※ 본 아티클은 CMP MEDIA LLC와의 라이선스 계약에 의해 국문으로 제공됩니다

특집 기사: 누가 그 골대를 옮겼을까? 빠르게 변화하는 CPU의 세계
Sponsored Feature: Who Moved the Goal Posts? The Rapidly Changing World of CPUs

라이언 쉬라우트 및 라이 데이비스(Ryan Shrout & Leigh Davies)
가마수트라 등록일(2009. 10. 19)

http://www.gamasutra.com/view/feature/4168/sponsored_feature_who_moved_the_.php

[본 특집호에서는 인텔의 비주얼 컴퓨팅 마이크로사이트 소속인 쉬라우트와 데이비스가 한 때 CPU 와 일반적으로 컴퓨터 칩에 관한 간단한 법칙을 복잡다단한 시나리오로 탈바꿈시켰던 지난 몇 년에 걸친 프로세서 아키텍처와 설계의 측면에서 발생한 변화의 바람을 고찰한다.]

소개

어떤 플랫폼에서 작업을 하든, 게임 개발이란 젖은 모래 위에 뭔가를 짓는 것과 같다고 할 수 있다. 그러니까 게임을 설계하기 시작한 시간과 그것이 세상에 처음 나온 날 사이의 그 어딘가에 전환을 꾀하고자 구상하고 있던 풍경인 것이다.

이 정도라면 게임 설계에는 단순한 변신 정도이고 완전히 새로운 하드웨어 플랫폼을 향한 보다 과감한 전환에 새로운 그래픽 기법을 포함하는 정도일 수 있겠으나 PC 에 탑재되는 하드웨어는 그 진화가 지속적이다. 말하자면 현장에서 현재의 하드웨어에 맞는 게임 코드를 작성하느라 분주한 가운데 하드웨어 업계는 프로세서 성능을 공격적으로 향상시키고 고객의 PC 에 새로운 기능을 추가시키고 있는 것이다.

최근 2005 년까지만 해도 메인 프로세서의 주된 목표는 세대세대마다 향상된 작동 주파수와 개선된 명령어 수준 병렬성을 제공하면서 단일 쓰레드 성능을 높이는데 있었다. 새로워진 프로세서가 나올 때마다 개발자들은 그런 프로세스가 "그저 빠를 것"이고 그에 따라 코드와 어플리케이션은 확장될 것이고, 심심찮게 출시 시점에 최상급의 PC 가 원래의 코드가 작성된 기반이었던 개발 시스템보다는 빠를 것이라고 가정하며 코드가 설계될 것이라고 생각했다.

그러나 지난 수년에 걸쳐 프로세서 아키텍처와 설계에 분 변화의 바람은 이런 일회성 원칙을 보다 복잡다단한 시나리오로 바꿔놓았다. 인텔 코어 i7 프로세서가 나오면서 게임 개발자들은 이제 한 때 사용 가능했던 컴퓨팅 쓰레드의 수를 최대 8 배로 올릴 수 있다.

프로세서 속도와 기능은 더 이상 선형적으로 증가하지 않으며 터보 부스트 같은 신기술은 시스템 성능에 상당한 영향을 미칠 수 있고 이 때문에 어떤 어플리케이션에도 맞도록 작성된 코드가 프로세서 세대를 거치면서 확장 가능할 뿐 아니라 앞으로 미래의 아키텍처에도 확실하게 대응할 수 있다는 사실은 중요해진다.

일정에 따른 혁신

프로세서 성능의 이런 근본적 변화는 프로세서 아키텍처 세계에서 일어난 숱한 기술적 혁신의 결과다. 지난 3 년 동안 인텔은 적어도 두 번의 주기를 갖는 개선된 성능의 "틱-톡" 모델을 선보여왔다. 즉 "틱"은 설계자가 동일한 물리적 다이 공간에서 더 많은 트랜지스터를 맞출 수 있도록 하는 다이 쉬링크 공정 기술을 도입하는 반면, "톡"은 성능 및 기능면에서의 향상과 함께 새로운 마이크로아키텍처를 선보이고 있다.

"틱"의 시기 동안 프로세스 기술은 2005 년에 도입된 65nm 세대의 제품부터 2007 년의 45nm 세대까지 진화했으며 2009 년 말까지 32nm 이 될 것이다. "톡"의 기간 동안 원래의 인텔 펜티엄 프로세서 아키텍처는 2006 년의 인텔 코어 마이크로아키텍처로 진화해 인텔 코어 i7 가 나오게 되었고 2008 년에는 새로운 마이크로아키텍처를 낳게 되었다. 그림 1 은 이러한 발전상을 도시하고 있다.

오늘날의 uArchitectures

<table>
<tr><td colspan="2">2005-06</td><td colspan="2">2007-08</td><td colspan="2">2009-10</td></tr>
<tr><td>틱</td><td>톡</td><td>틱</td><td>톡</td><td>틱</td><td>톡</td></tr>
<tr><td>Intel® Pentium® D</td><td>Intel® Core™ 2</td><td>Intel® Core™ 2 processors</td><td>Intel® Core™ i7 프로세서</td><td>WESTMERE 프로세서</td><td>SANDY BRIDGE 프로세서</td></tr>
<tr><td colspan="2">65nm</td><td colspan="2">45nm</td><td colspan="2">32nm</td></tr>
<tr><td colspan="6">4 개의 상이한 u-아키텍처

