프로세스 대수에 기반을 둔 수퍼스칼라 프로세서 프로그램의 시간 분석

(Process Algebraic Approach to Timing Analysis of Superscalar Processor Programs)

유 희 준 † 이 기 흔 †† 최 진 영 †††

(Hee-Jun Yoo) (Ki-Huen Lee) (Jin Young Choi)

요 약 다중 포트를 가진 레지스터의 장점은 읽기 접근에 대해서는 한번에 여러 명령어에서 레지스터를 공유할 수 있다는 것이다. 여기서는 높은 수준에서 이러한 다중 포트 레지스터를 가진 파이프라인 수퍼 스칼라 프로세서에서의 타이밍 특성과 자원 제한을 묘사하기 위한 정형방법을 제시한다. 특히, 파이프 라인 명령어가 순서대로 들어오는 경우에 대해서 먼저 명세하고, 순서에 상관없이 어느 싸이클에 검색 가능한 명령어들 중에서 동시에 실행 가능한 명령어 짝을 찾아 실행시키는 수퍼 스칼라 파이프라인 방식인 비순차(Out-of-Order) 명령어 수퍼 스칼라 방식에서의 타이밍 분석으로 확장하였다. 명령어 명세에는 프로세스 대수(Process Algebra)기반의 정형기법인 ACSR(Algebra of Communicating Shared Resources)을 이용하여 모델링한다.

Abstract Multi-ports register could shared several instructions at the same time in read operation. We address a formal methods for describing timing analysis and resource restriction in pipeline super scalar process that having multi-port registers. First, we specify in-order pipeline instructions, and then, extend timing analysis in out-of-order super-scalar. In this case, we find instruction pairs in any cycle which can execute same time. We use ACSR(Algebra of Communicating Shared Resources), a branch of formal methods based on process algebra, for instruction specification and modelling.

1. 서 론

다중 포트 레지스터는 수퍼 스칼라 프로세서에서 명령어의 병렬 실행을 효과적으로 향상시키기 위해 개발된 마이크로 프로세서 구조이다. 수퍼 스칼라 프로세서는 동시에 여러 명령어를 실행 유니트에 이슈(issue)할 수 있는 구조이지만, 명령어들 사이의 데이타 의존성 때문에 같은 레지스터를 읽고 쓰는 명령어들 사이에는 그 실행 순서가 유지 되어야하며, 분기 명령어 같은 명령어 같이 실행 순서가 바뀌는 경우가 있기 때문에 모든 프로그램에서 최대한의 명령어 병렬성을 얻을 수 없다. 다중 포트 레지스터를 사용하는 경우에는 여러 명령어가 한 레지스터에 대해 읽기 접근을 수행할 수 있으므로 어느 정도는 병렬성을 높일 수 가 있다[3][4][5]. 그러나, 이 경우에도 쓰기 접근에 대해서는 공유되어서는 안되기 때문에 그 경우에는 모든 포트가 한 명령어에 독점된다.

이 논문에서는 다중 포트를 사용하는 ToyP라는 모델 프로세서를 모델링하기 위해서 프로세스 대수(Process Algebra) ACSR(Algebra of Communicating Shared Resources)[1]를 정의하였다.

ACSR은 CCS[2](Calculus of Communicating System)의 일종이다. ACSR은 CCS에 시간과 자원의 개념이 첨가된 기법으로 시스템을 애매모호함이 없이 정확하게 명세를 할 수 있다.

여기서의 시도는 프로세스 대수에 기반한 정형기법을 통해 파이프라인 수퍼 스칼라 프로세스에서 일련의 명

· 한국학술진흥재단의 신진교수 공모과제 연구비의 지원을 받았습니다.(997-003-E0031)

† 학생회원 : 고려대학교 컴퓨터학과
hyoo@formal.korea.ac.kr
†† 비 회 원 : 고려대학교 컴퓨터학과
klee@formal.korea.ac.kr
††† 종신회원 : 고려대학교 컴퓨터학과 교수
choi@formal.korea.ac.kr

논문접수 : 1999년 2월 24일
심사완료 : 2000년 1월 3일

령어로 된 프로그램을 실행하는데 있어 타이밍 특성과 자원 제한을 분석하기 위한 방법을 제시함으로써 ISA(Instruction Set Architecture) 레벨의[6] 기술(記述)을 늘리고자 하는 것이다.

이러한 파이프라인 실행의 모델링에 대한 기존 연구로는 최초로 파이프라인 실행 행태를 반영하여 프로그램의 최악 실행시간을 분석하고자 한 연구는 Zhang등의 연구[7]가 있다. 이 연구에서는 Intel 80C188 프로세서의 두 단계 파이프라인 실행을 모델링하여 명령어의 중첩된 실행 행태를 최악 실행시간 분석에 반영하였지만, 모델로 삼은 Intel 80C188 프로세서는 두 단계만으로 이루어진 지극히 간단한 파이프라인 실행 구조를 갖추고 있어서 이 모델을 일반적인 파이프라인 실행 구조에 적용하기는 어렵다. 서울대학교 실시간 시스템 연구그룹의 Lim등이 제안한 확장된 타이밍 스키마[8]에서는 자원예약표를 이용하여 파이프라인 실행을 모델링하였다. 자원예약표는 명령어가 파이프라인 실행되는 모습을 파악하기 위해 제안된 기법으로 실행시간의 흐름에 따른 각 실행자원의 이용 상태를 표의 형태로 나타낸다. 플로리다 대학의 Healy등은 Lim등의 연구에서 사용한 자원예약표와 같은 형태의 파이프라인 실행도(pipeline diagram)[9]를 사용하여 파이프라인 실행을 모델링하였다. 프린스턴 대학의 Li등은 프로그램을 정수 선형 계획법을 사용해 표현하기 위해 각 기본 블록의 최악 실행시간을 상수로 결정하여 사용하였다. Li등의 연구와 Zhang등의 연구에서는 기본 블록 사이의 파이프라인 실행으로 인한 중첩 실행을 반영하지 않았기 때문에 분석 대상 프로그램의 특성에 따라서는 분석 결과와 실제 실행시간 사이에 큰 차이가 발생할 수 있지만,[18] 여기서 제안된 방법은 일반적인 명령어의 동작을 ACSR이라는 프로세스 대수에 기반한 정형기법을 사용하여 명령어 동작을 수학적인 모델링을 하여 검증한 결과로서 기존에 수행된 연구가 시스템이나 명령어의 실행 순서에 따라서 많은 차이를 보이는 데에 반하여, 시스템에 의존하지 않고 일반적인 명령어 집합에 대해서 같은 성능을 보이며 이와같은 명령어가 사용되는 모든 시스템에 적용 가능하다는 장점이 있다. 또한, ACSR은 VERSA(Verification, Execution, and Rewriting System for ACSR)[12]이라는 정형 검증 도구를 가지고 있어서 명세된 ACSR 구문이 원하는 동작성을 보이는 지를 검사할 수 있다.

