서와 실행 순서가 반드시 일치할 필요가 없는 어플리케이션인가?

5	Double-Checked Locking Optimization	Execution Speed(H) 일반 lock 보다는 해당 응용에서 좀더 효율적으로 동작함 Memory Demand(N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth(N) 패턴사용에 따라 좌우되지 않음 Load Balancing(N) 패턴사용에 따라 좌우되지 않음	* 특정 데이터를 유일하게 관리해야 하는가? * 특정 데이터에 대한 locking이 필요한가? 그렇다면 그 빈도가 잦은가? * 각 프로세서에서 cache memory를 사용해 데이터에 접근하는가?
6	Event-based Asynchronous	Execution Speed (H) 대용량의 I/O작업과 다른 Processing을 병렬 처리함에 따라 전반적인 실행 시간 감소 가능 Memory Demand (N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth (H) 대용량의 I/O 작업을 효율적으로 병행하기 위한 면이 고려됨에 따라 Bandwidth 향상효과 기대 가능 Load Balancing (N) 패턴사용에 따라 좌우되지 않음	* 시간이 많이 소모되는 작업을 background에서 수행하도록 하고, 다른 작업 수행하는 형태가 많은 어플리케이션인가?
7	Event-based, implicit invocation	Execution Speed (L) 해당 컴포넌트의 수행시간을 Guarantee할 수 없음 Memory Demand (N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth (N) 패턴사용에 따라 좌우되지 않음 Load Balancing (H) 분산화되어 구현된 Neutral 시스템에서 Load Balancing을 보장함	* 해당 시스템을 구성하는 컴포넌트들은 객체 간의 events를 사용하여 통신하는가? * 이벤트를 dispatch 하고, listener를 등록하는 과정에 많은 시간이 소모되는가? * 특정 이벤트의 수행을 위한 관련 프로시저들이 서로 다른 위치에 존재하는가? * 객체 간의 통신 과정이 복잡하게 이뤄지는 어플리케이션인가?
8	Half-Sync / Half-Async	Execution Speed (H) Layer 간의 communication이 single point(Message Queue)로 단순화됨에 따라 처리 속도 향상 Memory Demand (H) Queuing Layer가 필요함에 따라 Queue 사이즈 만큼의 메모리 추가 불가피 Memory Bandwidth (H) 시간소비적인 네트워크 I/O 작업등을 Async하게 처리함에 따라 전반적인 I/O량 증가 Load Balancing (H) Queuing Layer가 Sync/Async 층에 대한 완충작용 담당	* 어플리케이션 구동 시 asynchronous processing 과 synchronous processing이 혼재되어 있는가? * Asynchronous processing과 synchronous processing의 각각의 성능에 현저한 불균형이 존재하는가? * 명확한 계층화(layering)가 가능한 구조인가? * Boundary-crossing penalty 문제가 성능에 critical한 영향을 미치는가? * Multiplexing 의 양이 많은 어플리케이션인가? * 어플리케이션 구현을 위하여 네트워크 처리계층의 behavior와 로직 처리계층의 behavior를 명확히 분리한 라이브러리 구축이 가능한 도메인인가?
9	Iterative Refinement	Execution Speed (H) 하나의 Processing이 내부적으로 여러 개의 iteration으로 나뉘고 병렬 수행 가능할 경우, 수행속도 증가 Memory Demand (N) iteration 숫자에 비례하여 중간 산출물 저장을 위한 추가적인 메모리 필요 Memory Bandwidth (N) 패턴사용에 따라 좌우되지 않음 Load Balancing (N) 패턴사용에 따라 좌우되지 않음	* 한 processing 단계 내부의 연산 작업들이 많은 병렬적인 tasks로 나뉘지는 어플리케이션인가? * 특정 기능을 달성하기 위하여, 상당한 iterations 과정을 필요로 하는 어플리케이션인가? * Iterations 연산 수행으로 인한 synchronization 비용을 고려하지 않아도 되는 어플리케이션인가?

10	Monitor Object	Execution Speed (L) Monitor Lock 을 사용할 경우, 멀티코어 프로세싱이 Locking이 걸린 순간 마다 싱글코어 프로세싱과 같이 동작하게 됨. Race Condition이 발생하지 않는 application에서는 Execution Speed 저하이유가 될 수 있음 Memory Demand (H) 공유되는 데이터가 중복되지 않아 메모리 사용량은 적음 Memory Bandwidth (L) 공유 데이터가 한 곳에만 존재하고 이를 모든 core들이 access하게 됨에 따라 bandwidth는 증가될 수 있음 Load Balancing (N) 패턴사용에 따라 좌우되지 않음	* 병렬적으로 요청된 프로세스들의 concurrency 유지가 중요한 어플리케이션인가? * 해당 어플리케이션에 대한 유지보수가 빈번하게 발생하는가? * Concurrency 관리에 관련된 이슈가 performance 향상과 관련된 이슈보다 더욱 중요한 도메인인가? * Concurrency를 달성하기 위한 프로세스들이 간단한 메커니즘 등을 통해 수행될 수 있어야 하는가?
11	Proactor	Execution Speed(H) 해당 events를 처리하는 logical한 threads제어만을 활성화시킴으로써context switching에 소모되는 시간 감소 Memory Demand(N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth(H) socket handle을 통해 asynchronously하게 데이터를 읽고 쓰는 작업함으로써 Throughput 증가 Load Balancing(N) 패턴사용에 따라 좌우되지 않음	* 다양한 client로부터 들어오는 서비스 요청을 비동기적으로 처리해야 하는 시스템인가? * Server는 동시에 여러 개의 client 요청을 수행해야 하는가? * 다양한 요청을 처리하는데 발생하는 context switching, synchronization, data movement 등과 같은 overhead가 문제가 되는 어플리케이션인가? * 동시에 발생하는 많은 사용자들의 요청을 처리하기 어렵고, 사용자가 요청한 long-duration request를 처리하기 위해서는 많은 시간이 걸리는 어플리케이션인가? * I/O 와 같은 long-duration operation과 일반적인 processing operation이 상당히 혼재되어 있는 어플리케이션인가?
12	Reactor	Execution Speed(L) 병렬적으로 발생하는 Event들을 효율적으로 Serialize시키고 있으나, 멀티코어 관점에서는 결과적으로 성능 저하되는 결과 Memory Demand(N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth (N) 패턴사용에 따라 좌우되지 않음 Load Balancing(N) 패턴사용에 따라 좌우되지 않음	* 동시에 수신된 서비스 요청들을 실제로 동기적으로 처리할 필요가 있는가? * 처리되어야 할 서비스에 대한 multiplexing, demultiplexing, dispatching 할 필요가 있는 시스템인가? * 다양한 client의 수만큼 많은 thread를 추가적으로 생성함으로써(Thread-Per-Connection), 심각한 성능저하를 발생시키는 시스템인가?
13	Read/Write Lock	Execution Speed(H) 일방 lock 보다는 해당 응용에 효율적으로 동작함 Memory Demand (N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth (N) 패턴사용에 따라 좌우되지 않음 Load Balancing (N) 패턴사용에 따라 좌우되지 않음	* 동일한 메모리에 대해 읽고 쓰는 작업을 동시에 해야 하는가? * Read/Write Thread가 충분히 많이 발생하는 어플리케이션인가? * 데이터에 대한 안정성 보장이 중요시 되는 어플리케이션인가?
14	Scoped Locking	Execution Speed(L) Lock 을 사용할 경우, 멀티코어 프로세싱이 Locking이 걸린 순간 마다 싱글코어 프로세싱과 같이 동작하게 됨. Race Condition이 발생하지 않는 application에서는 Execution Speed 저하이유가 될 수 있음 Memory Demand(N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth(N) 패턴사용에 따라 좌우되지 않음 Load Balancing(N) 패턴사용에 따라 좌우되지 않음	* C++ 언어를 통해 개발하는가? (본 사항은 패턴이라기 보다는 C++에 사용하는 idiom입니다. 내용을 보시면, 따로 garbage collector가 존재하는 java 등의 언어에서는 사용이 불가합니다.) * 특정 데이터를 유일하게 관리해야 하는가? * 특정 데이터가 메모리에 상주할 수 있을 정도로 작은 크기 인가? * 특정 데이터에 접근하는 코드의 범위가 명확한가?