3 개의 상이한 공정 기술

근본적으로 다른 토폴로지</td></tr>
</table>

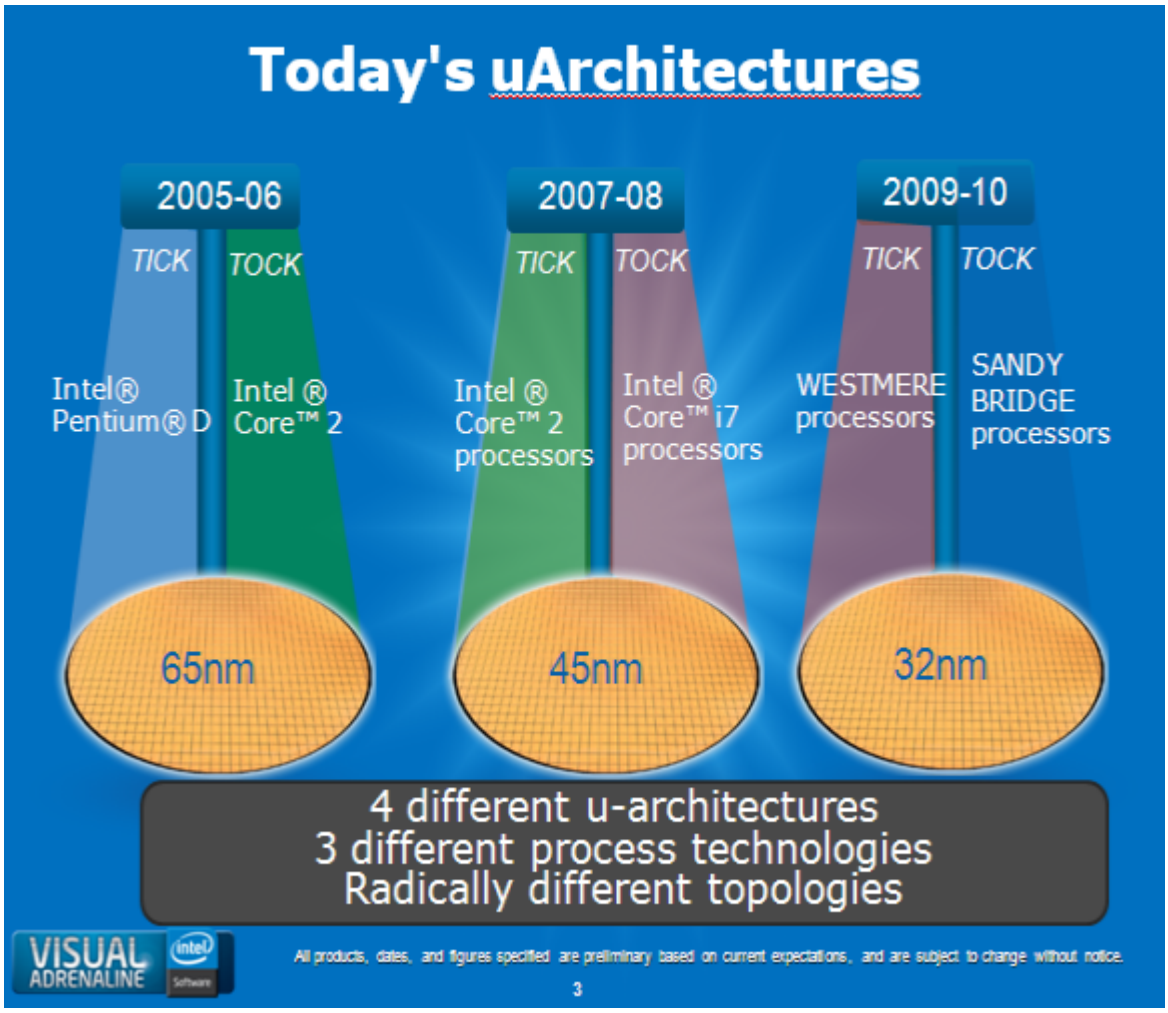


그림 1. 인텔 "틱-톡" 모델의 아키텍처 발달상

앞으로 나올 샌디 브리지 마이크로아키텍처는 2010 년에 선보이게 되며 그림 1 의 다른 아키텍처와 함께 종전 세대의 프로세서와는 극적으로 차이를 보이게 될 것이다. 하나의 아키텍처에 맞게 작성되어 최적화된 코드가 연이은 아키텍처에서 실행된다고 해도, 새로운 프로세서들이 그 코드가 그것을 기반으로 실행하고 그것에 적응하는 하드웨어를 열거할 수 있다고 보장하며 출시되면서 개발자들은 종종 성능 향상에 힘쓰며 잠재적인 성능 함정은 피할 수 있다.

멀티 쓰레드를 염두에 둔 프로그래밍

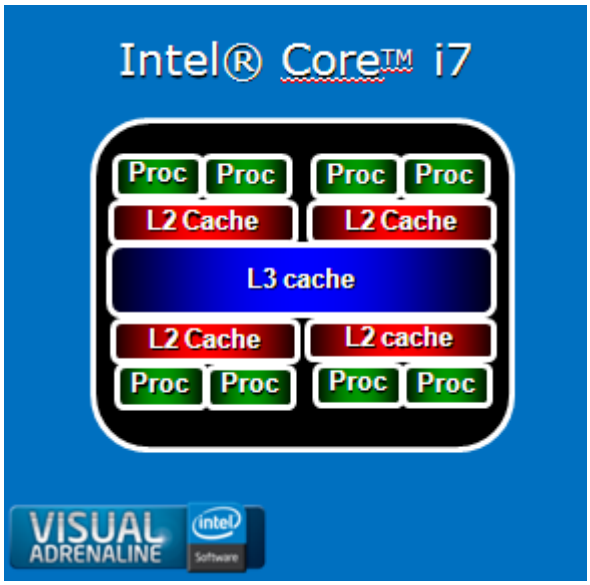
수 년 동안 하드웨어와 소프트웨어 개발 주기는 대개 서로와 무관했다. 컴파일러 및 기타 낮은 수준의 어플리케이션 프로그래머는 플랫폼에서 최고의 성능을 뽑아내기 위해 기저에 하드웨어에 대한 복잡한 지식을 필요로 한 반면, 게임은 코더가 작업에 맞는 최상의 알고리즘을 찾는데 집중하는 동안 하드웨어에 맞는 게임 코드를 최적화하는데 컴파일러에 기대었다. 오늘날 동일 수준의 하드웨어 전문지식이 이제 게임 개발처럼 보다 일반적인 컴퓨팅 작업에 요구된다. 그 이유는 쉽게 이해할 수 있다.

게임 개발에는 평균적으로 1 년 내지 3 년이 소요되고 출하 당시의 제목이 여러 개에 걸칠 것으로 예상 되기 때문에 시중에 더 오래 머물 수 있도록 프로그램되어야 한다. 대규모의 게임

엔진이 구상에서 출하까지 총 개발 기간이 최대 5 년이 소요된다는 것이 흔한 일은 아니다. 인텔이 신속한 개발로 틱-톡 모델을 내놓고 있다는 것은 평균적인 게임 엔진도 다수의 상이한 프로세서 아키텍처에 걸쳐야 할 필요성이 있음을 뜻한다.

(단순히 더 높이 클럭된 프로세서라기보다는) 멀티 코어 프로세서를 도입하는 것은 개발자가 하드웨어를 활용할 수 있도록 확장되는 코드를 작성해야 함을 의미한다. 코어 카운트가 커짐으로써 프로세서 파워를 개선시키는데 초점을 두면서 설계자가 멀티 쓰레딩을 염두에 둔 채 목적을 갖고 소프트웨어를 구축하는 것이 아닌 이상, 하드웨어는 더 이상 기존의 코드를 쉽게 가속할 수 없다. 그림 2 의 인텔 코어 i7 프로세서를 고려해본다. 이는 이러한 프로세서를 충분히 활용하기 위한 것으로 게임이라면 계층적 캐시를 통해 통신하는 8 개의 쓰레드를 사용해야 할 것이다.