이 논문의 구성은 2장에서는 ACSR의 기본적인 문법과 그 동작 의미를 설명하고, 3장에서는 여기서 사용할 ToyP 프로세서의 순차 명령어(In-Order Instruction)을 ACSR으로 명세하고, 4장에서는 ACSR을 이용해서 ToyP의 비순차 명령어(Out-of-Order Instruction)명세한 후, 5장에서 결론을 맺겠다.

2. 프로세스 대수 ACSR

ACSR(Algebra of Communicating Shared Resources)은 CCS(Calculus for Communi- cating Systems)에 기반한 프로세스 대수(Process Algebra)로써 시간, 자원, 우선순위, 동시성 등 실시간 시스템에 필요한 여러 개념을 포함한다.

2.1 ACSR의 Syntax

한 개의 ACSR 프로세스는 프로세스의 실행은 레이블이 있는 전이 시스템(labelled transition system)으로 정의된다. 한 예로 프로세스 P는 다음과 같은 행태(behavior)를 보일 수 있다. P는 ACSR 프로세스를 의미한다. 다음은 레이블이 있는 전이 시스템의 한 예이다.

$$P_1 \xrightarrow{\alpha_1} P_2 \xrightarrow{\alpha_2} P_3 \xrightarrow{\alpha_3} \cdots$$

즉, 처음에는 프로세스 P_1이 α_1이라는 액션(action)을 실행하고 프로세스 P_2로 전이하고, 프로세스 P_2는 α_2라는 액션을 실행한다. 두 프로세스 사이의 통신은 이벤트(event)와 역 이벤트(inverse event)로 이루어진다. 위의 두 이벤트사이에 이루어진 통신은 더 이상의 동기화를 막기 위해 내부 τ 액션으로 변환되어 외부로부터의 간섭을 금지시킨다. ACSR에서는 어떤 액션에 우선순위를 관련시켜서 우선순위가 제공되는데, 예를 들어 $a.P + b.Q$에서 a의 우선순위가 b보다 높으면 a가 먼저 실행된다. 다음의 문법은 ACSR 프로세스를 정의한다.

$$P ::= \mathrm{NIL} \mid A:P \mid (a,n).P \mid P+Q \mid P \| Q \mid [P]_I \mid P[f] \mid P \setminus F \mid P \setminus\setminus H \mid recX.P \mid X$$

NIL은 정지하여 아무런 액션을 하지 않는 프로세스를 의미한다. 즉, 데드락(deadlock)된 프로세스를 의미한다. ACSR에는 액션을 표시하는 두 종류의 표기가 있다. 첫 번째로, $A:P$는 자원을 소비하는 액션 A를 주어진 단위 시간 내에 실행하고 다음 프로세스인 P로 진행한다. 여기서 액션 A는 자원 r과 우선순위 p의 쌍 (r,p)로 나타낸다. 이에 반해 $(a,n).P$는 시간의 진행이 없는 이벤트 (a,n)를 실행하고 P로 진행하는 것이다. 여기서 a는 포트를 의미하고 n은 우선순위이다. 이 두 가지의 차이점은 이벤트의 경우 시간이 고려되지 않는다는 점

이다. 선택 연산자(Choice operator) $P+Q$는 비결정성(non-determinism)을 표현한다. 주변환경의 제약조건에 의해 프로세스 P가 선택되거나 프로세스 Q가 선택된다. 동시성 연산자 (concurrent operator) $P||Q$는 병렬 프로세스를 표현하게 해준다. 즉 프로세스 P와 Q가 동시에 진행함을 나타낸다. 클로즈 연산자(Close operator) $[P]_I$는 프로세스 P가 집합 I에 정의된 자원을 독점함을 표시한다. 리레이블링 함수(Relabelling function) $P[f]$는 프로세스 안에 있는 채널들의 레이블을 변환해 주는 함수이다. 제한 연산자(Restriction operator) $P\backslash F$는 P의 행태(behavior)를 제한한다. 즉 프로세스 P는 집합 F안에 정의된 이벤트는 실행할 수 없음을 표현한다. 숨김 연산자(Hiding operator) $P\backslash\backslash H$는 프로세스 P의 동작으로부터 집합 H안의 리소스를 숨긴다. $recX.P$는 리커젼(recursion)을 나타내며 같은 동작이 무한히 반복되는 것을 표현한다.

2.2 ACSR의 Operational Semantics

ACSR의 의미는 두 단계로 정의가 되는데, 여기서는 첫 번째 단계만 설명하기로 한다. 첫 번째 단계는 우선순위(priority)를 무시하고 의미를 설명한 뒤 두 번째 단계에서 우선 순위를 고려하는 것이다. 다음은 시간-소모 액션과 순간적인 이벤트에 대한 법칙이다.

$$\text{ActT}\ \frac{-}{A:P\xrightarrow{A}P}$$

$$\text{ActI}\ \frac{-}{(a,n).P\xrightarrow{(a,n)}P}$$

예를 들면, 프로세스 $\{(r_1,p_1),(r_2,p_2,)\}:P$는 하나의 단위 시간동안 자원 r_1과 자원 r_2를 동시에 사용하고, P를 수행한다. 마찬가지로 프로세스 $(a,p).P$는 이벤트 "(a,p)"를 수행하고 P로 진행한다.

