고정 주기 스케줄링 기반의 실시간 시스템에서의 최악 demand paging 비용 분석

이영호[1] 임성수[2] 김지홍[1]
[1]서울대학교 전기컴퓨터공학부, [2]국민대학교 전산과학과
buriburi205@davinci.snu.ac.kr, sslim@kookmin.ac.kr, jihong@davinci.snu.ac.kr

Analysis of Worst-case Demand Paging Cost in Fixed-Priority Scheduling Real-Time Systems

Young-Ho Lee[1], Sung-Soo Lim[2], Jihong Kim[1]
[1]School of Computer Science and Engineering, Seoul National University, Seoul, Korea
[2]School of Computer Science, Kookmin University, Seoul, Korea

요 약

일반적으로 실시간 시스템에서는 실시간성을 만족시키기 위해 프로그램 코드를 부팅 시간에 DRAM에 적재시킨 후 수행하는 shadowing 기법을 사용한다. 그러나 shadowing 기법은 시스템의 부팅 시간을 증가시키고 불필요한 DRAM 영역을 차지한다는 단점이 있다. 프로그램의 크기가 커지고 복잡해짐에 따라 실시간 시스템에서도 demand paging의 필요성이 증가하고 있다. 그러나 demand paging 환경에서는 페이지 폴트(page fault)에 의한 시간 지연 때문에 태스크의 최악 응답 시간을 예측하기 어렵다는 단점이 있다. 본 논문에서는 demand paging 기반의 고정 주기 실시간 시스템에서 demand paging 비용을 고려한 태스크 최악 응답 시간 분석 기법을 제안한다. 제안하는 기법은 각 태스크의 실행 경로와 이전 태스크 인스턴스에 의한 페이지 재사용 양상에 따라 demand paging 비용이 달라진다는 점을 고려한다. 분석 과정은 페이지 순서 분석 단계와 페이지 재사용 분석 단계로 이루어지며, 태스크내(intra-task) 페이지 재사용 태스크간(inter-task) 페이지 재사용을 모두 고려한다. 실험 결과 제안한 기법은 실측값과 비교하여 최대 30% 이내의 오차로 최악 응답 시간을 예측함을 보였다.

1. 서 론

일반적으로 가상 메모리와 demand paging 기법은 실시간 임베디드 시스템보다는 서버나 데스크톱 환경에서 주로 사용되어 왔는데, 주된 이유는 가상 메모리와 demand paging 기법이 실시간 시스템 상에서 실시간성을 만족시키기 어렵기 때문이다. 가상 메모리와 demand paging 기법은 다음과 같은 이유로 실시간 시스템에서 사용하기에 적합하지 않다고 알려져 있다.

- 주소 변환: 가상 주소로부터 물리 주소로 주소 변환을 수행하는 과정에서 TLB lookup이 수행되는데, TLB lookup이 실패한 경우에만 페이지 테이블을 참조한다. 따라서 주소 변환에 걸리는 최악 실행 시간을 예측하기 어렵다.
- demand paging: 태스크 수행중에 발생하는 페이지 폴트(page fault)는 태스크의 메모리 접근 패턴(access pattern), 페이지 캐시(page cache)의 크기, 페이지 교체 알고리즘(replacement algorithm) 등에 따라 달라지기 때문에, 최악 demand paging 비용을 예측하기 어렵다.

그러나 실시간 시스템이 복잡해지고 다기능을 요구함에 따라 실시간 시스템에서도 가상 메모리와 demand paging 기법의 필요성이 커지고 있다. 가상 메모리는 서로 다른 실시간 태스크 간에 메모리 보호(memory protection)를 수행함으로써 태스크 수행 중 오류가 발생해도 다른 태스크에 영향을 미치지 않게 한다는 장점이 있다. 또한 demand paging 기법은 태스크 수행 중에 적재될 필요가 있는 페이지만 적재함으로써 불필요한 메모리 사용량을 줄이고, 시스템의 부팅 시간을 빠르게 할 수 있다는 장점이 있다.

최근 가상 메모리와 demand paging 기법을 실시간 시스템에서 사용하기 위한 연구가 진행되고 있다. [13][14]에서는 가상 메모리에서의 실시간 주소 변환 기법을 제안하였으며, [15][16]에서는 실시간 demand paging을 위한 컴파일러 수준에서의 최적화 기법에 대해 제안하였다. 그러나 최악 demand paging 비용을 예측하기 위한 연구는 미비한 상태이다.