<table>
<tr><td colspan="4">Intel® Core™ i7</td></tr>
<tr><td>Proc</td><td>Proc</td><td>Proc</td><td>Proc</td></tr>
<tr><td colspan="2">L2 캐시</td><td colspan="2">L2 캐시</td></tr>
<tr><td colspan="4">L3 Cache</td></tr>
<tr><td colspan="2">L2 캐시</td><td colspan="2">L2 캐시</td></tr>
<tr><td>Proc</td><td>Proc</td><td>Proc</td><td>Proc</td></tr>
</table>



오늘날 이뤄지는 프로그래밍 결정은 향후의 아키텍처를 기반으로 한 게임의 성능에 극적으로 영향을 미칠 것이다. 그래서 프로세서가 지속적으로 빨라지고 그 위력을 더해가고 있지만, 개발자들은 컴퓨팅 파워를 그들의 재량껏 적절하게 사용할 수 있어야 한다.

인텔 아키텍처-계속되는 진화

이런 쟁점을 다루고자 인텔 펜티엄 4 프로세서(그림 3)를 시작으로 가장 최근의 인텔 아키텍처의 진화상을 살펴본다.

Intel® Pentium® 4 프로세서	
42,000K-124,000K 트랜지스터 122mm^2	
클럭 사이클 당 3 개의 명령어	
파이프라인 스테이지 31	
Intel® SSE2	
SIMD 유닛 2 X 64 비트	
추가분	
64 비트 가능, Intel® SSE3	
코어 당 최대 2 개까지의 SMT	

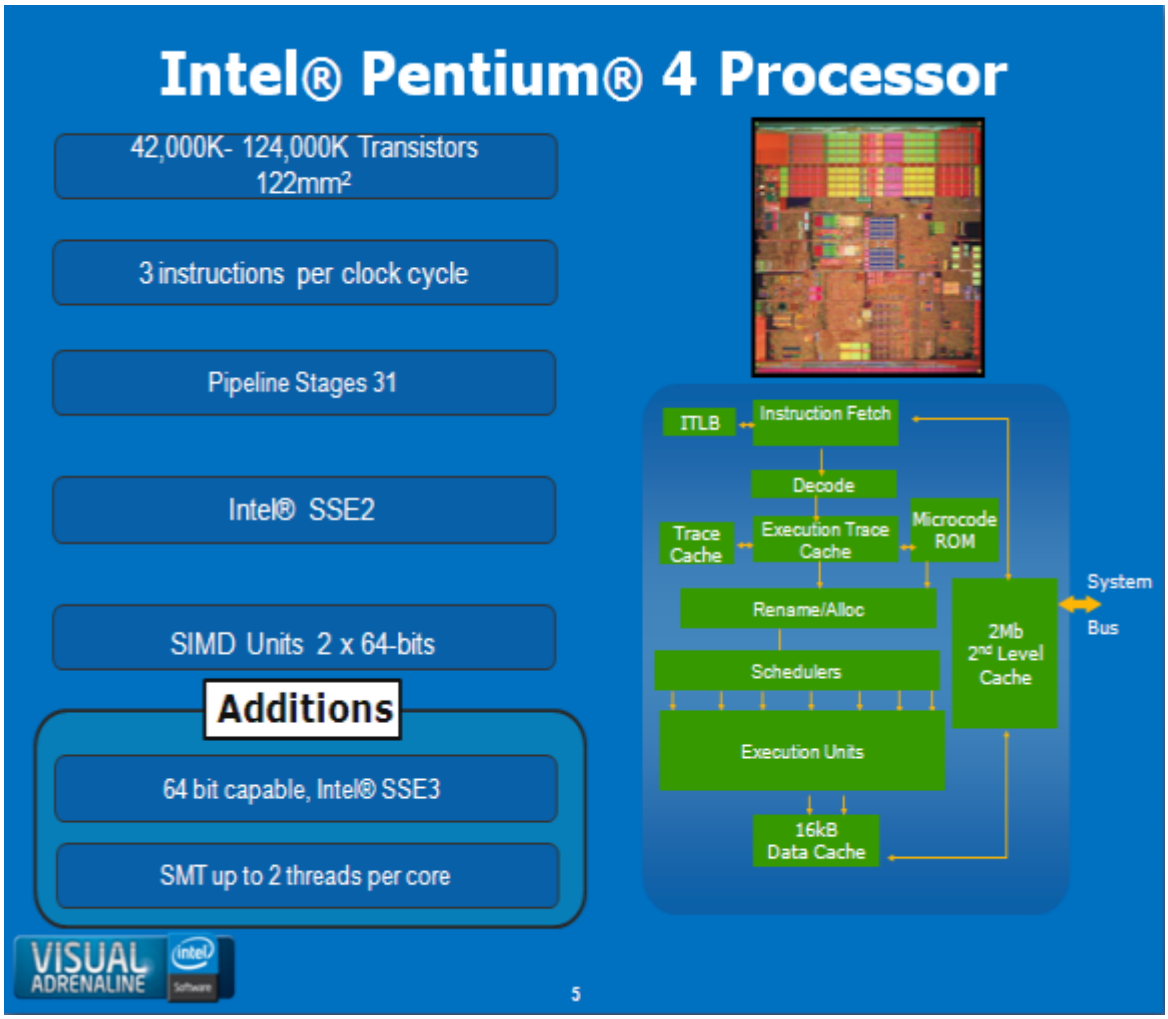


그림 3. 인텔 펜티엄 4 프로세서 마이크로아키텍처의 특성

2005 년 인텔 펜티엄 4 프로세서 아키텍처는 시중에 나온 인텔의 소비자 프로세서 가운데 최상급이었다. 정확한 모델에 따라 4 천 2 백만 개부터 1 억 2 천 4 백만 개에 이르는

트랜지스터를 그 특징으로 하였고 다이의 크기는 약 122nm2 였다. 이 아키텍처에서는 클럭 주기당 3 개의 명령어가 발생될 수 있었고 파이프라인은 후에 총 31 개의 스테이지로 비교적 깊었다.