15	Strategized Locking	Execution Speed(H) Lock 을 사용할 경우 일반적인 방식보다는 좀 더 응용에 효율적인 lock을 사용할 수 있음 Memory Demand(N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth(N) 패턴사용에 따라 좌우되지 않음 Load Balancing(N) 패턴사용에 따라 좌우되지 않음	* 사용하고자 하는 여러 개의 Locking Mechanism이 하나의 인터페이스로 표현 가능한가? * 여러 개의 Locking Mechanism이 Component에 적용 가능하고, 필요에 따라 특정한 하나의 Mechanism이 수행되는 형태인가? * 컴포넌트는 특정 Synchronization Mechanism으로부터 독립적인가?
16	Thread Pool	Execution Speed (H) Logical Concurrency를 지원하는 패턴 Memory Demand (L) thread별로 공유 데이터를 복사해 줘야 하기 때문에 메모리 사용량이 증가한다. Memory Bandwidth (N) 패턴사용에 따라 좌우되지 않음 Load Balancing (H) Thread Pool 사용 자체로 Load Balancing 가능	* 수행해야 하는 작업에 비해 하드웨어 자원이 충분하지 않은가? * 동일한 요청을 동시에 처리해야 하는가?
17	Thread-Safe Interface	Execution Speed (L) 불필요한 Lock의 획득, 해제를 최소화하고 있으나 잠재적인 Overhead 요소가 존재하며, 멀티코어 운영 관점에서 어느 순간에는 Serializ e되는 시점이 존재하므로 실행시간 감소 효과 미미해 보임 Memory Demand (N) 패턴사용에 따라 좌우되지 않음 Memory Bandwidth (N) 패턴사용에 따라 좌우되지 않음 Load Balancing (N) 패턴사용에 따라 좌우되지 않음	* 특정 데이터에 대해서 반드시 하나의 Thread만 접근하도록 관리해야 하는가? * 함수 내부에서 반복적으로 locking이 필요한 데이터에 접근하는가? * Locking Overhead를 최소화하는 것이 상위 우선순위를 가지는 요구사항인가? * Self-deadlock의 발생을 방지하는 것이 중요한 어플리케이션인가?
18	Thread-Specific Storage	Execution Speed (H) Thread 고유의 데이터를 유지함으로써 Global Data 로의 접근을 최소화 --> Locking Overhead 최소화 Memory Demand (L) 쓰레드 별로 local storage를 할당하므로 메모리 사용량은 증가 Memory Bandwidth (H) Global data에 대한 access가 local data로 바뀌어지면 bandwidth의 요구량은 줄어듦 Load Balancing (N) Thread Pool 사용 자체가 Load Balancing 전략을 가지진 않음	* 동일한 전역 데이터에 대한 각 thread들에 대해, 접근이 빈번한가? * 공유되는 전역 데이터에 대해 각 thread가 Local에서 유지해야 할, 상태 정보를 명확히 정의할 수 있는가? * 전역 데이터에 대한 starvation이나 deadlocking이 자주 발생하는 어플리케이션인가?

[0218] [표 9]의 근거를 기반으로, [표 8]에 수록된 아키텍처 패턴 별 아키텍처 드라이버 값에 관한 정보는 지원도구인 아키텍처 결정 지원도구(Architectural Decision Supporter)에 탑재되어 참고자료로 제공될 수 있다.

[0219] [표 8]의 정보를 통해 검색된 패턴은 두 가지 그룹으로 나뉘어 제공된다. 첫 번째는 주어진 시스템의 4가지 아키텍처 드라이버 속성값과 정확히 일치하는 정보를 가지고 있는 패턴으로, 추천패턴(Recommended Pattern)으로 분류된다. 두 번째는 4가지 아키텍처 드라이버 속성값과 부분적인 일치를 보이는 패턴으로, 부분적용 가능 패턴(Partially Adaptable Pattern)으로 분류된다. 리더 앤 팔로워(Leader/Followers) 패턴과 스레드 풀(Thread Pool) 패턴은 4개의 아키텍처 드라이버 값에 모두 부합되는 추천패턴이며, 나머지 아키텍처 드라이버 중 부분적으로 관련되어 있는 패턴들은 부분적용 가능 패턴으로 분류될 수 있다.

[0220] 도 22는 일 실시 예에 따른 멀티미디어 아키텍처 패턴을 결정하는 일 예를 나타낸 도면이다.

[0221] 예시 시스템인 비디오 디코더 시스템에서 싱글코어에서 멀티코어로 전환하면서 기대되는 요구사항(2210)에 의하면 4가지의 아키텍처 드라이버(2220)를 필요로 한다. 4가지의 아키텍처 드라이버(2220)는 실행시간/속도(Execution Time/Speed), 메모리 요구(Memory Demand), 메모리 대역폭(Memory Bandwidth) 및 부하 밸런스 필

요성(Load Balancing Needs)를 포함한다.