본 논문에서는 demand paging 기반의 고정 주기 실시간 시스템에서 demand paging 비용을 고려한 태스크 최악 응답 시간 분석 기법을 제안한다. 제안하는 기법은 각 태스크의 실행 경로와 이전 태스크 인스턴스에 의한 페이지 재사용 양상에 따라 demand paging 비용이 달라진다는 점을 고려한다. 본 논문에서는 이러한 페이지 재사용 양상을 태스크내(intra-task) 페이지 재사용 및 태스크간(inter-task) 페이지 재사용으로 구분하고, 이를 모두 고려한 최악 demand paging 비용 예측 기법을 제시한다. 분석 기법은 크게 페이지 순서(page sequence) 분석 단계와 페이지 재사용(page reuses) 분석 단계로 나누어진다. 실험 결과 제안한 기법은 실측값과 비교하여 최대 30% 이내의 오차로 최악 응답 시간을 예측하였다.

본 논문의 구성은 다음과 같다. 2장에서는 최악 응답 시간 분석 기법에 대한 기존 연구를 살펴본다. 3장에서는 논문에서 제안하는 최악 demand paging 비용 분석 기법에 대해 살펴본다. 4장에서는 제안한 기법의 정확성을 검증하기 위해 최악 응답 시간 분석 도구를 구현하여 시뮬레이션을 수행하고, 이를 demand paging이 구현된 임베디드 보드 환경에서의 실측 결과와 비교한다. 마지막으로 5장에서는 결론 및 향후 과제를 제시한다.

2. 관련 연구

실시간 시스템을 구성할 때는 스케줄링 가능성 분석(schedulability analysis)을 통해 시스템 내의 태스크 집합이 실시간 성능 요구조건 (i.e., deadline)을 만족시키는지를 판단할 수 있어야 한다. 본 논문에서 제안하는 기법은 최악 응답 시간(worst-case response time: WCRT) 분석 기법을 바탕으로 한다. 최악 응답 시간 분석 기법에서는 각 태스크에 대해 최악 응답 시간을 계산하고, 데드라인보다 작거나 같을 경우 해당 시스템은 스케줄 가능하다고 한다. 태스크 τ_i의 최악 응답 시간 R_i는 τ_i의 최악 실행 시간(worst-case execution time: WCET)과 상위 우선순위를 가지는 태스크에 의한 시간 지연의 합으로 나타내며, 식 1과 같다.

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (1)$$

식 1에서 $hp(i)$는 τ_i보다 높은 우선순위에 해당하는 태스크 집합을 의미한다. C. Lee[6] 등은 태스크가 선점될 때 캐시에 의해 발생하는 선점 비용(preemption cost)를 고려한 최악 응답 시간 분석 기법을 제안하였다. 식 2는 [6]에서 제안한 태스크 최악 응답 시간을 나타내며, 이때 $PC_i(R_i)$는 τ_i에 의해 발생한 총 선점 비용을 나타낸다.

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + PC_i(R_i) \qquad (2)$$

본 논문에서는 최악 응답 시간 분석 기법을 바탕으로 demand paging 환경에서의 태스크 최악 응답 시간을 제안한다. demand paging 환경에서는 각 태스크가 페이지 폴트(page fault)를 처리하는데 걸리는 시간에 따라 태스크 최악 응답 시간이 결정된다. 따라서 각 태스크에 대한 최악 demand paging 비용을 계산하고, 이를 최악 응답 시간 분석 기법과 결합하는 방법이 필요하다.

3. 최악 demand paging 비용 분석

본 장에서는 demand paging 기반의 고정 주기 실시간 시스템에서 demand paging 비용을 고려한 최악 응답 시간 분석 기법을 제안한다.

3.1. 페이지 순서 분석(page sequence analysis)