이 프로세서는 인텔 스트리밍 SIMD 익스텐션 2(인텔 SSE2) 명령어를 통합했고 64 비트의 싱글 인트럭션 멀티플 데이터(SIMD) 유닛을 갖추었다. 따라서 모든 128 비트 SIMD 명령어는 실행하는데 2 번의 주기를 필요로 했다. 이 아키텍처의 수명주기가 끝을 향해갈 무렵 인텔은 64 비트 프로세싱, 인텔 SSE3 인스트럭션 및 동시 다중 쓰레딩(SMT)-인텔 하이퍼-쓰레딩 기술(인텔 HT 기술)을 뒷받침하는 지원을 추가했다. 낮은 수준의 프로그래머 최적화는 분기 예측 행위 개선이나 인텔 SSE 를 이용해 데이터 알고리즘 핫 스팟을 병렬 처리하는데 초점을 맞추는 경향이 있었다.

Intel® Core™ 마이크로-아키텍처	
582,000K 트랜지스터 (듀얼코어), 107mm^2	
820,000K 트랜지스터 (쿼드코어) 214mm^2	
클럭 사이클 당 4 개의 명령어; 14 스테이지 파이프, 마이크로퓨전 및 매크로퓨전	
각 코어쌍이 공유 L2 캐시에 접근 가능하다	
코어 1 당 쓰레드 최대 4 개의 코어	
SIMD 유닛 3* 128 비트 싱글 사이클 SSE	
45NM	
새로운 명령어 SSE4	
SIMD 성능 제고사항	

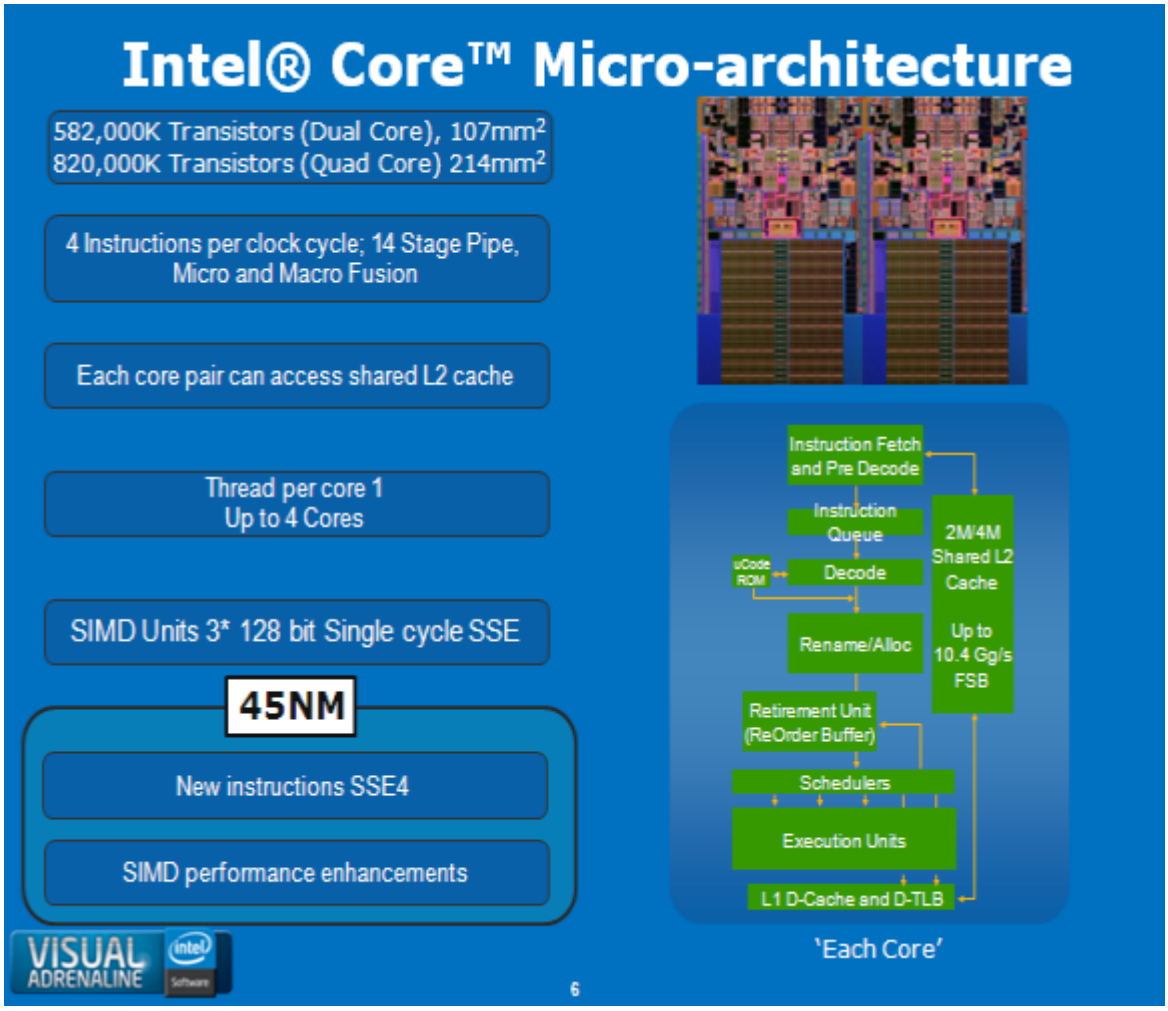


그림 4. 인텔 코어 마이크로아키텍처의 특성

그림 4 는 고성능 멀티코어 프로세서 계열의 토대인 인텔 코어 마이크로아키텍처를 나타내고 있다. 트랜지스터 수는 듀얼 코어 프로세서의 경우 5 억 8 천 2 백만, 쿼드 코어 프로세서의 경우 8 억 2 천만 개인 가운데 인텔 펜티엄 4 프로세서와 비교해 최대 7 배나 증가되었다. 한가지 중요한 발전이라면 클럭 당 이슈된 명령문이 4 개로 증가했고 잘못된 분기 예측이 일어날 때 퍼포먼스 히트를 감소시키는 14-스테이지 파이프라인으로 전환되었다는 사실이다.

이는 또한 쿼드 코어 프로세서가 실제로 상대적으로 더 느린 프론트 사이트 버스에 걸쳐 통신해야 하는 두 개의 공유된 L2 캐시를 갖추고 있는 반면, 싱글 다이에 존재하는 두 개의 코어 사이에 한 개의 공유된 L2 캐시를 갖는 최초의 데스크톱 인텔 아키텍처였다. 게임 개발자들은 그저 다이에서 트랜지스터를 효율적으로 사용하고자 최대 4 개의 프로세싱 쓰레드까지 확장해야 할 필요성에 직면했다. 기능면에서의 병렬성은 사운드와 같은 미들웨어 쓰레드와 추가적인 경량의 운영체제(OS)와 함께 2 개 내지 3 개의 메인 게임 쓰레드에서는 일반적이었다.