- [0222] 4가지 아키텍처 드라이버에 대해 우선순위가 결정되고, [표 8]의 매핑 테이블에 기초하여 검색된 동시성(Concurrency) 관련 아키텍처 패턴(2230)은 총 12개다. 그 중 리더 앤 팔로어(Leader/Follower) 패턴과 스레드 풀(Thread Pool) 패턴은 네 개의 아키텍처 드라이버 값과 모두 관련되어 있으므로 추천패턴(Recommended Pattern)으로 결정될 수 있다. 나머지 하나 이상의 아키텍처 드라이버 값과 상관관계를 가지는 나머지 9개의 패턴들은 부분적용 가능 패턴(Partially Adaptable Pattern)으로 결정될 수 있다.
- [0223] 도 23은 일 실시 예에 따른 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치의 블록도이다.
- [0224] 도 23을 참조하면, 일 실시 예에 따른 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환 장치는 입력부(2310), 추출부(2320), 제1 처리부(2330), 제2 처리부(2340), 출력부(2350) 및 저장부(2360)를 포함한다.
- [0225] 입력부(2310)는 싱글코어 기반의 레거시 소스 코드를 입력 받는다. 싱글코어 기반 레거시 시스템은 싱글코어를 위해 생성된 일반적인 시스템을 의미할 수 있다. 예를 들면, H.264 비디오 디코더(decoder) 시스템을 의미한다. 싱글코어 기반의 레거시 소스 코드는 싱글코어 기반 소프트웨어에서 프로세서를 통해 추출되고, 추출된 싱글코어 기반의 레거시 소스 코드는 입력부(2310)에 입력될 수 있다.
- [0226] 추출부(2320)는 상기 싱글 코어 기반의 레거시 소스 코드에 기초하여, 멀티 코어 기반으로의 전환을 위한, 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항을 추출한다.
- [0227] 추출부(2320)는 레거시 시스템의 소스코드를 함수설명(Function Description), 자료구조(Data Structure) 및 호출관계(Call Relationship)에 따라 분류함으로써, 소프트웨어 아키텍처의 기능적 요구사항을 추출할 수 있다.
- [0228] 추출부(2320)는 데이터베이스에 저장된, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환과 관련된 품질속성에 기초하여, 소프트웨어 아키텍처의 비기능적 요구사항을 추출할 수 있다. 예를 들면, 싱글 코어에서 멀티 코어 기반으로의 아키텍처 전환과 관련된 품질속성은 성능(performance)일 수 있다.
- [0229] 제1 처리부(2330)는 상기 기능적 요구사항 및 비기능적 요구사항을 고려하여 아키텍처의 대안들을 생성하고, 상기 아키텍처의 대안들 중에서, 품질속성에 기초한 적어도 하나의 최적의 아키텍처를 결정한다.
- [0230] 제1 처리부(2330)는 아키텍처 드라이버와 적어도 하나의 아키텍처 패턴간의 상관관계, 상기 적어도 하나의 아키텍처 패턴의 선택에 필요한 체크리스트 및 제약사항에 기초하여 아키텍처의 대안을 생성할 수 있다.
- [0231] 제1 처리부(2330)는 동시성 관련 아키텍처 패턴들과 아키텍처 드라이버들이 매핑된 매핑 테이블에 기초하여 아키텍처 대안을 생성할 수 있다.
- [0232] 제1 처리부(2330)는 아키텍처 대안들의 트레이드-오프(Trade-off) 분석을 통하여 최적의 아키텍처를 결정할 수 있다.
- [0233] 제2 처리부(2340)는 상기 적어도 하나의 최적의 아키텍처에 대응하는 컴포넌트를 결정하여, 상기 적어도 하나의 최적의 아키텍처를 개체화(Instantiation)하고, 상기 적어도 하나의 최적의 아키텍처에 대하여, 멀티 코어 개수에 비례하여 시스템의 성능이 개선되는지를 정량적으로 검증(Evaluation)한다.
- [0234] 제2 처리부(2340)는 아키텍처 결정 지원도구(Architectural Decision Supporter)를 이용하여 컴포넌트를 검색할 수 있다. 제2 처리부(2340)는 아키텍처 결정 지원도구를 통해 소프트웨어 자산 저장소에서, 아키텍처 상의 컴포넌트를 검색할 수 있다. 소프트웨어 자산에는 컴포넌트의 기능, 컴포넌트의 명칭, 컴포넌트의 기능 및 명칭과 매핑된 컴포넌트들이 저장되어 있다.
- [0235] 제2 처리부(2340)는 아키텍처 결정 지원도구에서 지원되는 참고자료에 기초하여, 상기 검색된 컴포넌트 중에서, 상기 구축된 아키텍처 상에 식별된 컴포넌트의 역할(role)을 수행하는 물리적 컴포넌트를 결정할 수 있다. 참고자료는 컴포넌트에 대한 상세정보 및 컴포넌트가 속한 소프트웨어 자산(Asset)을 포함할 수 있다.
- [0236] 제2 처리부(2340)는 싱글코어 기반으로 개발된 코드에 기초하여 정량적으로 성능 측정이 가능한 형태로 생성된, 시스템 프로토타입을 이용하여, 구축된 아키텍처를 정량적인 검증할 수 있다.
- [0237] 출력부(2350)는 상기 정량적으로 검증된 상기 적어도 하나의 최적의 아키텍처를 출력한다.
- [0238] 저장부(2360)는 소프트웨어 아키텍처(architecture)의 기능적 요구사항 및 비기능적 요구사항, 품질속성, 아

키텍처의 대안들 및 아키텍처 결정 지원도구를 저장한다. 또한, 저장부(2360)는 소프트웨어 자산 및 트레이드-오프 분석에 필요한 아키텍처의 장점 및 단점을 저장할 수 있다.

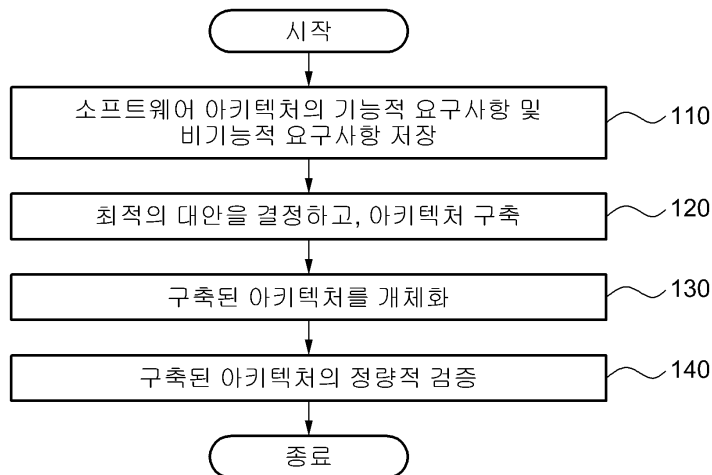
[0239] 위에서 설명된 실시 예에 따른 방법들은 다양한 컴퓨터 수단을 통하여 수행될 수 있는 프로그램 명령 형태로 구현되어 컴퓨터 판독 가능 매체에 기록될 수 있다. 상기 컴퓨터 판독 가능 매체는 프로그램 명령, 데이터 파일, 데이터 구조 등을 단독으로 또는 조합하여 포함할 수 있다. 상기 매체에 기록되는 프로그램 명령은 본 발명을 위하여 특별히 설계되고 구성된 것들이거나 컴퓨터 소프트웨어 당업자에게 공지되어 사용 가능한 것일 수도 있다.

[0240] 이상과 같이 본 발명은 비록 한정된 실시예와 도면에 의해 설명되었으나, 본 발명은 상기의 실시예에 한정되는 것은 아니며, 본 발명이 속하는 분야에서 통상의 지식을 가진 자라면 이러한 기재로부터 다양한 수정 및 변형이 가능하다.