다음은 두 프로세스 중에서 선택을 나타내는 법칙이며 이것은 시간-소모 액션과 순간적 이벤트에 동일하게 적용된다. 두 프로세스 P와 Q 중에서 시스템 내부의 상황에 따라 하나만 선택되어 다음 단계로 진행한다.

$$\text{ChoiceL}\ \frac{P\xrightarrow{\alpha}P'}{P+Q\xrightarrow{\alpha}P'}$$

$$\text{ChoiceR}\ \frac{Q\xrightarrow{\alpha}Q'}{P+Q\xrightarrow{\alpha}Q'}$$

프로세스의 병렬성은 다음과 같은 법칙으로 설명이 된다. 우선 첫 번째 법칙은 두 개의 시간-소모 전이에 관한 것이다.

$$\text{ParT}\ \frac{P\xrightarrow{A_1}P',Q\xrightarrow{A_2}Q'}{(P\|Q)\xrightarrow{A_1\cup A_2}(P'\|Q')}\quad(\rho(A_1)\cap\rho(A_2)=\varnothing)$$

여기서 시간-소모 전이는 완전히 동기적(synchronous)이며, $\rho(A)$는 액션 A에 의해 사용되는 자원들의 집합을 나타낸다. 조건 $\rho(A_1)\cap\rho(A_2)=\varnothing$은 각 자원은 완전히 순차적이며, 한 단위 시간동안 주어진 자원을 오직 한 프로세스만이 사용할 수 있다는 것을 나타낸다. 서로 다른 자원의 집합을 사용하는 두 개의 프로세스가 병렬로 수행될 경우, 각 프로세스가 사용하는 자원의 집합으로 나뉘어져 개별적으로 수행될 수 있다.

그리고 다음의 세 가지 법칙은 이벤트 전이에 대한 것으로 시간-소모 액션과 달리 이벤트는 비동기적(asynchronous)으로 발생할 수 있다.

$$\text{ParIL}\ \frac{P\xrightarrow{(a,n)}P'}{(P\|Q)\xrightarrow{(a,n)}(P'\|Q)}$$

$$\text{ParIR}\ \frac{Q\xrightarrow{(a,n)}Q'}{(P\|Q)\xrightarrow{(a,n)}(P\|Q')}$$

$$\text{ParCom}\ \frac{P\xrightarrow{(a,n)}P',\ Q\xrightarrow{(\bar{a},m)}Q'}{(P\|Q)\xrightarrow{(\tau,n+m)}(P'\|Q')}$$

처음의 두 법칙은 이벤트가 임의로 인터리빙(interleaving)되어 수행됨을 보여주고, 마지막 법칙은 두 개의 동기적 프로세스에 관한 것이다. 위에 표기된 두 전이는 발생하는 이벤트가 병렬로 수행되는 두 개의 프로세스에 영향을 주는 것이 아니라 둘 중에 하나의 프로세스에 대해서만 영향을 미치는 경우, 두 프로세스가 병렬 합성이 이루어져도 원래에 영향을 주었던 프로세스에 대해서만 전이가 발생한다. 세 번째의 전이는 병렬 합성되는 두 개의 프로세스가 서로 통신을 이루는 경우이다. 두 프로세스가 이벤트와 역 이벤트를 발생하는 전이를 한다면, 이러한 두 개의 프로세스가 병렬 합성될 때에, 두 이벤트는 내부 액션으로 변화하여 전이된다.

제한 연산자는 시스템의 행태로부터 제외되는 순간적 이벤트의 부분 집합을 정의한다. 이것은 $F(\tau\notin F)$에 속한 레이블들의 집합을 정해 놓고 이 레이블들에 포함되지 않는 이벤트들의 행태만을 이끌어냄으로써 이루어진다. 시간-소모 액션에서는 아무런 영향도 받지 않는다. 단지, 사용되는 자원을 중에서 동기화 되어야만 하는 자원을 제한하여 두 프로세스사이에서만 통신이 이루어질 수 있도록 해준다.

$$\text{ResT}\ \frac{P \xrightarrow{A} P'}{P \setminus F \xrightarrow{A} P' \setminus F}$$

$$\text{ResI}\ \frac{P \xrightarrow{(a,n)} P'}{P \setminus F \xrightarrow{(a,n)} P' \setminus F} \quad (a, \overline{a} \notin F)$$

제한에서는 프로세스들에 전용인 포트를 부여하는 반면, 클로스 연산자는 전용의 자원을 부여한다. 어떤 프로세스 P가 $[P]_I$와 같은 폐쇄 구문(closed context)안에 있을 때, 자원 I에 대한 더 이상의 공유는 없다.

$$\text{CloseT}\ \frac{P \xrightarrow{A_1} P'}{[P]_I \xrightarrow{A_1 \cup A_2} [P']_I} \quad A_2 = \{(r,0) \mid r \in I - \rho(A_1)\}$$

$$\text{CloseI}\ \frac{P \xrightarrow{(a,n)} P'}{[P]_I \xrightarrow{(a,n)} [P']_I}$$

리레이블링 함수는 프로세스안의 채널의 레이블을 변환시켜준다. 이벤트는 아무 영향도 받지 않고 남아있는다.

$$\text{Rel}\ \frac{P \xrightarrow{a} P'}{P[f] \xrightarrow{f(a)} P'[f]}$$

숨김 연산자는 리소스 정보를 외부 환경으로부터 숨긴다. 따라서 이벤트는 아무 영향도 받지 않은 상태로 남아있게 된다.

$$\text{HideT}\ \frac{P \xrightarrow{A} P'}{P \setminus\setminus H \xrightarrow{A'} P' \setminus\setminus H}$$

$$\text{where}\ A' = \{(r,n) \in A \mid r \notin H\}$$

$$\text{HideI}\ \frac{P \xrightarrow{(a,n)} P'}{P \setminus\setminus H \xrightarrow{(a,n)} P' \setminus\setminus H}$$

다음은 반복에 관한 전이이다.

$$\text{Rec}\ \frac{P[recX.P] \xrightarrow{a} P'}{rec\ X.P \xrightarrow{a} P'}$$

여기서, $P[recX.P]$는 P내의 X의 발생에 대해 $recX.P$로 대체하는 표준적 표기법을 나타낸다.

두 ACSR 프로세스의 등가(equivalence)는 우선순위의 개념이 첨가된 강 바이시뮬레이션 (strong bisimulation)이라는 개념에 의해 결정된다. 이 개념은 만약 프로세스 P(또는 Q)가 액션 a를 수행하여 한 스텝 진행하면 P(또는 Q) 또한 액션 a를 하며 한 스텝 진행이 되어야하며, 다음의 프로세스 역시 바이시밀러(bisimilar)하다는 의미이다.