인텔 코어 마이크로아키텍처는 코어 수와 보다 효 율적인 파이프라인을 증가시켰을 뿐 아니라 SIMD 유닛의 성능을 두 배로 했는데, 왜냐하면 128 비트 프로세싱은 싱글 프로세서 클럭 주기 상에서 이뤄질 수 있기 때문이다. 45nm 기술을 가져 왔던 "틱"은 인텔 SSE4 인스트럭션 세트 및 강화된 SIMD 설치 속도의 경우 3 배나 더 빨라진 셔플 유닛의 성능 향상 또한 가져옴으로써 부동소수점 집약적 알고리즘을 최적화하기 위한 기능을 증가시켰다.

빠른 SOA(어레이 구조) 프로세싱에 적당하지 않은 데이터 구조 때문에 SIMD 를 사용해 최적화하는데 어려움이 있었던 알고리즘은 훨씬 개선된 셔플 유닛의 성능과 보다 광범위해진 데이터 재정렬 명령어 때문에 실행 가능해졌다.

Intel® Core™ i7 아키텍처	
731,000K 트랜지스터 (4 코어, 8 쓰레드), 263mm^2	
동시적 멀티쓰레딩/ 코어 2 당 쓰레드 데스크톱에서 최대 4 개의 코어	
추가적인 캐싱 계층	
클럭 사이클 당 4 개의 명령어; 16 스테이지 파이프, 강화된 마이크로퓨전 및 매크로퓨전	
더욱 깊어진 버퍼	
SIMD 유닛 3* 128 비트 싱글 사이클 SSE	
32-비트 및 64-비트 모드 모두에서의 마이크로퓨전*	

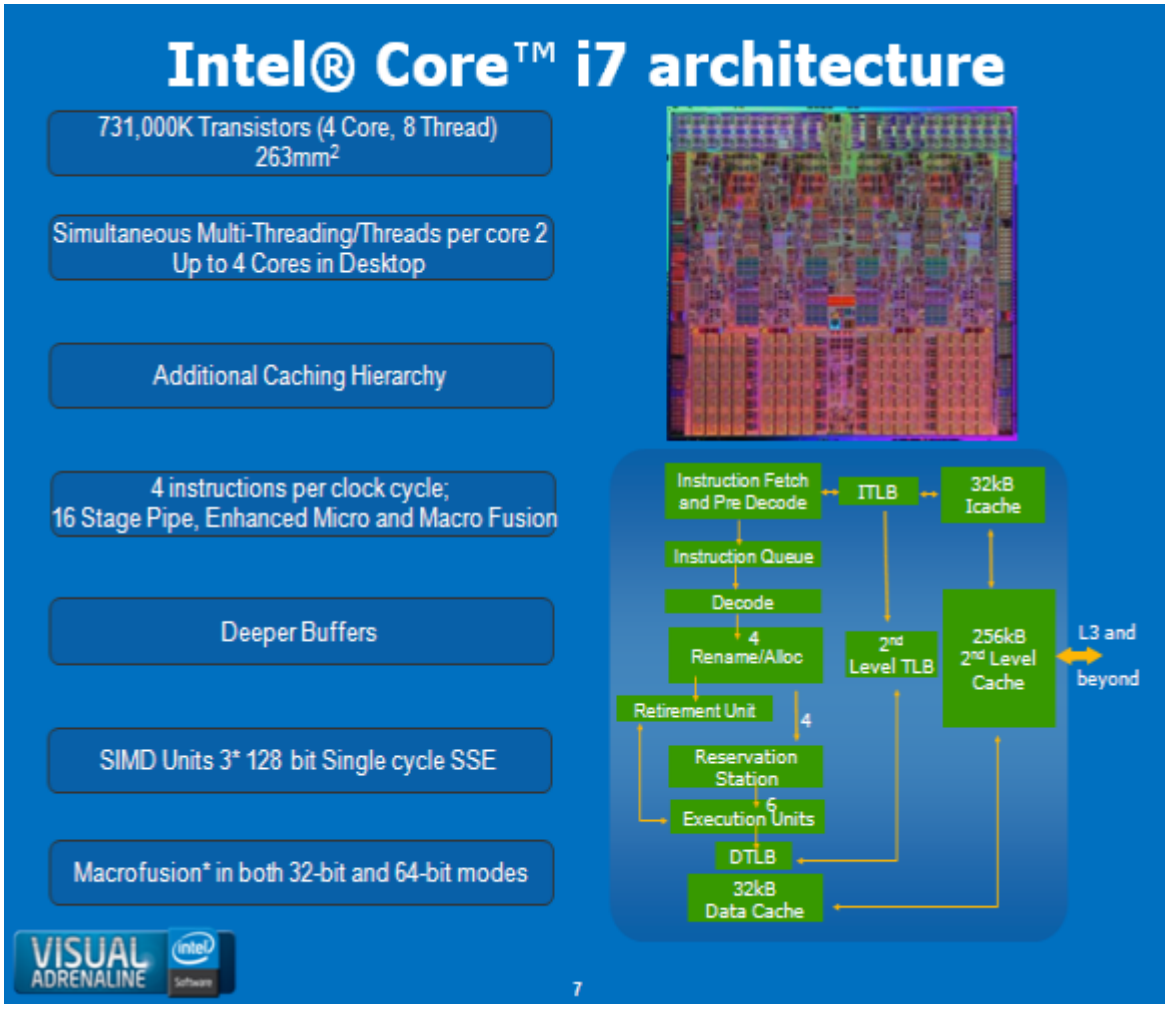


그림 5. 인텔 코어 i7 프로세서의 예시를 통한 차세대 인텔 코어 마이크로아키텍처

2008 년 인텔에서 내놓은 최신 프로세서 마이크로아키텍처는 (그림 5) 인텔 코어 i7 프로세서과 함께 도입되었다. 이것은 인텔이 내놓은 최초의 네이티브한 모놀리식의 쿼드 코어 프로세서를 특징으로 하며 인텔 코어 2 쿼드 프로세서(7 억 3 천백만 대 8 억 2 천만)보다 트랜지스터를 더 적게 사용했다. 펜티엄 4 에서 마지막으로 등장했던 인텔 HT Technology 가 이 쿼드코어 프로세서에 재도입되어 증가된 수의 쓰레드를 지원하기 위한 비순차적 엔진에서의 자원 증가와 함께 최대 8 개의 동시 쓰레드에 대한 인텔 코어 i7 프로세서 지원을 가능케 한다.