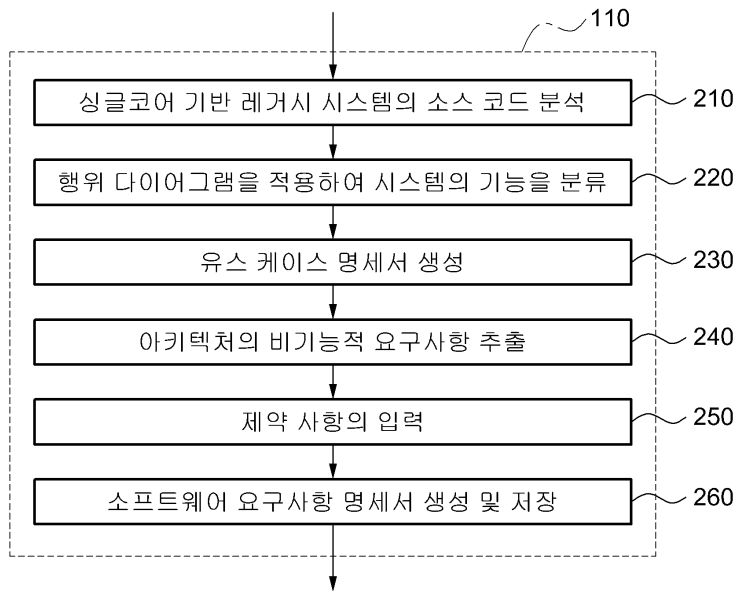
[0241] 그러므로, 본 발명의 범위는 설명된 실시예에 국한되어 정해져서는 아니 되며, 후술하는 특허청구범위뿐 아니라 이 특허청구범위와 균등한 것들에 의해 정해져야 한다.