페이지 순서(page sequence)란 임의의 태스크 인스턴스가 해당 주기동안 수행될 때 적재될 수 있는 페이지의 집합을 의미한다. 태스크 τ_i가 N_{ps}개의 페이지 순서를 가진다고 할 때, τ_i의 j번째 페이지 순서는 $PS_{i,j}$로 나타낸다 (j=1, 2, 3, ..., N_{ps}). 페이지 순서는 실행 경로에 따라 달라지기 때문에, 본 논문에서는 흐름 제어 그래프(control flow graph: CFG)을 사용하여 태스크 τ_i의 페이지 순서를 분석한다. CFG는 기본 블록(basic block) 단위의 프로그램 흐름을 나타낸 그래프로, 기본 블록을 노드(node)로, 기본 블록 간의 전이를 간선(edge)으로 표현한다. 태스크 τ_i에 속하는 페이지 순서를 구하는 과정은 다음과 같다. 먼저 τ_i에 대한 CFG로부터 τ_i의 모든 실행 경로를 나열하고, 각 실행경로에 속한 기본 블록에 대해, 해당 기본 블록이 속한 페이지를 연결하면 특정 실행 경로에 대한 페이지 순서를 얻을 수 있다. 이를 모든 실행 경로에 대해 반복하면 τ_i의 모든 페이지 순서를 파악할 수 있다. 그림 1은 태스크 τ_i에 속한 페이지 순서를 얻는 과정을 보여준다.

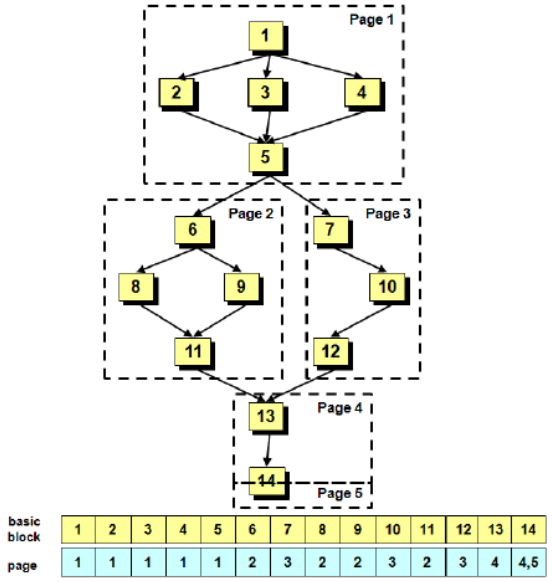

그림 1. CFG 상에서의 페이지 순서

예를 들어, 그림 1에서 $\omega_{0,0} = \{1, 2, 5, 6, 8, 9, 11, 13, 14\}$이므로, $PS_{0,0}$는 $page(w_{0,0}) = \{1, 2, 4, 5\}$와 같다. τ_i의 j번째 실행 경로를 $\omega_{i,j}$로 나타낼 때, $PS_{i,j}$는 식 3과 같이 나타낼 수 있다 (단, $w_{i,j} = \{0, 1, ..., k\}$).

$$PS_{i,j} = page(\omega_{i,j}) = \{page(0), page(1), ..., page(k)\} \qquad (3)$$

식 3에서 $page(w_{i,j})$는 $w_{i,j}$에 속한 페이지의 집합을 의미하며, $w_{i,j}$는 각각 기본 블록 0, 1, ..., k까지 구성된다고 가정한다. 단, 그림 1의 기본 블록 15와 같이, 특정 기본 블록이 여러 페이지에 포함될 수 있기 때문에, $page(k)$는 1개 이상이 존재할 수 있다.

3.2. 페이지 재사용 분석(page reuses analysis)

demand paging 환경에서는 이전에 수행된 태스크 인스턴스가 적재한 페이지 때문에 특정 태스크 인스턴스의 demand paging 비용이 줄어들 가능성이 있다. 이러한 페이지 재사용은 두 가지 형태로 발생할 수 있다. 먼저 여러 실행 경로(i.e., $w_{i,j}$) 간에 동시에 적재되는 페이지와 같이 태스크 내의 태스크 인스턴스 간에 발생할 수 있는데, 본 논문에서는 이를 태스크 내 페이지 재사용(intra-task page reuses)로 정의한다. 반면 공유 라이브러리를 사용하는 경우와 같이, 여러 태스크 간에도 페이지 재사용이 발생할 수 있다. 본 논문에서는 이를 태스크 간 페이지 재사용(inter-task page reuses)로 정의한다.

페이지 재사용 분석의 목적은 각 태스크 인스턴스간에 재사용되는 페이지를 고려하여 태스크 인스턴스 별 최악 demand paging 비용을 구하는 데 있다. 본 논문에서는 그래프 이론을 바탕으로 한 페이지 재사용 분석 기법을 제안한다. 제안하는 기법은 태스크내 페이지 재사용 및 태스크간 페이지 재사용 모두를 고려한다.

그림 2는 태스크내 페이지 재사용 분석 과정을 보여준다.