트랜지스터의 감소에도 불구하고 다이 크기는 세 가지 수준의 캐시 계층을 포함하는 보다 복잡한 아키텍처 설계 덕분에 263mm 으로 증가되었다. 파이프라인은 두 스테이지만 확장된 반면, 디코더에서의 마이크로퓨전과 매크로퓨전 모두 강화되었다. 인텔 코어 마이크로아키텍처부터 인텔 코어 i7 프로세서 마이크로아키텍처에 이르는 계층적 변화의 정도는 그 차이가 분명하게 나타나는 것은 아니지만 인텔 펜티엄 프로세서 마이크로아키텍처에서 인텔 코어 마이크로아키텍처까지의 변화만큼이나 극적이다.

인텔 코어 i7 아키텍처에 대해

<table>
<tr><td colspan="2">공통 코어</td><td colspan="3">모듈라 언코어</td></tr>
<tr><td>아키텍처 상태</td><td>아키텍처 상태</td><td colspan="2">코어</td><td>언코어</td></tr>
<tr><td colspan="2">데이터 캐시</td><td colspan="3">DRAM</td></tr>
<tr><td rowspan="2">CPU 프론트 엔드</td><td rowspan="2">비순차적
실행엔진</td><td colspan="2">마지막 Leverl 캐시</td><td rowspan="2">Pwr & Clk</td></tr>
<tr><td>IMC</td><td>QPI</td><td>QPI</td></tr>
<tr><td colspan="2">공통 “코어”</td><td colspan="3"></td></tr>
<tr><td colspan="2">서버, 데스크톱, 모바일의 경우 동일한 코어</td><td colspan="3">“언코어”에서 구분됨</td></tr>
<tr><td colspan="2">각 코어는 SMT 를 지원한다</td><td colspan="3">코어의 #</td></tr>
<tr><td colspan="2">코어 2 에 비교되는 아키텍처 개선점</td><td colspan="3">메모리 채널의 #</td></tr>
<tr><td colspan="2"></td><td colspan="3">QPI 링크의 #</td></tr>
<tr><td colspan="2"></td><td colspan="3">캐시 크기</td></tr>
<tr><td colspan="2"></td><td colspan="3">메모리 타입</td></tr>
<tr><td colspan="5">최상의 성능은 게임의 적절한 쓰레딩에서 나온다</td></tr>
<tr><td colspan="5">암달이라면 언제나 낭패를 볼 것이다.</td></tr>
</table>

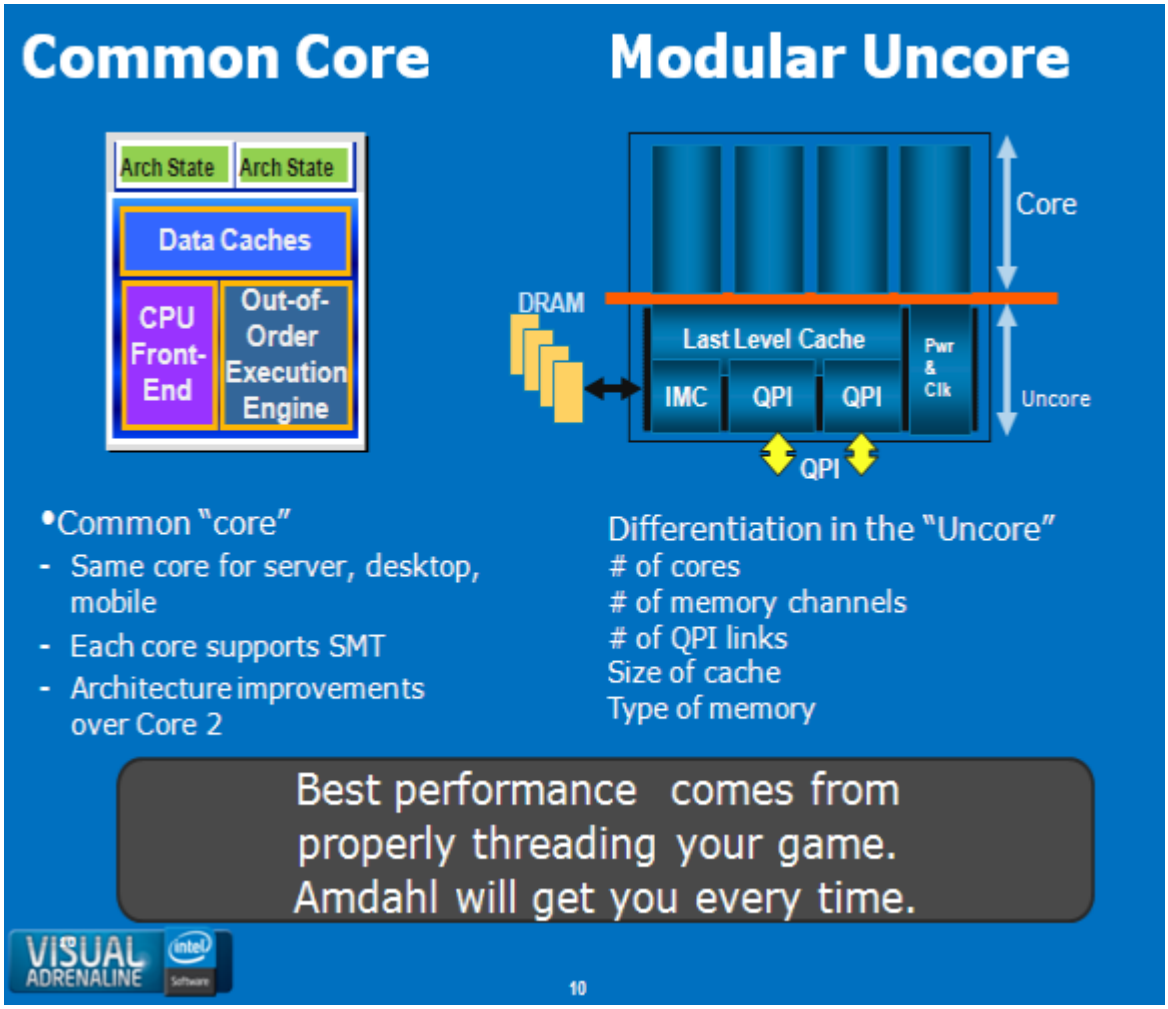


그림 6. 인텔 코어 i7 프로세서 설계의 코어 및 언코어 다이어그램

인텔 코어 i7 프로세서 아키텍처는 철저히 모듈성을 갖추도록 설계됨으로써 프로세서 당 코어의 수가 변할 수 있다. 인텔 코어 i7 프로세서에 탑재한 각각의 코어는 동일하며 정렬되지 않은

로드와 캐시 분할 그리고 종전의 아키텍처와 비교해 개선된 스토어를 처리할 때 더 나은 분기 예측, 향상된 메모리 성능을 지원한다. 각각의 공통 코어는 SMT 를 지원하며 이들 2 개의 레지스터 세트는 L2 및 L1 데이터 캐시 및 비순차적 실행 유닛을 공유한다. 각각의 코어는 해당 OS 에 두 개의 논리 프로세서로써 나타난다.