도면

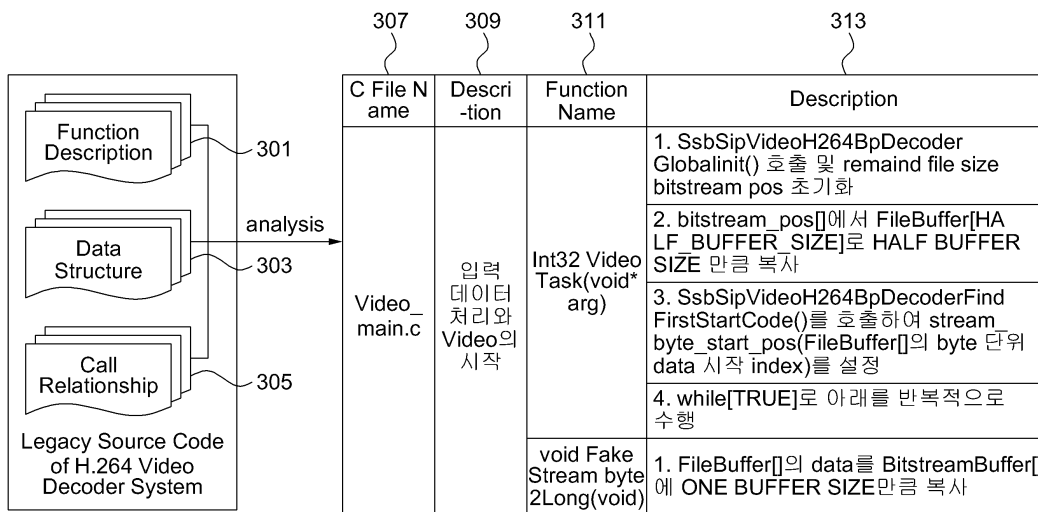
도면1



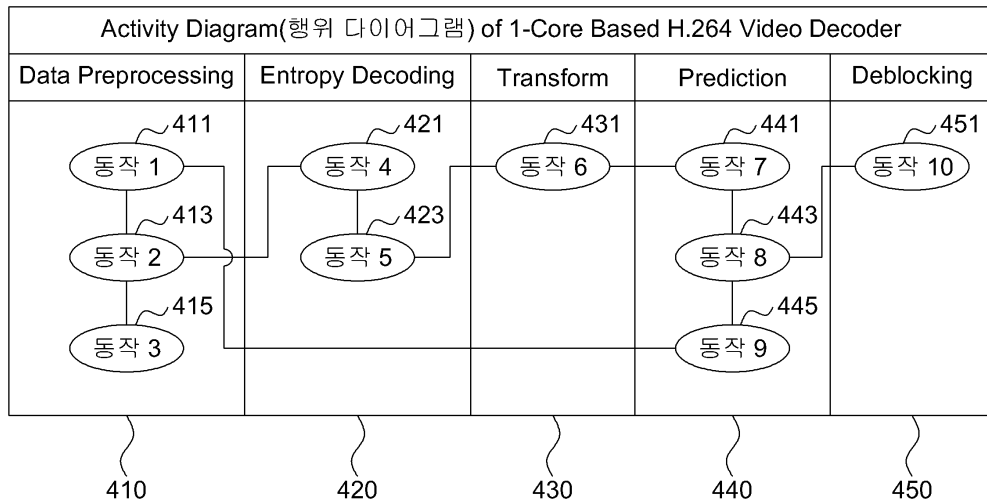
도면2



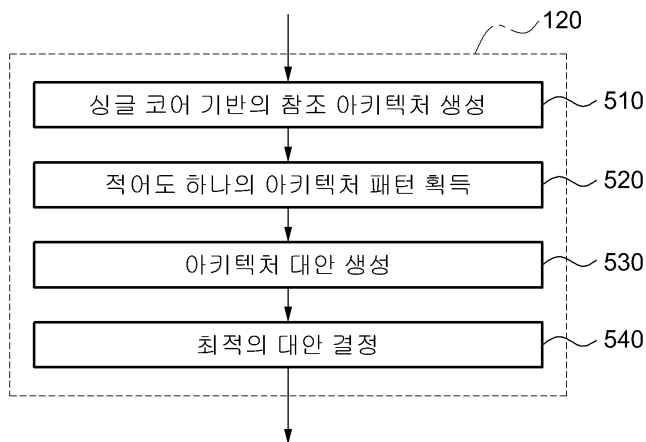
도면3



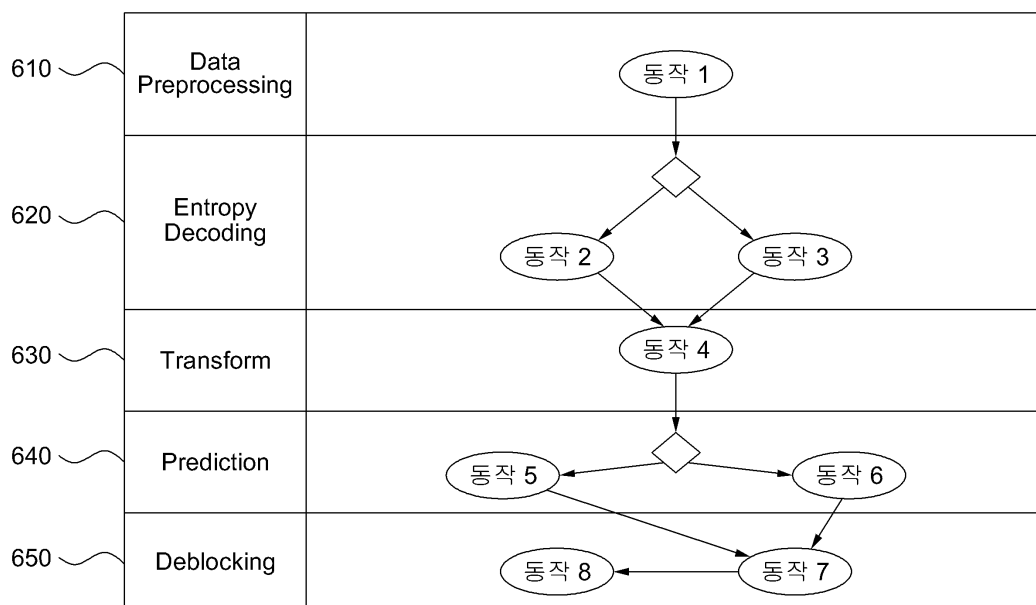
도면4



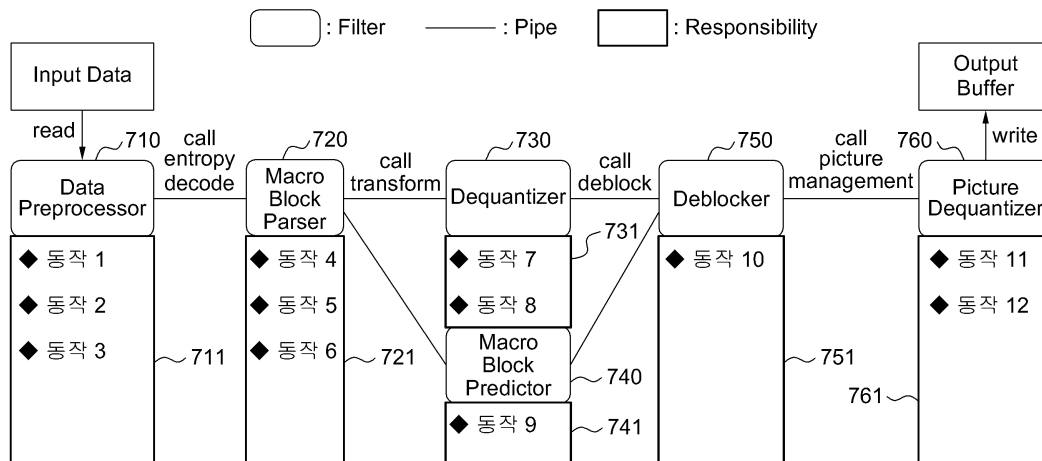
도면5



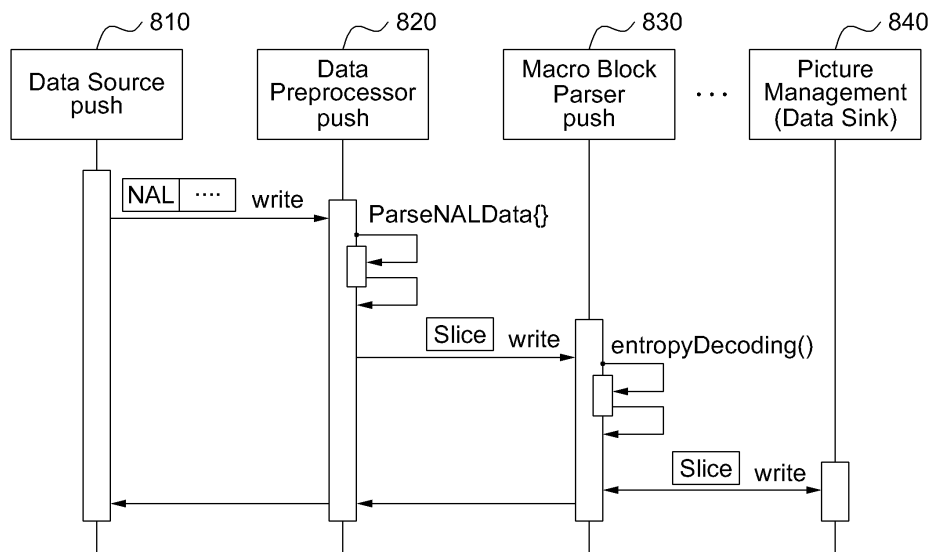
도면6



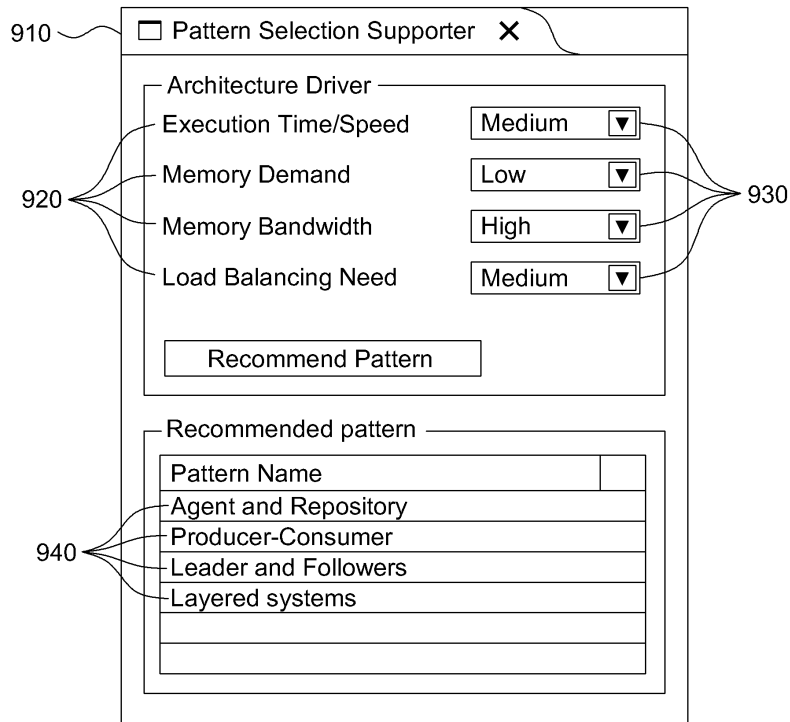
도면7



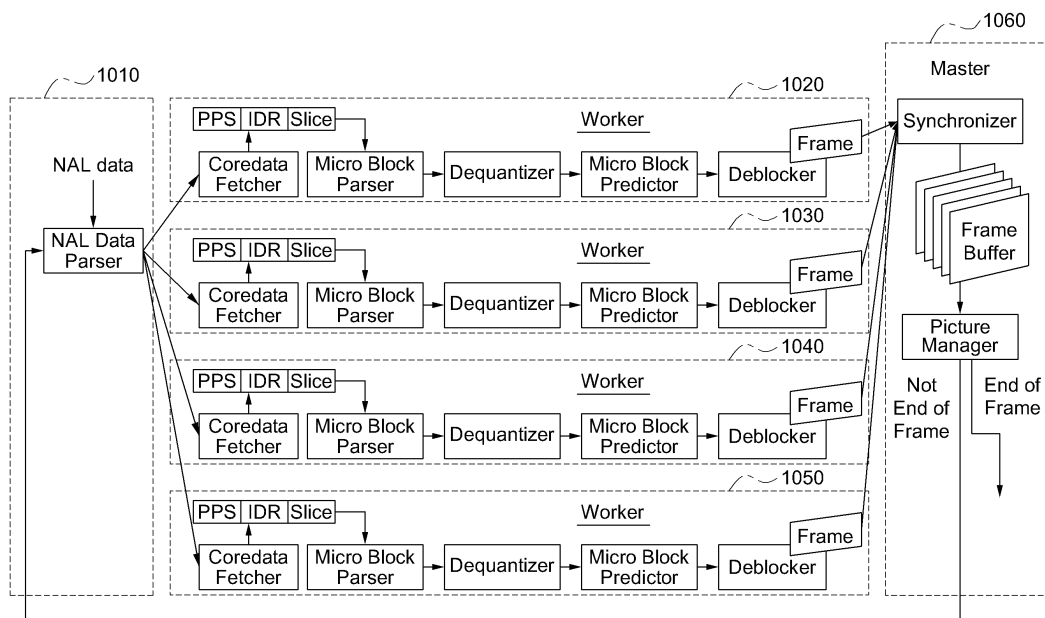
도면8



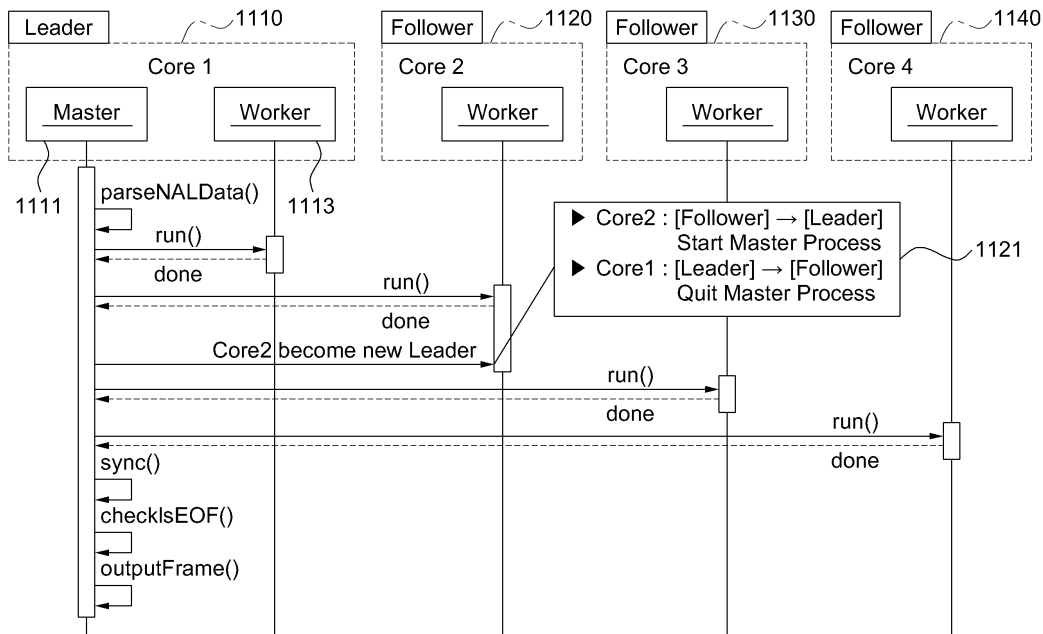