이러한 바이시뮬레이션은 서론에서 소개한 VERSA를 이용하여 이 절에서 소개한 ACSR 문법에 따라 시스템의 전이를 하여, 비교하고자 하는 두 개의 프로세서가 등가관계에 있는 지를 자동적으로 검증하여 준다. 논문에서도 VERSA를 이용하여 명세된 명령어 집합에 대하여 검증하였다.

3. 순차 명령어에 대한 타이밍 분석

3.1 다중 포트 레지스터 프로세서 ToyP의 명령어 모델링

다중 포트 레지스터는 수퍼 스칼라 프로세서에서 명령어의 병렬 실행을 효과적으로 향상시키기 위해 개발된 마이크로 프로세서 구조이다. 수퍼 스칼라 프로세서는 동시에 여러 명령어를 실행 유니트에 배출(issue)할 수 있는 구조이다. 여기서는 ToyP 라는 32비트 가상 수퍼 스칼라 프로세서의 명령어를 사용하였다.

이 장에서는 모델링될 가상 비순차 파이프라인 수퍼스칼라 프로세서에 대한 명세를 서술한다.

모델 프로세서는 메모리 워드 크기가 32비트로 모든 명령어는 전부 한 워드(32비트)로 구성되며 별도의 워드를 필요로 하지 않는다고 가정한다. 데이타 이동의 단위 또한 32비트로 이루어지며 레지스터 사이의 데이타 이동도 32비트, 버스 크기도 전부 32비트인 것으로 가정하다. 그러나 하나의 명령어에서 메모리 가능 범위는 16비트로서 64K 단위로 하나의 메모리 블록을 구성한다.

파이프라인 단계 파이프라인을 구성하는 기능 유니트는 넷으로 이루어져 있다. 명령어 인출(fetch), 이슈(issue), 실행(execution), 되쓰기(write-back)가 그것이다. 각각의 단계는 모두 하나의 클럭 주기에 이루어지는 것으로 가정하며 모든 명령어는 이 네 단계를 빠짐없이 거치게 된다. 그러나 모델링에서는 이 중 이슈와 실행 단계만이 프로세스 에이전트로서 표현되게 된다. 연속적인 명령어의 흐름에서 보면 매 주기마다 각 기능 유니트는 언제나 명령어가 실행 중인 상태에 있게 되므로 두 기능 유니트만을 본다 해도 명령어의 실행을 모델링하는데는 무리가 없다.

모델 프로세스는 최대 세 개까지의 명령어를 동시에 실행할 수 있고, 여섯 개의 레지스터를 가진다. 레지스터는 읽기와 쓰기, 두 가지 동작을 가지는데 이것은 다시 여섯 개의 읽기/쓰기 포트를 통해 일어난다. 하나의 명령어는 최대 두 번의 읽기 동작이 가능하다. 예를 들어, Add R2,R1,R1 명령어는 R1 레지스터에 두 번 읽기 접근을 한다. 그러므로 세 명령어가 동시에 한 레지스터에 대해 두 번씩 읽기 접근을 할 때, 모든 명령어가 레지스터 충돌을 겪지 않으려면 적어도 한 레지스터에는 여섯 개의 읽기/쓰기 포트가 필요하다. 그러나 쓰기 접

근에 대해서는 둘 이상의 명령어가 중복될 수 없으므로 쓰기 접근은 포트를 하나만 쓰는 것이 아니라 여섯 개 모두를 독점하는 것으로 가정한다.

앞서 설명한 대로 레지스터는 ACSR 자원으로서 모델링된다. 어떤 명령어 프로세스가 레지스터 자원에 접근하기 위해서는 레지스터 자체가 아닌 레지스터 포트를 얻어야 하므로 레지스터 포트 하나 하나를 모델링한다. 레지스터 여섯 개는 1번부터 6번까지 R1,R2,..,R6가 존재하고 Ri_j는 레지스터 i번의 j번 포트를 나타낸다. 또, 그냥 Ri라고 하면 Ri_1, Ri_2,...Ri_6까지의 모든 포트 자원을 나타내는 것으로 한다.

예제를 위해 ToyP라는 가상적인 32비트 수퍼 스칼라 프로세서의 명령어를 사용할 것이다. ToyP 프로세서는 많은 상업적인 프로세서들의 특징을 갖추고 있으며, 앞서 말한 Harcourt 등에 의해 개발되었다. 그들은 일찌기 명령어 명세를 위해 SCCS라는 프로세스 대수를 이용했었다[10]. SCCS에는 자원이란 개념이 없었으므로 각 자원을 바이너리 세마포어(binary semaphore)로 나타내어, 제시된 명세는 매우 복잡하고 거추장스러웠다. 더군다나 SCCS는 우선순위 개념 또한 없어 CCS를 위해 개발된 우선순위 연산자를 끌어 들여야 했다. 아래에는 [11]에서 모델링한 ToyP 명령어에다 [14]에서 추가한 간단한 분기 명령어를 추가해서 나타내었다.

표 1 ACSR로 표현된 순차 명령어

Add Ri, Rj, Rk	Ri ← Rj + Rk
Mov Ri, Rj	Ri ← Rj
Load Ri, Rj, #c	Ri ← Mem[Rj + c]
Store Ri, Rj, #c	Mem[Rj + c] ← Ri
Jump #c	PC ← c

$$\mathrm{Add}\,\mathrm{R}i,\mathrm{R}j,\mathrm{R}k \overset{\text{def}}{=} \sum_{1\le l\le 6}\sum_{1\le m\le 6}\{\mathrm{R}i,\mathrm{R}j_l,\mathrm{R}k_m\}:\mathrm{Done}$$

$$\mathrm{Mov}\,\mathrm{R}i,\mathrm{R}j \overset{\text{def}}{=} \sum_{1\le l\le 6}\{\mathrm{R}i,\mathrm{R}j_l\}:\mathrm{Done}$$

$$\mathrm{Load}\,\mathrm{R}i,\mathrm{R}j,\#c \overset{\text{def}}{=} \sum_{1\le l\le 6}\{\mathrm{R}i,\mathrm{R}j_l\}:\{\mathrm{R}i\}:\mathrm{Done}$$

$$\mathrm{Store}\,\mathrm{R}i,\mathrm{R}j,\#c \overset{\text{def}}{=} \sum_{1\le l\le 6}\sum_{1\le m\le 6}\{\mathrm{R}i_l,\mathrm{R}j_m\}:\{\mathrm{R}i_l\}:\mathrm{Done}$$

$$\mathrm{Jump}\,\#c \overset{\text{def}}{=} \{\mathrm{J}\}:\mathrm{Insts}(c)$$

보통 Add와 Mov명령어는 한 명령어 사이클에 연산이 실행된다. 반면 메모리와 관련한 명령어인 Load, Store는 실행되는데 두 사이클이 필요하다. Jump는 다른 어떤 명령어와도 동시에 실행되지 못하며 프로그램 카운터(PC)를 분기 목적지를 가르키도록 한다.