인텔 코어 i7 프로세서 마이크로아키텍처는 또한 이 프로세서 설계에 언코어 부분을 도입했다. 이 언코어에는 메모리 컨트롤러, 파워 및 클러킹 컨틀로러 같은 기타 프로세서 특성이 담겨 있다. 이것은 공통 코어의 경우와는 다른 주파수에서 동작한다. 이 아키텍처는 단일 언코어 설계에 부착될 코어의 수를 달리할 수 있고 멀티 소켓 서버와 워크스테이션을 고려하기 위해 여분의 QPI 링크를 추가하는 것처럼 언코어가 그것의 의도된 시장에 따라 달라질 수 있도록 지원한다.

인텔 코어 i7 프로세서로부터 최상의 성능 이점을 얻으려면 개발자들은 멀티쓰레딩을 공격적으로 구현해야 한다. 4 개 이하의 쓰레드용으로 작성되었던 게임 엔진은 가용한 프로세싱 파워의 일부만을 활용하게 되고 종전의 쿼드 코어 프로세서보다 제한된 성능 이득만을 보게 될 것이다. 개발자들은 기능적 병렬성을 넘어야 하고 단순히 물리 또는 AI 를 가르는 수준을 넘어 대신 다수의 쓰레드 전반에 걸쳐 분배되는 개별 기능을 이용해 데이터 수준의 병렬성으로 전환해야 한다.

그래픽 집약적인 게임의 핵심 요건 가운데 한 가지는 렌더링 엔진 자체가 다수의 쓰레드 전반에 걸쳐 확장되도록 하는 것이고 게임 및 비디오 카드 사이의 그래픽 파이프라인은 프로세서 및 그래픽 프로세싱 유닛(GPU) 모두의 사용을 제한하는 병목이 되지 않는다. DirectX 11 에서의 멀티 쓰레드 방식의 렌더링 컨텍스트 지원이 이런 게임 프로그래밍 영역에 크게 유용하길 바란다.

<table>
<tr><td colspan="5">크고 강해진 메모리 시스템</td></tr>
<tr><td>32kB L1 데이터 캐시</td><td>32kB L1 Inst. 캐시</td><td></td><td>32kB L1 데이터 캐시</td><td>32kB L1 Inst. 캐시</td></tr>
<tr><td colspan="2">코어</td><td></td><td colspan="2">코어</td></tr>
<tr><td colspan="2">통합 256kB 8 Way L2 캐시 10 Cycle</td><td></td><td colspan="2">통합 256kB 8 Way L2 캐시 10 Cycle</td></tr>
<tr><td colspan="5">포괄적 L3 캐시 (35-40 번의 사이클)

다이 상의 스누프 필터라는 이점을 제공한다

다양한 코어 수에 따라 크기를 달리할 수 있도록 구축됨

향후를 고려해 L3 크기를 쉽게 증가할 수 있도록 구축됨</td></tr>
<tr><td>코어 2</td><td>0.6

코어 i7</td><td>IMC</td><td>9776

코어 2</td><td>33376

코어 i7</td></tr>
<tr><td colspan="2">상대적 메모리 지연</td><td></td><td colspan="2">스트림 대역폭-Mbytes/Sec (triad)</td></tr>
</table>