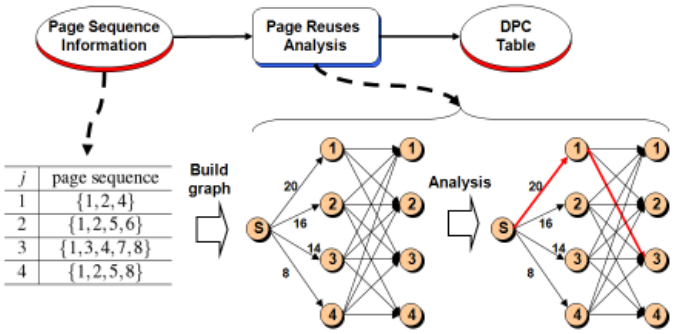

그림 2. 태스크내 페이지 재사용 분석 과정

먼저 페이지 순서 분석 단계로부터 얻어진 각 태스크에 속하는 페이지 순서 정보를 이용하여 그래프를 생성한다. 그래프에서 각 노드는 페이지 순서를 나타내며, 각 열은 태스크 인스턴스의 순서를 뜻한다. 그리고 첫 번째 노드 'S'는 그래프의 논리적인 시작을 의미하고, 간선의 가중치는 특정 태스크 인스턴스에서 특정 실행 경로에 대한 최악 demand paging 비용을 나타낸다. 첫 번째 단계에서 생성된 그래프를 사용하여 페이지 재사용 양상을 분석하고, DPC(demand paging cost) 테이블을 생성한다. DPC 테이블 내의 원소 $\theta_{i,j}$는 태스크 인스턴스 $\tau_{i,j}$에 대한 최악 demand paging 비용을 나타낸다. DPC 테이블은 이후 태스크 최악 응답 시간을 계산하는데 사용된다. 그림 3은 DPC 테이블의 구성을 나타낸다.

Tasks	Worst-case demand paging cost			
τ_1	$\theta_{1,1}$	$\theta_{1,2}$	$\theta_{1,3}$	...
τ_2	$\theta_{2,1}$	$\theta_{2,2}$	$\theta_{2,3}$	...
τ_3	$\theta_{3,1}$	$\theta_{3,2}$	$\theta_{3,3}$	...
...			...	
τ_n	$\theta_{n,1}$	$\theta_{n,2}$	$\theta_{n,3}$	...

그림 3. DPC(demand paging cost) 테이블

예를 들어, 태스크내 페이지 재사용 양상을 고려한 태스크 인스턴스 $\tau_{i,1}$, $\tau_{i,2}$, $\tau_{i,3}$의 최악 demand paging 비용은 각각 $\theta_{i,1}$, $\theta_{i,2}$, $\theta_{i,3}$이다. 그림 4는 $\theta_{i,j}$를 계산하는 과정을 보인다.

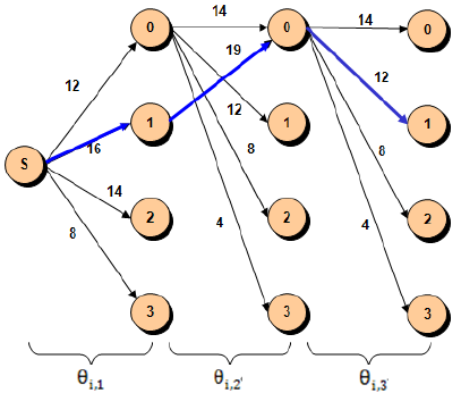

그림 4. 태스크내 페이지 재사용 분석 과정

그림 4에서 $\tau_{i,1}$은 1번 경로를 수행할 때 demand paging 비용이 최대가 되며, 이때의 $\theta_{i,1}$의 값은 16이다. $\tau_{i,2}$는 $\tau_{i,1}$의 실행 경로에 해당하는 $w_{i,1}$로부터 최악 demand paging 비용을 갖는 실행 경로를 취한다. 이 경우, $\tau_{i,2}$는 실행 경로가 $w_{i,0}$일 때 최대 demand paging 비용을 가지며, 이때 $\theta_{i,2}$의 값은 19이다.