ToyP의 전체 시스템은 우선 순위가 전부 같은 레지스터들의 유한 집합 R로 구성되어 있다고 생각한다. 하나의 액션은 R의 부분집합으로 정의되며 한 사이클 타임이 걸린다. 명료성을 위해 이 다음부터는 액션에서 우선 순위가 모두 같다고 생각하고 빼도록 하겠다. 그리고 액션 $\varnothing$(또는 {})은 한 시간 단위 동안 아무 것도 하지 않는 것을 나타내며, A, B, C는 액션을 나타내는 문자이다.

정의 3.1 명령어 실행의 종료를 나타내는 프로세스 Done을 다음과 같이 정의한다.

$$\mathrm{Done} \overset{def}{=} rec\,X.\phi:X$$

표 2 ToyP 프로세서의 순차 명령어에 대한 실행 모델링

$$\mathrm{Insts(PC)} \overset{def}{=} \phi:\mathrm{Insts(PC)} + \mathrm{Super\text{-}Insts(PC)}$$

$$\begin{aligned}\mathrm{Super\text{-}Insts(PC)} \overset{def}{=}\ & (\mathrm{Mem(PC)}\|\mathrm{Mem(PC+4)}\|\mathrm{Mem(PC+8)})\,next\ \mathrm{Insts(PC+12)}\\ & +(\mathrm{Mem(PC)}\|\mathrm{Mem(PC+4)})\,next\ \mathrm{Insts(PC+8)}\\ & +\mathrm{Mem(PC)}\,next\ \mathrm{Insts(PC+4)}\\ & +\mathrm{Mem(PC)}\end{aligned}$$

$$\mathrm{Program} \overset{def}{=} [\mathrm{Super\text{-}Insts(PC)}]_R$$

프로세스 Done은 병렬 연산자 ||에 대해 다음과 같은 특성이 있다. 모든 프로세스 P에 대해 P || Done = P이다.

다음의 이진 연산자 *next*는 이어지는 사이클에 명령어를 배출(issue)하는 것을 모델링한다. 자원 J는 분기 명령어를 검색하기 위한 것으로 분기 명령어는 똑같이 자원 J를 사용함으로써 *next* 연산자와 충돌하여 일반적으로 순서적인 명령어 배출이 이루어지지 않도록 한다.

정의 3.2 임의의 P, Q에 대해, 다음과 같이 정의된다.

$$P\,next\,Q \overset{def}{=} P\|\{\mathrm{J}\}:Q$$

[표 1]에 나와 있는 대로 ACSR을 이용하여 ToyP 명령어를 모델링한다. [표 1]에 의해 얻어진 ToyP 프로그램에 대한 ACSR 프로세스의 집합을 프로그램 명세라고 한다. 다음 예제는 ToyP 프로그램을 프로그램 명세로 번역하는 방법을 보여 준다.

예제 3.1 메모리 위치 PC에 다음 프로그램이 있을

때,

```
PC:     Add  R1, R1, R1
PC+4:   Jump #(PC+12)
PC+8:   Load R2, R3, #8
PC+12:  Add  R1, R3, R3
```

이는 다음의 ACSR 프로세스로 나타내어 질 수 있다.

$$\begin{aligned}
\mathrm{Mem(PC)} &\overset{def}{=} \{\mathrm{R1}\}:\mathrm{Done}\\
\mathrm{Mem(PC+4)} &\overset{def}{=} \{\mathrm{J}\}:\mathrm{Insts(PC+12)}\\
\mathrm{Mem(PC+8)} &\overset{def}{=} \{\mathrm{R2},\mathrm{R3}_1\}:\{\mathrm{R2}\}:\mathrm{Done}\\
\mathrm{Mem(PC+12)} &\overset{def}{=} \{\mathrm{R1},\mathrm{R3}_2,\mathrm{R3}_3\}:\mathrm{Done}\\
\mathrm{Mem(PC+16)} &\overset{def}{=} \mathrm{NIL}
\end{aligned}$$

3.2 ACSR을 이용한 ToyP 실행 모델링

데이타 해저드(data hazard)를 가진 모든 명령어는 ACSR 식에서 같은 자원을 사용하는 프로세스로 묘사되며 같이 실행될 경우 충돌하여 실행될 수 없는 프로세스가 된다. 예를 들어 같은 레지스터 Ri에 쓰기를 하는 명령어와 읽기를 하는 명령어가 있다면 쓰기 명령어의 경우는 레지스터 포트를 전부 독점하므로 {Ri}의 액션을 하는 ACSR 프로세스가 될 것이고 읽기 명령어는 레지스터 포트를 하나만 사용하므로 {Ri_l}의 액션을 하는 ACSR 프로세스로 표현될 것이다([표 1]). 이 두 프로세스가 동시에 수행되면 사용하는 레지스터가 같으므로 지정된 범위($\leq$ 6)를 벗어나는 것이어서 NIL이 될 수 밖에 없다. 이런 식으로 데이타 해저드를 가진 모든 명령어 프로세스는 NIL이 되어 명령어 모델링에서는 전부 제거되게 된다. ACSR의 이러한 성질은 타이밍 분석을 용이하게 하고 최종 명세에서 실제로 사용되어 지는 레지스터 자원의 개수를 정확히 산정할 수 있게 해 준다.

ToyP 프로그램의 실행은 각 명령어 주기에서 가장 많은 자원이 소비될 수 있는 쪽으로 진행되게 된다[11]. 이 과정을 다음 예제 3.2에서 자세히 살펴 보도록 하겠다. 예제 3.2은 동시에 실행되는 프로세스끼리 어떻게 자원의 공유가 일어나는지 잘 보여 주고 있다..