도면9



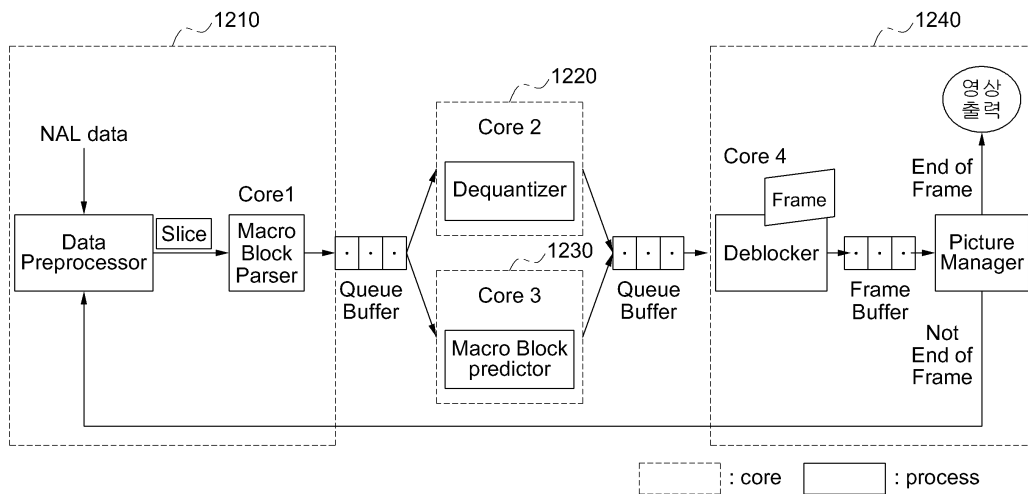
도면10



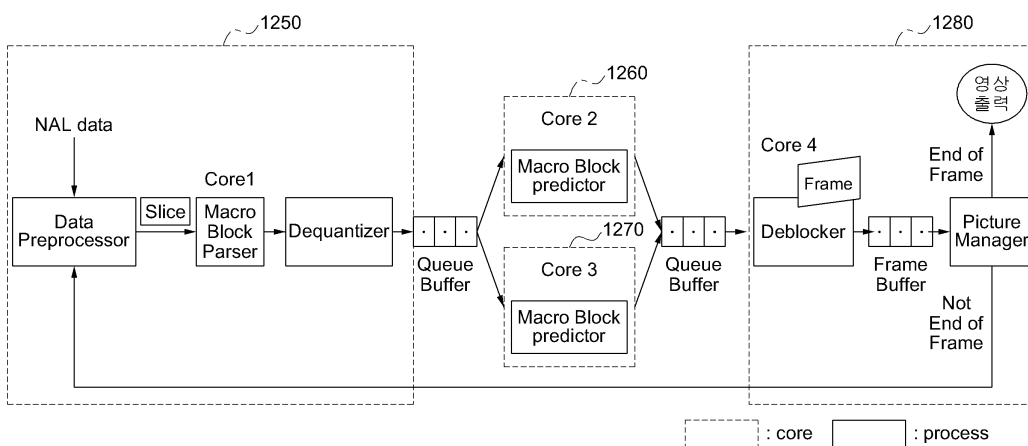
도면11



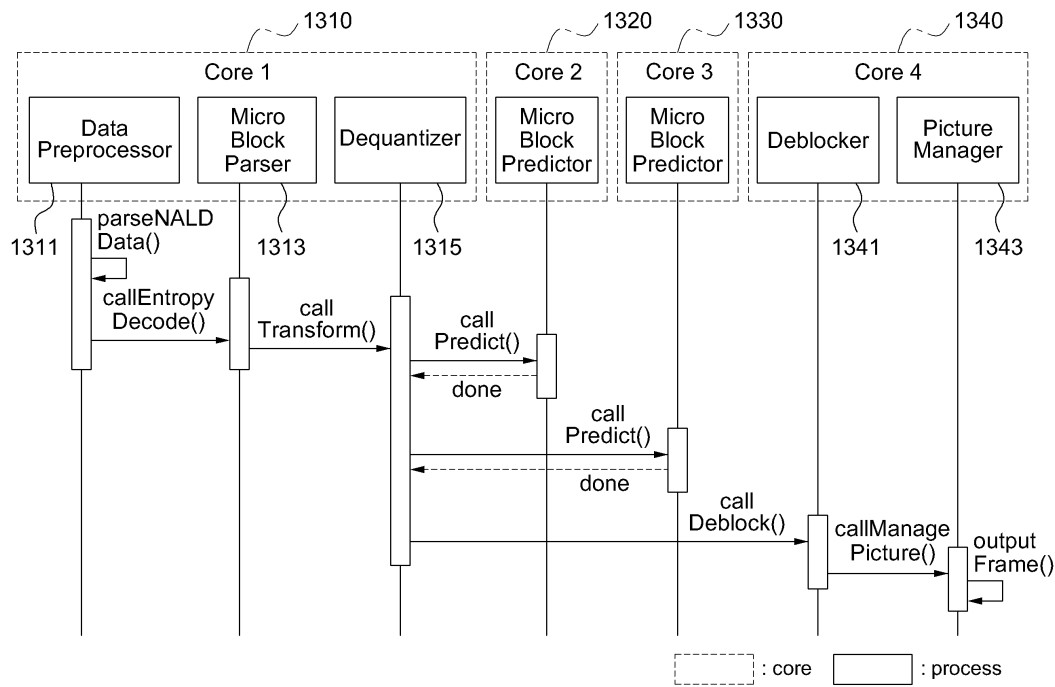
도면12a



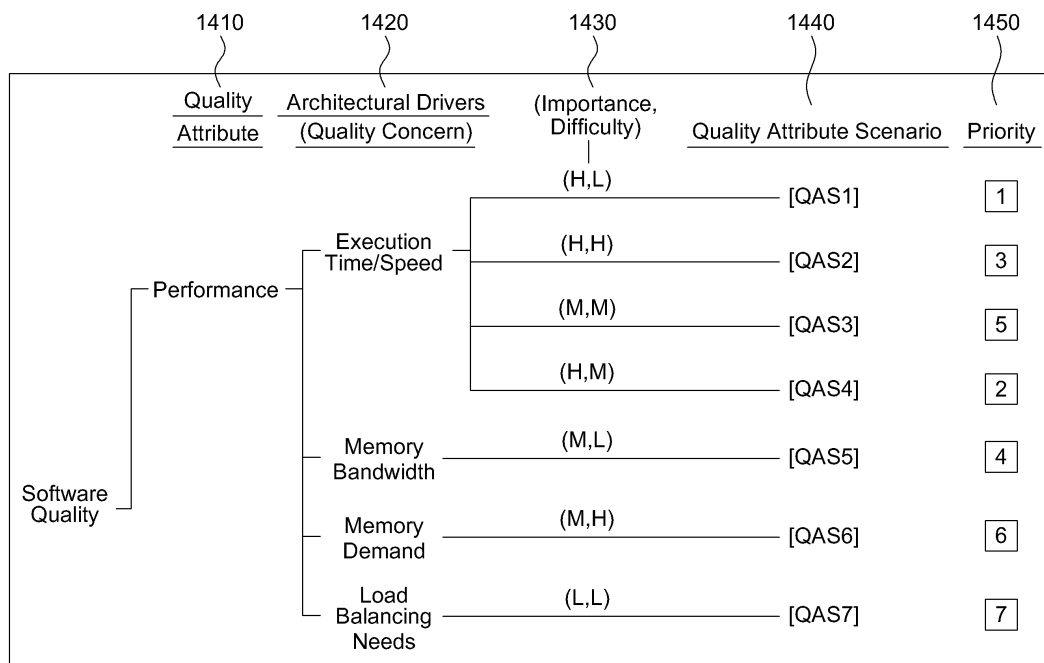
도면12b



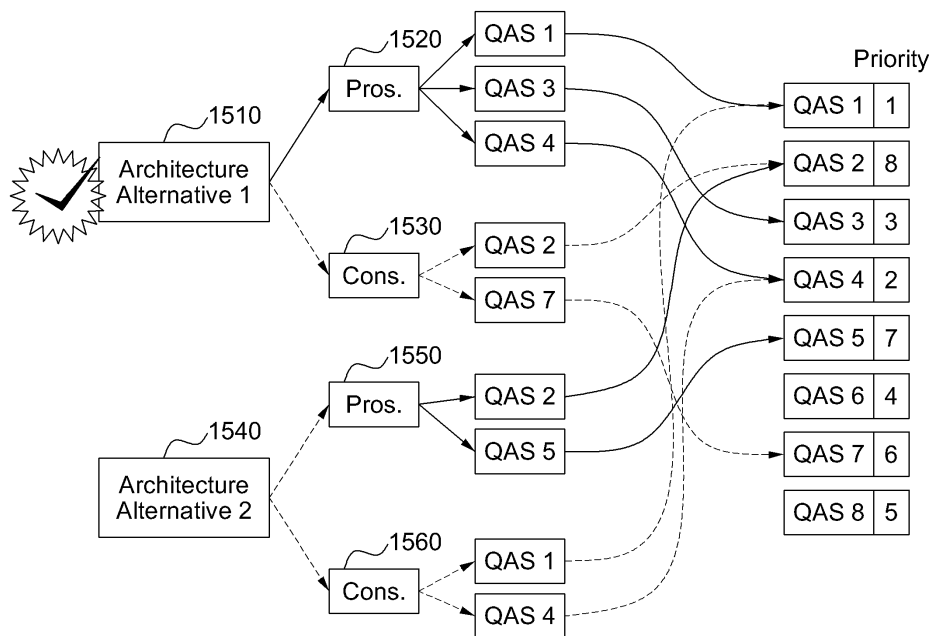
도면13



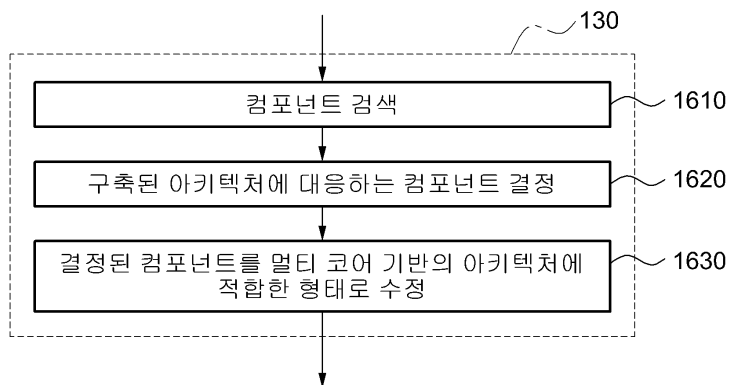
도면14



도면15



도면16



도면17

Search X

Condition of Search

Keyword

Prediction

Search

Result of Search

Keyword

Prediction

5 Searched

No.	Content	Type
1	SsLibH264BpPredictionFetch	Artifact.Name
2	SsLibH264BpPrediction	Artifact.Name
3	SsLibH264BpPredictionIntra	Artifact.Name
4	Prediction 블록은 VLD 블록...	Artifact.Description
5	SsLibH264BpPredictionMain	Artifact.Name

도면18

Artifact Editor X

Properties of Artifact

Name

SsLibH264BpPrediction

Type

Source Code

Description of Artifact

I=Macro block 의 타입 값을 저장한다.