즉, $\theta_{i,k}$는 $\theta_{i,1}$로부터 재귀적으로 계산할 수 있다. 먼저 $\theta_{i,1}$은 식 4와 같이 정의된다.

$$\theta_{i,1} = max\{n(PS_{i,j})\} \times \pi \qquad (4)$$

위 식에서 π는 최악 페이지 폴트 지연시간(page fault delay)를 나타낸다. 또한 $\theta_{i,k}$는 식 5와 같이 정의된다.

$$\theta_{i,k} = max\{n(PS_{i,j} - \Pi_{i,k-1})\} \times \pi \qquad (5)$$

태스크내 페이지 재사용 뿐만 아니라 태스크간 페이지 재사용을 고려하기 위해서는, 다른 태스크의 인스턴스가 이전에 적재한 페이지를 고려해야 한다. 그러나 특정 태스크 인스턴스 이전에 실제로 수행된 태스크 인스턴스들을 알 수 없기 때문에, 본 논문에서는 최악의 경우를 가정하여 $\tau_{i,j}$의 최악 demand paging 비용을 계산한다. 최악의 경우는 $\tau_{i,j}$에서 페이지 폴트가 발생하기 전에 상위 태스크 인스턴스에 의해 페이지 폴트가 모두 발생하는 것이다. 그림 5는 태스크간 페이지 재사용을 고려할 때, 태스크 집합이 최악 demand paging 비용을 가지는 경우를 보인다.

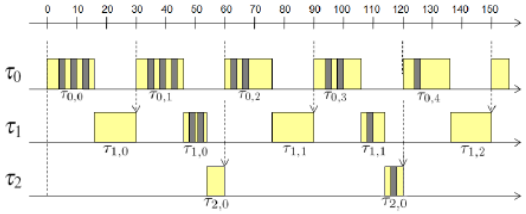

그림 5. 태스크간 페이지 재사용을 고려한 최악 태스크 배치

태스크간 페이지 재사용을 고려한 태스크 인스턴스 $\tau_{i,j}$의 demand paging 비용 $\theta_{i,k}$는 식 6과 같다.

$$\theta_{i,k} = max\{n(PS_{i,j} - \Pi_{i,k-1} - \bigcup_{l=0}^{i-1} \Pi_{l,k}\left[\frac{T_i}{T_l}\right])\} \times \pi \qquad (6)$$

3.3. 태스크 최악 응답 시간 계산

demand paging 환경에서의 태스크 최악 응답 시간을 계산하기 위해서는, 태스크내 페이지 재사용 및 태스크간 페이지 재사용을 고려한 최악 demand paging 비용을 계산하고 이를 최악 응답 시간 식에 추가해야 한다. 식 7는 본문에서 제안한 최악 응답 시간 수식을 나타낸다.

$$R_i^{k+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_i + DPC_i(R_i^k) \qquad (7)$$

식 5에서 $DPC_i(R_i^k)$는 주기 R_i동안 수행된 태스크 τ_i와 τ_i보다 높은 우선순위를 갖는 태스크에 의해 발생한 총 demand paging 비용을 나타낸다. $DPC_i(R_i^k)$의 정의는 식 6과 같다.

$$DPC_i(R_i^k) = \sum_{j=1}^{i} \sum_{l=1}^{g_j} \theta_{j,l} \quad (8)$$

식 8에서 $\theta_{j,l}$은 태스크 $\tau_{j,l}$의 최악 demand paging 비용을 나타내며, 페이지 재사용 분석 과정을 수행한 후 얻어진 DPC table로부터 얻어진다. g_j는 R_i^k 단계로부터 얻어진 τ_j의 최대 요청 비율을 나타내는데, 식 9를 만족한다.

$$g_j \leq \left\lceil \frac{R_i^k}{T_j} \right\rceil \quad (9)$$

4. 실 험

4.1. 실험 환경

본 논문에서는 제안한 기법의 정확성을 검증하기 위해 최악 응답 시간 분석 도구를 구현하여 시뮬레이션을 수행하고, 이를 demand paging이 구현된 임베디드 보드 환경에서의 실측 결과와 비교하였다.

그림 6은 최악 응답 시간 분석 도구를 통한 시뮬레이션 과정을 보인다. 최악 응답 시간 분석 도구는 정적 프로그램 분석기와 응답 시간 분석기로 구성되며, 태스크 집합 (i.e., 실행 가능한 바이너리)을 입력으로 받아 demand paging 비용을 고려한 최악 태스크 응답 시간을 출력한다. 정적 프로그램 분석기는 기존의 링크 시간 최적화 도구중 하나인 Diablo [9]를 기반으로 구현하였으며, 페이지 순서 분석과 페이지 재사용 분석을 수행한 후 DPC table을 출력한다. 응답 시간 분석기는 DPC table로부터 태스크 집합 내에서 가장 낮은 우선순위를 갖는 태스크의 최악 응답 시간을 계산한다.