예제 3.2 예제 3.1에 나와 있는 ToyP 프로그램 명세를 이용하여 확장하는 과정을 보여 주고 있다.

$$\begin{aligned}
&\mathrm{Program}\\
&= \begin{bmatrix} \mathrm{Mem(PC)} \| \mathrm{Mem(PC+4)} \| \mathrm{Mem(PC+8)}\ \mathit{next}\ \mathrm{Insts(PC+12)} + \\ \cdots + \mathrm{Mem(PC+4)}\ \mathit{next}\ \mathrm{Insts(PC+8)} + \mathrm{Mem(PC+4)} \end{bmatrix}_R\\
&= \begin{bmatrix} \{(\mathrm{R1},6),(\mathrm{R2},6),(\mathrm{R3},1),(\mathrm{J},2)\}:\{(\mathrm{R2},6)\}:\mathrm{Done} \| \mathrm{Insts(PC+12)} \| \cdots + \\ \cdots \{(\mathrm{R1},6),(\mathrm{J},1)\}:\mathrm{Insts(PC+8)} + \{(\mathrm{R1},6)\}:\mathrm{Done} \end{bmatrix}_R\\
&= [\mathrm{NIL} + \mathrm{NIL} + \{(\mathrm{R1},6),(\mathrm{J},1)\}:\mathrm{Insts(PC+8)} + \{(\mathrm{R1},6)\}:\mathrm{Done}]_R\\
&= [\{(\mathrm{R1},6),(\mathrm{J},1)\}:\mathrm{Insts(PC+8)}]_R
\end{aligned}$$

첫번째 주기에서 충돌하는 프로세스를 모두 제거하면 결국 처음 명령어 만이 실행될 수 있음을 알 수 있다. 다음 주기에서는 분기 명령어가 있으므로 바로 네번째 명령어로 건너가서 실행되도록 한다. 최종적인 확장식은 다음과 같이 된다.

$$\mathrm{Program} = [\{\mathrm{R1},6\}:\{\mathrm{J},1\}:\{(\mathrm{R1},6),(\mathrm{R3},1)\}:\mathrm{Done}]_R$$

4. 비순차 명령어에대한 타이밍 분석

4.1 In-Order vs. Out-Order

순차 방식을 이용할 경우에 아래의 예에서는 높은 병렬성을 얻을 수 없다.

1) Add R1, R1, R1
2) Load R2, R3$_1$, #8
3) Mov R4, R1$_1$
4) Store R4$_1$, R3$_2$, #4
5) Add R5, R5, R6$_1$

그 이유는 3) 명령어에서 레지스터 R1을 독점하고 있기 때문에 아까처럼 1)과 2) 명령어가 동시에 실행될 수 없다. 대신 1)과 3) 명령어가 동시에 실행될 수 있는 것을 알 수 있다. 그러나 앞에서 사용한 명령어 스케줄링 알고리즘에선 이것을 검사하여 실행할 수 없다. 그러한 파이프라인 명령어 배열 전략을 순차(in-order) 방식이라고 한다. 들어오는 명령어 순서대로 짝을 찾기 때문에 명령어 순서가 항상 지켜지는 것은 보장하지만 높은 병렬성을 얻기는 힘들다. 반대로 순서에 상관없이 어느 사이클에 검색 가능한 명령어들 중에서 동시에 실행 가능한 명령어 짝을 찾아 실행시키는 수퍼 스칼라 파이프라인 방식(즉, 위의 명령어 세트에서 1)과 3) 명령어를 동시에 내보낼 수 있는 방식)을 비순차(out-of-order) 방식이라고 한다. 비순차 수퍼 스칼라 방식이 좀더 진보되고 최적화된 방식이긴 하지만 훨씬 더 복잡한 마이크로 프로세서 회로를 요구한다. 실제 실용화된 마이크로 프로세서들에서는 명령어 캐쉬와 파이프라인 사이에 또

다른 버퍼를 두어 정해진 수의 명령어를 버퍼로 옮긴 다음 순서를 재배열하여 파이프라인으로 들여보낸다.

4.2 비순차 ToyP 명령어 모델링

비순차 명령어 집합도 순차 명령어 집합과 같은 명령어들을 가지고 작업을 하였다. 여기서의 차이는 순차 명령어에서는 Jump 명령어가 단지 현재의 프로그램 카운터를 지정된 분기점으로 이동시키는 역할을 하는 것과 달리 여기서 명세한 Jmp는 비순차 명령어 집합에서 분기된 부분에서부터 다시 동시에 최대한 많은 명령어를 실행할 수 있도록 동시에 실행 가능한 명령어 쌍을 다시 찾게 된다. 이러한 역할을 수행하기 위해서 Issue라는 프로세스를 정의해 주었다. 여기서는 추가된 Jmp 명령어에 대한 모델링만을 소개하겠다. Add, Mov, Load, Store는 앞서 설명한 순차 명령어 모델링([표 1])을 참고하기 바란다.

Jmp #c PC ← c

위에서 정의된 Jmp는 명령어의 동작상으로 순차 정렬과 같은 행동을 하게 된다. 하지만, 비순차 명령어에서는 최대한의 병렬성을 얻기 위하여 분기점 다음 위치에서 동시에 수행될 수 있는 가장 많은 수의 명령어를 찾아내어야 하기 때문에 [표 3]과 같이 Issue 프로세스를 이용하여 명세하였다. 나머지 4개의 명령어는 [표 1]을 참고하기 바란다.

표 3 ACSR로 표현된 비순차 명령어

$$Jmp\,\#c \overset{def}{=} Issue(c, c+4, c+8)$$

레지스터 사용에 있어 최대한의 병렬성을 얻기 위해, 하나의 명령어 프로세스는 그 프로세스가 실행되는 시점에서 사용 가능한 모든 레지스터 자원의 조합에 따른 각각의 프로세스를 선택 합성함으로써 이루어 진다. 즉, Mov R*i*, R*j*에 해당하는 ACSR 프로세스는 $\sum_{1\le i\le 6}\{Ri, Rj_i\}:Done$으로서 이는 R*j*의 읽기 쓰기 포트 여섯 개 중 어느 하나라도 유효하다면 사용 가능하도록 해 준다.

이렇게 해서 변환된 ACSR 프로세스의 집합을 프로그램 명세라고 한다. 다음은 이런 프로그램 명세의 한 예이다.

예제 4.1 다음과 같은 메모리 위치에 명령어들이 저장되어 있다고 하자. 0번과 4번의 명령어는 R1 레지스터에서 쓰기 동작과 읽기 동작이 이루어지므로 동시에 수행될 수 없다. 따라서 분기된 후에 많은 수의 명령어를 동시에 수행하기 위해서 실행 가능한 명령어의 쌍을 찾아야만 한다.