Reference

None

ID

251185426

Version

0.1

Rights

everyone

Contexts

Context

-Select Context-

Add

-Select Related Artifacts-

Add

No	Context Name	Context ID
1	Video	2750203657

No	Artifact ID	Artifact Name

Variability Point Properties

Name

ID

1713185775

VP

Context

-Select Context-

Reference

Add

Activity

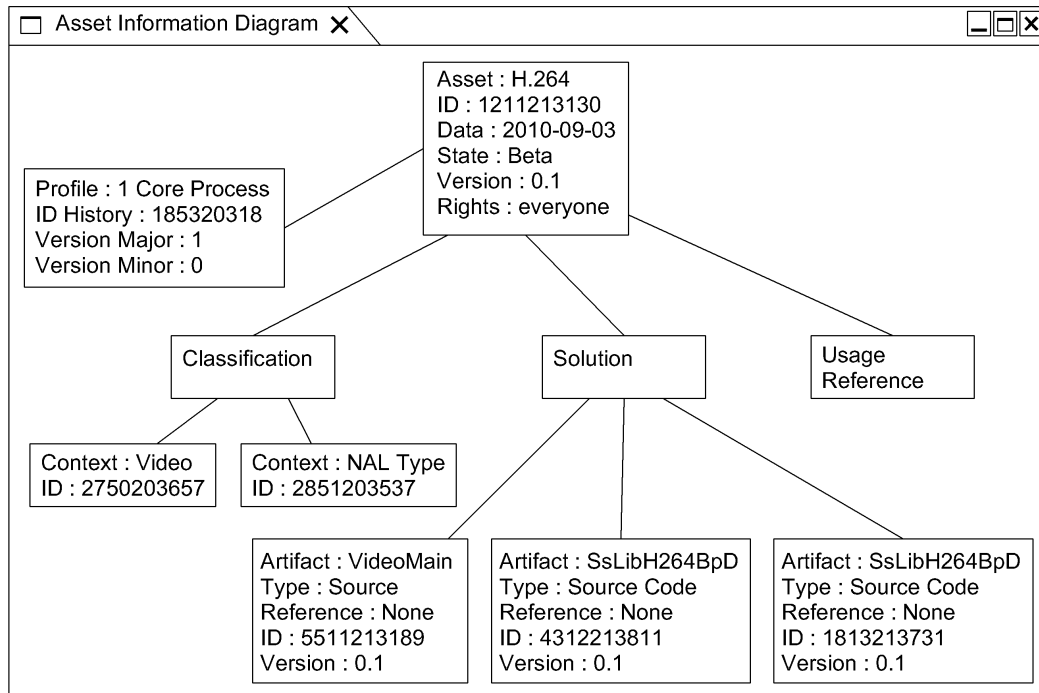
-Select Activity-

Create

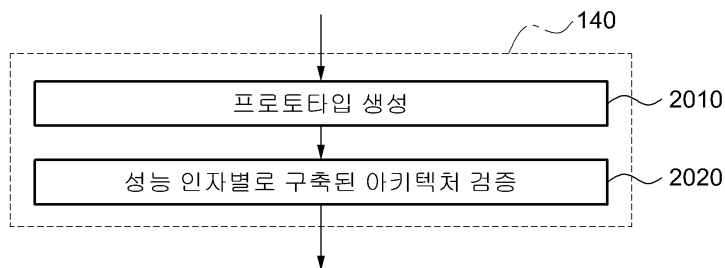
Update

Delete

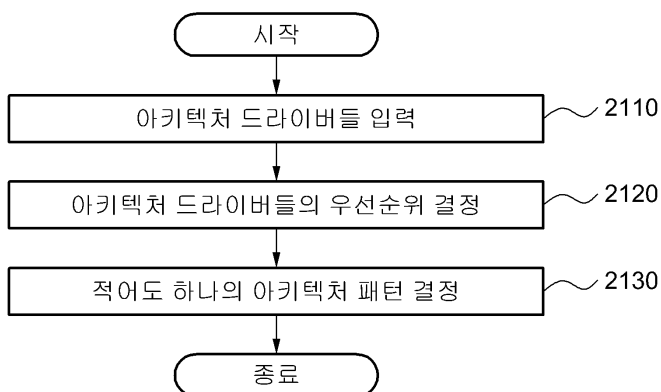
도면19



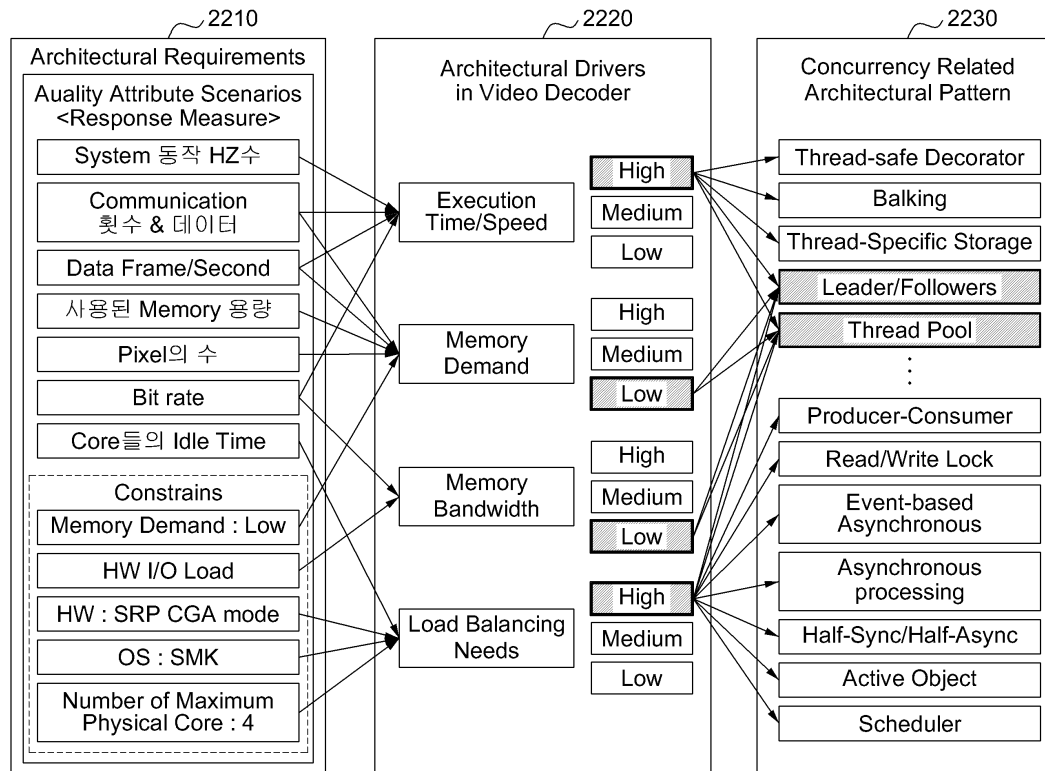
도면20



도면21



도면22



도면23