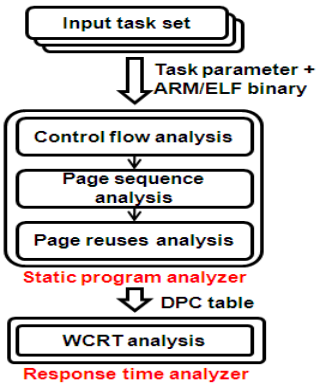

그림 6. 최악 응답 시간 분석기 구조

실측 하드웨어 환경은 ARM920T 코어 기반의 S3C2410 프로세서, 64MB SDRAM과 1GB NAND 플래시 메모리가 장착된 삼성 Reindeer 개발보드를 사용하였다. 운영체제는 임베디드 단말 기기에 널리 사용되는 상업용 운영체제중 하나인 Nucleus Plus [8]를 사용하였다. 단, Nucleus Plus에는 demand paging 기법이 구현되어 있지 않기 때문에, Nucleus Plus 내에 two-level pseudo-LRU [19] 교체 알고리즘 기반의 demand paging 모듈을 구현하였다.

제안한 기법의 정확성을 평가하기 위해서는 벤치마크 프로그램이 많은 코드 페이지를 가지며, 다양한 접근 패턴을 가져야 한다. 그러나 기존의 벤치마크 프로그램 (e.g., MiBench)들은 대부분 코드 페이지의 개수가 10개 이하이고, Nucleus Plus 상에서 동작하기 위해 다양한 루틴 (i.e., C 라이브러리 호출, 파일 접근, 메모리 할당 등)을 Nucleus Plus API를 사용하여 수정해야 하는 문제가 있다. 따라서 본 논문에서는 합성 (synthetic) 벤치마크 프로그램을 사용하여 입력 태스크 집합을 구성하였다. 표 1은 실험에 사용한 합성 벤치마크 프로그램을 나타낸다.

표 1. 합성 벤치마크 파라미터

Task	Code size (KB)	Page reuses ratio	C_i (us)
task 1	81992	10%	340
task 2	81992	21%	1230
task 3	82084	30%	1380
task 4	81920	40%	1750
task 5	81920	52%	1500
task 6	81968	60%	3020
task 7	81932	70%	2900
task 8	81936	80%	2400
task 9	81900	90%	3005

표 1에서 페이지 재사용 비율 (page reuses ratio)는 태스크 내의 페이지 순서가 얼마나 중첩되는지를 나타내는 비율로, 태스크 τ_i의 페이지 재사용 비율은 식 10과 같이 정의된다.

$$page\ reuses\ ratio_i = \frac{\sum_{k=1}^{N_{pages}} f(k)}{N_{pages}} \quad (10)$$

위의 식에서 N_{pages}는 τ_i에 속한 총 페이지의 수를, $f(k)$는 페이지 k가 속한 페이지 순서의 수를 나타낸다. 페이지 재사용 비율이 0%일 경우 어떤 실행 경로가 수행되더라도 중복되는 페이지를 적재하지 않는다는 뜻이며, 페이지 재사용 비율이 100% 인 경우는 모든 실행 경로가 한 페이지 순서에 포함되는 것을 뜻한다. 본 실험에서는 페이지 재사용 비율을 10%에서 90%까지 변화시켜 실험하였다.

표 2는 표 1의 합성 벤치마크 프로그램을 사용하여 구성한 태스크 집합을 나타낸다.

표 2. 태스크 집합 파라미터

Task set	Task	Shared pages	C_i (us)	T_i (us)
Task set 1	task 1		340	8000
	task 5		1500	2400
	task 8		2400	40000
Task set 2	task 2		1230	20000
	task 3		1380	40000
	task 4		1750	60000
Task set 3	task 1	A	760	18000
	task 5	A,B	2500	32000
	task 8	A,B	3500	40000
Task set 4	task 2	A, B	2250	40000
	task 3	B	2075	80000
	task 4	A, B	2905	120000

태스크 집합 파라미터는 태스크 집합에 포함된 합성 태스크 이름(task), 공유 페이지 여부(shared pages), 최악 실행 시간 (C_i), 주기(T_i)로 구성된다. 본 논문에서는 여러 태스크 간에 공유하는 페이지가 있는 경우를 고려하기 위해 각각 5개씩의 공유 페이지를 포함하는 A, B 공유 페이지 집합을 정의하였다. 본 실험에서는 시뮬레이션 및 실측 환경에서 표 2의 태스크 집합을 10 주기동안 수행시킨 후, 최악 응답 시간과 메모리 사용량을 비교하였다.