0 : Add R1, R1, R1
4 : Load R2, R1, #8
8 : Mov R4, R2
12 : Add R5, R5, R6
16 : Jmp 0
20 : Store R3, R1, #12

메모리 주소가 4씩 증가하는 것은 메모리 주소가 바이트 단위로 매겨지고, 모든 명령어의 크기가 4바이트이기 때문이다. 위 명령어들은 [표 3]의 변환식에 따라 다음과 같은 ACSR 프로세스로 변환된다.

$$Mem(0) \overset{def}{=} \{R1\}:Done$$
$$Mem(4) \overset{def}{=} \{R1_1, R2\}:\{R2\}:Done$$
$$Mem(8) \overset{def}{=} \{R5, R6_1\}:Done$$
$$Mem(12) \overset{def}{=} \{R5, R6_1\}:Done$$
$$Mem(16) \overset{def}{=} Issue(0,4,8)$$
$$Mem(20) \overset{def}{=} \{R1_2, R3_1\}:\{R1_2\}:Done$$

4.3 ACSR을 이용한 실행 모델링

실행 모델링은 두 개의 프로세스로 나뉘어 진다. 하나는 분기 명령어를 검출하고 명령어를 다음 실행 단계로 넘기는 Issue(PC_1, PC_2, PC_3) 프로세스, 또 하나는 재배열 버퍼에 저장된 명령어들 중에 가능한 한 많은 명령어를 찾아 실행하는 Exec(PC_1, PC_2, PC_3)이다. 각각의 매개 변수 PC_1, PC_2, PC_3는 재배열 버퍼의 명령어 주소 인덱스로서 명령어가 저장된 메모리 위치를 가리킨다.

수퍼 스칼라 프로세서에서 한 번에 두 명령어가 하나의 레지스터에 대해 읽고 쓰기를 겹쳐서 하는 경우를 데이타 해저드라고 한다. 이러한 데이타 해저드는 실행 모델링에서 두 명령어 프로세스 사이의 자원 사용 충돌로 나타난다. 예제 4.1의 프로그램 명세에서 프로세스 Mem(0)와 Mem(4)는 자원 R1를 사용하는데 있어 서로 충돌하고 있으며 실제로 두 명령어는 레지스터 R1에 읽기와 쓰기를 동시에 함으로써 데이타 해저드를 갖고 있다. 이렇게 서로 자원 사용에 있어 충돌하는 두 프로세스는 ACSR의 확장 법칙에 따라 NIL이 되어 잘못된 실행을 표시한다.

데이타 해저드와 비슷하게 분기 명령어 다음에 나오는 명령어가 그 명령어와 동시에 실행되는 것을 콘트롤

표 4 ACSR을 이용한 ToyP의 비순차 명령어에 대한 실행 모델링

$$\text{Issue}(PC_1, PC_2, PC_3) \overset{def}{=} \{B\} : (\text{Issue}(PC_1, PC_2, PC_3) + \text{EXEC}(PC_1, PC_2, PC_3))$$

$$\begin{aligned}\text{EXEC}(PC_1, PC_2, PC_3) \overset{def}{=}\ & \text{Mem}(PC_1) \| \text{Mem}(PC_2) \| \text{Mem}(PC_3) \| \text{Issue}(PC_3 + 4, PC_3 + 8, PC_3 + 12) \\ & + \text{Mem}(PC_1) \| \text{Mem}(PC_2) \| \text{Issue}(PC_3, PC_3 + 4, PC_3 + 8) \\ & + \text{Mem}(PC_1) \| \text{Mem}(PC_3) \| (\text{Mem}(PC_1) \| \text{Mem}(PC_3)) \setminus [\varnothing, f_R] \| \text{Issue}(PC_2, PC_3 + 4, PC_3 + 8) \\ & + \text{Mem}(PC_1) \| \text{Mem}(PC_2, PC_3, PC_3 + 4) \\ & + \text{Mem}(PC_1)\end{aligned}$$

$$\text{Program} \overset{def}{=} [\text{Issue}(0,4,8)]_{R \cup B}$$

$$f_R = \{Si_j / Ri_j \mid 1 \le i \le 6, 1 \le j \le 6\}$$

해저드라고 한다. 분기 명령어는 바로 다음에 순차적으로 이어지는 명령어와 동시에 실행되어서는 안되며 선택적 분기에 따른 목적 주소 위치로부터 새로이 명령어 배열을 인출해야 한다. 그러므로 분기 명령어가 해독됨과 동시에 이미 인출 및 이슈 유니트에 올라온 명령어들은 모두 플러쉬되어야 하며 이것은 한 클럭 주기의 실행 지연을 가져온다. 모델링에서는 콘트롤 해저드 또한 데이타 해저드와 마찬가지로 자원 충돌로 묘사된다. 분기 명령어는 그 자체로 Issue 프로세스이며 Issue 프로세스는 분기 검출을 위해 자원 B를 사용한다. 그러므로 원래의 Exec 프로세스에서 실행되는 Issue와 분기 명령어의 Issue는 자원 B를 사용하는데 서로 충돌하게 되어 같이 실행될 수 없고 분기 명령어는 오직 단독으로만 실행될 수 있다.

Exec 프로세스는 한꺼번에 실행되는 명령어의 수와 순서에 따라 [표 4]의 식에서 6가지 선택을 가진다. 첫 번째는 가장 많은 세 개의 명령어를 동시에 실행하는 것이고 두 번째, 세 번째, 네 번째는 배열을 달리 해 가며 명령어 두 개를 실행하는 것, 다섯 번째는 버퍼에서 가장 맨 앞의 명령어 하나만을 실행하는 것, 그리고 마지막 하나는 분기 명령어를 실행하는 것이다. Exec 프로세스는 Program의 정의에서 모든 레지스터 자원과 분기 검출 자원(B)의 집합에 대해 폐쇄되어 있으므로 언제나 가장 많은 명령어를 실행하는 쪽으로 행동하게 된다.

5. 결 론

이 논문에선 ACSR을 이용하여 수퍼 스칼라 프로세서 프로그램에서 명령어 수준 병렬성의 시간적인 특성과 자원 제한을 명세하는 테크닉을 제시하였다. 이것을 예시하는데 ToyP 라는 간단한 가상 수퍼 스칼라 프로세서를 이용하였으며, 프로세서는 레지스터 자원의 집합으로, 명령어는 매 사이클마다 그 레지스터 자원의 일부를 공유하는 ACSR 프로세스로 표현하였다. 또한 ToyP 프로그램도 ACSR 프로세스의 인덱스화된(indexed) 집합으로 표현하고, ACSR에서 제공하는 법칙을 이용하여 해당 ACSR 프로세스를 간략화 함으로써 본래 ToyP 프로그램의 시간적 특성을 분석할 수 있었다.

논문에서는 프로세스 대수 기반의 ACSR을 사용하여 명령어의 시간 분석을 하여 정형 검증 도구인 VERSA를 이용하여 수행된 모델링의 시간 분석 실행에 대한 정형 검증을 수행하였다. 기존의 시간 분석에 대한 연구에서는 파이프라인 명령어의 최악 실행시간에 대한 실측이나, 시뮬레이션에 의한 검증을 수행한 것에 반하여, 논문에서 제시한 방법은 시스템등에 의존하지 않은 일반적인 모델링을 통하여 검증도구를 이용하여 실행 모델링의 시간 분석을 수행하여서 정확한 분석을 할 수 있다.

여기서 제시된 ACSR은 시간과 자원을 제공하기에 단위 시간에 레지스터라는 자원의 충돌을 줄이면서 수퍼스칼라 프로세서 프로그램의 명령어가 최대한의 병렬성을 얻을 수 있도록 모델링하였다. 논문에서 제시된 모든 ACSR 명세들은 ACSR 검증 도구인 VERSA를 이용하여 실험된 것들이며 만족할 만한 결과를 얻을 수 있었다. 앞으로의 연구과제는 실제 수퍼 스칼라 프로세서까지 그 명세를 확장하는 것이다.

참 고 문 헌

[1] Pratrice Bremond-Gregoire, Jin-Young Choi, Insup Lee, "A Complete Axiomatization of Finite-state ACSR Processes," *Information and Computation*, No 138, 1997

[2] R.Milner, *Communication and Concurrency*. Prentice Hall, 1989.

[3] M. Johnson. "Superscalar Micro-processor Design". *Prentice-Hall*, 1991.

[4] E. Harcourt, J. Mauney, and T. Cook. "Specification

of Instruction-Level Parallelism". *In Proc. of the North American Process Algebra Workshop*, 1993.
[5] R. Rau and J. Fisher. "Instruction- Level Parallel Processing: History, Overview, and Perspective". *Journal of Supercomputing*, July, 1993.
[6] T. Cook, P. franzon, E. Harcourt, and T. Miller. "System-Level Specification of Instruction Sets". In *Proc. of the International Conference on Computer Design*, 1993.
[7] N. Zhang, A. Burns, and M. Nicholson, "Pipelined Processors and Worst-Case Execution Times," *Real-Time Systems*, Vol. 5, No. 4, pp. 319-343, October 1993.
[8] S. -S. Lim, Y. H. Bae, G. T. Jang, *etc.*, "An Accurate Worst Case Timing Analysis for RISC Processors," *IEEE Transactions on Software Engineering*, Vol. 21, No. 7, pp. 593-604, July 1995.
[9] C. A. Healy, D. B. Whalley, and M. G. Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching," *In Proceedings of the 16th Real-Time Systems Symposium*, pp. 288-297, December 1995.
[10] J. Camilleri and G. Winskel. "CCS with Priority Choice". In *Proc.of IEEE Symposium on Logic in Computer Science*, 1991.
[11] Jin-Young Choi, Insup Lee, and Inhye Kang. "Timing Analysis of Superscalar Processor Programs Using ACSR," *IEEE Real-Time Systems Newsletter*, Volume 10, No. 1/2, 1994.
[12] D. Clarks, I. Lee, and H. Xie. "VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems". *Technical Report MS-CIS-93-77*, Dept. of CIS, Univ. of Pennsylvania, Sept 1993.
[13] 정창성 역. "고급 컴퓨터 구조학". 이한 출판사, 1997. (K. Hwang. Advanced Computer Architecture. *McGraw-Hill Book Co.-Singapore*, 1996).
[14] 이기혼, 최진영, "분기 명령어를 포함한 수퍼 스칼라 프로그램의 타이밍 분석", 정보 과학회 추계 학술 대회, 1997.
[15] 이기혼, 최진영, "ACSMR을 이용한 수퍼 스칼라 프로그램의 타이밍 분석", 정보 과학회 춘계 학술 대회, 1998.
[16] 유회준, 이기혼, 최진영, "ACSMR을 이용한 수퍼 스칼라 프로세서 프로그램의 비순차 정렬 명령어의 타이밍 분석", 병렬처리시스템 학술발표회, 1998.
[17] 이기혼, 최진영, "ACSR을 이용한 비순차 수퍼 스칼라 프로세서의 시간 분석", 정보 과학회 추계 학술 대회, 1998.
[18] 임성수, 민상렬, "실시간 시스템을 위한 취악 실행기간 분석 기법", 정보과학회지, pp32-38, 1996.8.

유 희 준
1997년 8월 고려대학교 컴퓨터학과 학사. 1999년 8월 고려대학교 대학원 컴퓨터학과 석사. 1999년 9월 ~ 현재 고려대학교 대학원 컴퓨터학과 박사과정. 관심분야는 컴퓨터이론, 정형기법(정형 명세, Formal verification), 실시간 시스템, 네트워크 보안

이 기 혼
1997년 2월 고려대학교 컴퓨터학과 학사. 1999년 2월 고려대학교 대학원 컴퓨터학과 석사. 관심분야는 컴퓨터이론, 정형기법(정형 명세, Formal verification), 실시간 시스템

최 진 영
1982년 서울대학교 컴퓨터공학과 학사. 1986년 Drexel University Dept. of Mathematics and Computer Science 석사. 1993년 University of Pennsylvania Dept. of Computer and Information Science 박사. 1993년 ~ 1996년 Research Associate, University of Pennsylvania. 1994년 ~ 1995년 Computer Scientist, Computer Command and Control Company(Part time). 1996년 ~ 1998년 고려대학교 컴퓨터학과 조교수. 1999년 ~ 현재 고려대학교 컴퓨터학과 부교수. 관심분야는 컴퓨터이론, 정형기법(정형 명세, Formal verification), 실시간 시스템, 분산 프로그래밍 언어, 소프트웨어 공학