4.2. 실험 결과

표 3, 표 4는 표 2에서 정의한 태스크 집합을 10 주기동안

표 3. 공유 페이지가 없는 다중 태스크의 실행 결과

Task set	Response time (us)			Memory usage (pages)		
	Naive approach	Our approach	Measured	Naive approach	Our approach	Measured
Task set 1	706550 (473.1%)	153690 (102.9%)	149330	370 (596.8%)	62(100.0%)	62
Task set 2	671850 (606.5%)	141185 (127.4%)	110780	350 (625.0%)	63(112.5%)	56

표 4. 공유 페이지가 있는 다중 태스크의 실행 결과

Task set	Response time (us)			Memory usage (pages)		
	Intra task only	Inter+Intra task	Measured	Intra task only	Inter+Intra task	Measured
Task set 3	225560 (117.9%)	198635 (103.8%)	191277	88 (122.2%)	73 (101.3%)	72
Task set 4	228465 (146.7%)	201540 (129.4%)	155723	87 (131.8%)	72 (109.0%)	66

수행한 결과를 나타낸다. 수행 결과는 최악 응답 시간 (response time)과 메모리 사용량 (memory usage)으로 나누어 비교하였다. 표 안의 비율은 실측 값에 대한 초과 비율을 의미하는데, 비율이 높을수록 실측 값에 비해 과예측 (over-estimated)되었음을 뜻한다. 또한 실측 값 (measured)은 실제 하드웨어 환경에서 수행한 시간(us)을 나타낸다.

표 3은 공유 페이지 (i.e., A, B)가 없는 태스크 집합 (task set 1, task set 2) 수행 결과를 나타낸다. 최악 응답 시간의 경우, naive 기법은 실측 값에 비해 평균 500% 이상 과예측된 반면 제안한 기법은 평균 110% 정도 과예측된 것으로 나타났다. 이는 naive 기법에 비해 제안한 기법에서 태스크간의 페이지 재사용을 고려하기 때문이다. 표 3의 메모리 사용량 결과를 보면, naive 기법의 경우 페이지 재사용을 전혀 고려하지 않기 때문에 적재된 페이지가 약 500% 이상 과예측된 반면, 제안한 기법에서는 거의 정확하게 예측한 것을 확인할 수 있다.

표 4는 공유 페이지가 존재할 경우 태스크 집합 (task set 3, task set 4) 수행 결과를 나타낸다. 먼저 최악 응답 시간 결과를 보면, 태스크내(intra-task) 분석만 수행한 경우 각각 117.9%, 146.7% 과예측 된 것으로 나타난 반면, 태스크간 (inter-task) 분석까지 수행한 경우에는 각각 103.8%, 129.4%로 예측의 정확도가 증가한 것을 확인할 수 있다. 이는 태스크 내 분석의 경우에는 공유 페이지에 의한 페이지 재사용을 고려하지 않는 반면, 태스크간 분석에서는 이러한 공유 페이지를 고려하기 때문이다. 표 4의 메모리 사용량은 태스크내 분석에 비해 태스크간 분석까지 수행할 경우, 페이지 적재 정도를 더 정확하게 분석함을 보여준다. 단, 제안한 기법은 태스크 내에 수행 불가능한 경로 (infeasible path)에 대해서도 페이지 적재를 고려하기 때문에, 수행 불가능한 경로가 많을수록 실측된 값에 비해 최악 응답 시간의 오차율이 커질 수 있다는 단점이 있다.

본 실험 결과를 통해, 제안한 기법이 페이지 재사용을 고려하지 않는 naive 기법에 비해 더 정확하며, 태스크 간의 공유 페이지가 존재할 경우에도 태스크간 분석을 통해 실측 값에 비해 적은 오차율로 최악 응답 시간을 예측할 수 있음을 알 수 있다.

5. 결론 및 향후 과제

본 논문에서는 고정 주기 실시간 시스템에서 demand paging 비용을 고려한 태스크 최악 응답 시간 분석 기법을 제안하였다. 제안한 기법은 각 태스크의 실행 경로와 이전 태스크 인스턴스에 의한 페이지 재사용 양상에 따라 demand paging 비용이 달라진다는 점을 고려한다. 분석 과정은 페이지 순서 분석 단계와 페이지 재사용 분석 단계로 이루어진다. 실험 결과 제안한 기법은 실측값과 비교하여 최대 30% 이내의 오차로 최악 응답 시간을 예측하였다.

본 논문에서 제안한 기법은 다음과 같은 개선점이 있다. 첫째, 실행 불가능한 경로를 제거함으로써 페이지 재사용 분석 기법의 정확도를 높이는 것이다. 둘째, LRU와 같은 페이지 교체 정책을 고려하도록 제안한 기법을 확장함으로써 demand paging을 사용하는 실시간 시스템에서 실제 사용 가능하도록 할 수 있다. 이러한 향후 연구는 이후 실시간 시스템에서의 prepaging 기법 등으로 확장될 수 있다.

감사의 글

이 연구를 위해 연구장비를 지원하고 공간을 제공한 서울대학교 컴퓨터연구소에 감사드립니다. 이 논문은 2008년도 두뇌한국21 사업에 의해 지원되었으며, 2008년도 정부(과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구입니다 (No. R0A-2007-000-20116-0)

참 고 문 헌

[1] C.L. Liu, and J.W. Layland. “Scheduling Algorithms Multiprogramming in a Hard Real-Time Environment”. Journal of the ACM, 20 (1). pages 46-61. 1973.

[2] J. Lehoczky, L. Sha, and Y. Ding. “The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior”. Proceedings of the 10th Real-Time Systems Symposium, pages 166-171, 1989.

[3] K. Tindell, “Adding Time-Offsets to Schedulability Analysis", Technical Report YCS 221, Dept. of Computer Science, University of York, England, January 1994.

[4] Palencia, J. C. and Gonzalez Harbour, M. 1998. “Schedulability Analysis for Tasks with Static and Dynamic Offsets”. In Proceedings of the IEEE Real-Time Systems Symposium, December, 1998.

[5] N.Audsley et al., “Applying new scheduling theory to static priority preemptive scheduling”, Software Engineering Journal, pp. 284-292, September 1993.

[6] C. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim, “Analysis of Cache-related Preemption Delay in Fixed-Priority Preemptive Scheduling,” IEEE Transactions on Software Engineering, pages 264-274, 1996.

[7] A.V. Aho, R. Sethi, and J.D. Ullman, "Compilers, Principles, Techniques, and Tools". Reading, Ma: Addison-Wesley, 1988.

[8] Mentor Graphics, http://www.mentor.com/

[9] Diablo, http://diablo.elis.ugent.be/

[10] L. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer", IBM Systems Journal, vol.5, no.2, pp.78-101, 1966.

[11] Aggarwal, A. and Chandra, A. 1988. "Virtual memory algorithms". In Proceedings of the Twentieth Annual ACM Symposium on theory of Computing (Chicago, Illinois, United

States, May 02 - 04, 1988). STOC'88.

[12] Chang, L., Kuo, T., and Lo, S. 2004. "Real-time garbage collection for flash-memory storage systems of real-time embedded systems". Trans. on Embedded Computing Sys. 3, 4 (Nov. 2004), 837-863

[13] M. D. Bennett and N. C. Audsley. "Predictable and efficient virtual addressing for safety-critical real-time systems". In Proceedings of the 13th Euromicro Conference on Real-Time Systems, pages 183-190, Delft, The Netherlands, June 2001.

[14] T. Geelen. "Dynamic loading in a real-time system: An overlaying technique using virtual memory". Technical report, Philips, Aug. 2005.

[15] I. Puaut and D. Hardy. "Predictable paging in real-time systems: a compiler approach". In Proceedings of the 19th Euromicro Conference on Real-Time Systems, pages 169 - -178, Pisa, Italy, 2007.

[16] D. Hardy and I. Puaut. "Predictable code and data paging for real time systems". In Proceedings of the 19th Euromicro Conference on Real-Time Systems, pages 266-275, 2008.

[17] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. "Timing predictability of cache replacement policies". Real-Time Systems, 37(2):99-122, 2007.

[18] D. Niehaus, E. Nahum, J. Stankovic, and K. Ramamritham. "Architecture and OS support for predictable real-time systems". Technical report, Mar. 1992.

[19] D. P. Bovet and M. Cesati, Understanding the Linux Kernel, Second edition, Oreilly & Associates, 2003.