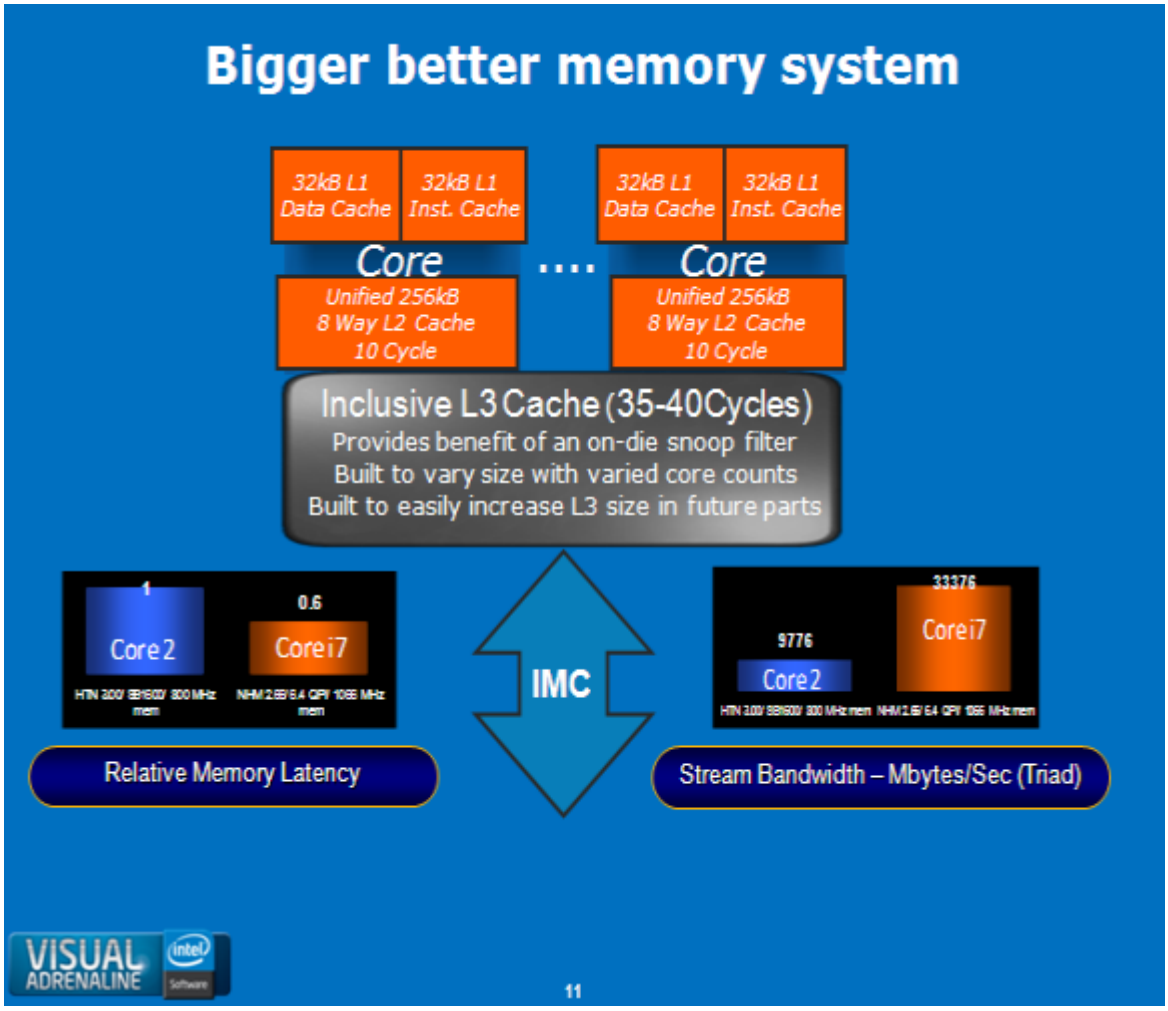


그림 7. 인텔 코어 i7 프로세서의 메모리 계층

그림 7 은 종전의 인텔 데스크톱 설계와는 확연히 다른 인텔 코어 i7 프로세서에서의 메모리 계층을 나타낸다. 각각의 코어에는 매우 빠른 4 개 주기 내에 접근 가능한 32 K L1 캐시가 있다. 인텔의 코어 프로세서 계열과 비교해, 256 K L2 캐시는 10-사이클 액세스 시간보다 훨씬 빠르고 더 높은 클럭율로 실행되기는 하지만, 훨씬 작다. 더 작은 L2 캐시는 언코어에 속하며 모든 공통 코어가 공유하는 훨씬 더 큰 추가적 L3 캐시에 의해 상쇄된다.

이 새로운 L3 캐시는 코어와 언코어 클럭 속도 사이의 가변비율에 따라 35-40 사이클 액세스 시간을 갖는다. L3 캐시는 L1 과 L2 캐시의 데이터를 반영하는 포괄적인 공유 캐시 설계를 갖추고 있다. 이 포괄적 캐시 설계가 처음에는 다이 상의 총 가용 캐시를 덜 효율적으로 사용하는 것처럼 보일 수 있지만, 이 포괄적 설계로 인해 메모리 지연시간이 훨씬 향상된다.

하나의 코어가 그 자체의 로컬 L1 과 L2 캐시 스토어에서 찾을 수 없는 데이터를 필요로 한다면, 캐시 결과에 대한 나머지 코어를 탐사하는 대신, 메인 메모리에 접근하기에 앞서 이 포괄적 L3 캐시를 쿼리하기만 하면 된다. L3 캐시는 새로운 통합형 메모리 컨트롤러에 연결되는데 이것은 코어 2 프로세서에서 사용된 종전의 프론트 사이드 버스보다 훨씬 높은 대역폭을 제공하는 시스템의 DRAM 에 연결된다. 멀티 소켓 시스템의 경우 해당 메모리 컨트롤러는 프론트 사이드 버스를 대신해 퀵패스 인터커넥트를 통해 또 다른 소켓에 통신할 수 있다.

종전의 고사양 데스크톱 시스템 세대와 비교해 메인 메모리의 성능은 대역폭(3 베만큼) 및 지연시간(0.5 배만큼) 모두 증가되었다. 8 개의 컴퓨팅 쓰레드를 실행할 지의 여부를 결정할 때 프로세서 코어가 부족하지 않도록 메모리 대역폭을 가용하게 해 두는 것이 중요핟.

인텔 스트리밍 SIMD 명령어 세트

또한 도입된 인텔 코어 i7 프로세스는 및 다양한 SIMD 명령어 세트로의 확장은 SSE4.2 라 불리는 인텔 프로세서에서 지원한다. 이 새로운 명령어는 특정 알고리즘의 가속화를 겨냥했고 그들 자체에는 (거기서 이득을 보게 되는 알고리즘을 사용하는지의 여부에 따라) 게임의 측면에서 제한된 이점이 있을 수 있는 반면, 일반적으로 SSE 의 지원은 PC 에서 최대한의 것을 얻어내는 매우 중요하다. 게임 개발자에게 종종 놀라운 것이 있다면 바로 종전 버전의 SSE 에 대한 지원이 정말 광범위하다는 사실이다.

인텔 SSE 명령어 세트는 원래 1999 년에 도입되었고 곧 2000 년에 인텔 SSE2 세트가 따라 나왔다(그림 8.). 오늘날 대부분의 게임 개발자는 2000 년 이후 구축된 하드웨어를 목표로 삼고 있으며 따라서 기본적으로 최소한 인텔 SSE2 에 대한 지원을 보장할 수 있다고 생각하는 것이 맞는 듯 보인다. 적어도 듀얼 코어 구성을 최소한의 사양으로써 삼아야 인텔 플랫폼 및 AMD 플랫폼 모두에서의 인텔 SSE3 에 대한 지원이 보장될 것이다.

<table>
<tr><td colspan="6">사용자가 보유한 SSE – 그것을 사용한다</td></tr>
<tr><td>Intel Ⓡ SSE
1999</td><td>Intel Ⓡ SSE2
2000</td><td>Intel Ⓡ SSE3
2004</td><td>IntelⓇ SSSE3
2006</td><td colspan="2">IntelⓇ SSE4</td></tr>
<tr><td>70

명령어

싱글-
정밀벡터

스트리밍 연산</td><td>144

명령어

더블-
정밀벡터

128-비트
벡터 정수</td><td>13

명령어

Complex
Arthmetic</td><td>32

명령어

디코드</td><td>SSE4.1 2007</td><td>SSE4.2 2008</td></tr>
<tr><td></td><td></td><td></td><td></td><td>47

명령어

비디오 가속기

그래픽 빌딩 블록

코프로세서 가속기</td><td>7 instructions

XML 가속기

어플리케이션 전용 가속기</td></tr>
<tr><td colspan="6">밸브 시스템* 조사 보고서 SSE2+의 시스템 가운데 95.72%</td></tr>
</table>

모든 듀얼코어 시스템은 SSE3 또는 그 이상을 지원한다

하드웨어 개선점은 컴파일러에 도움이 된다

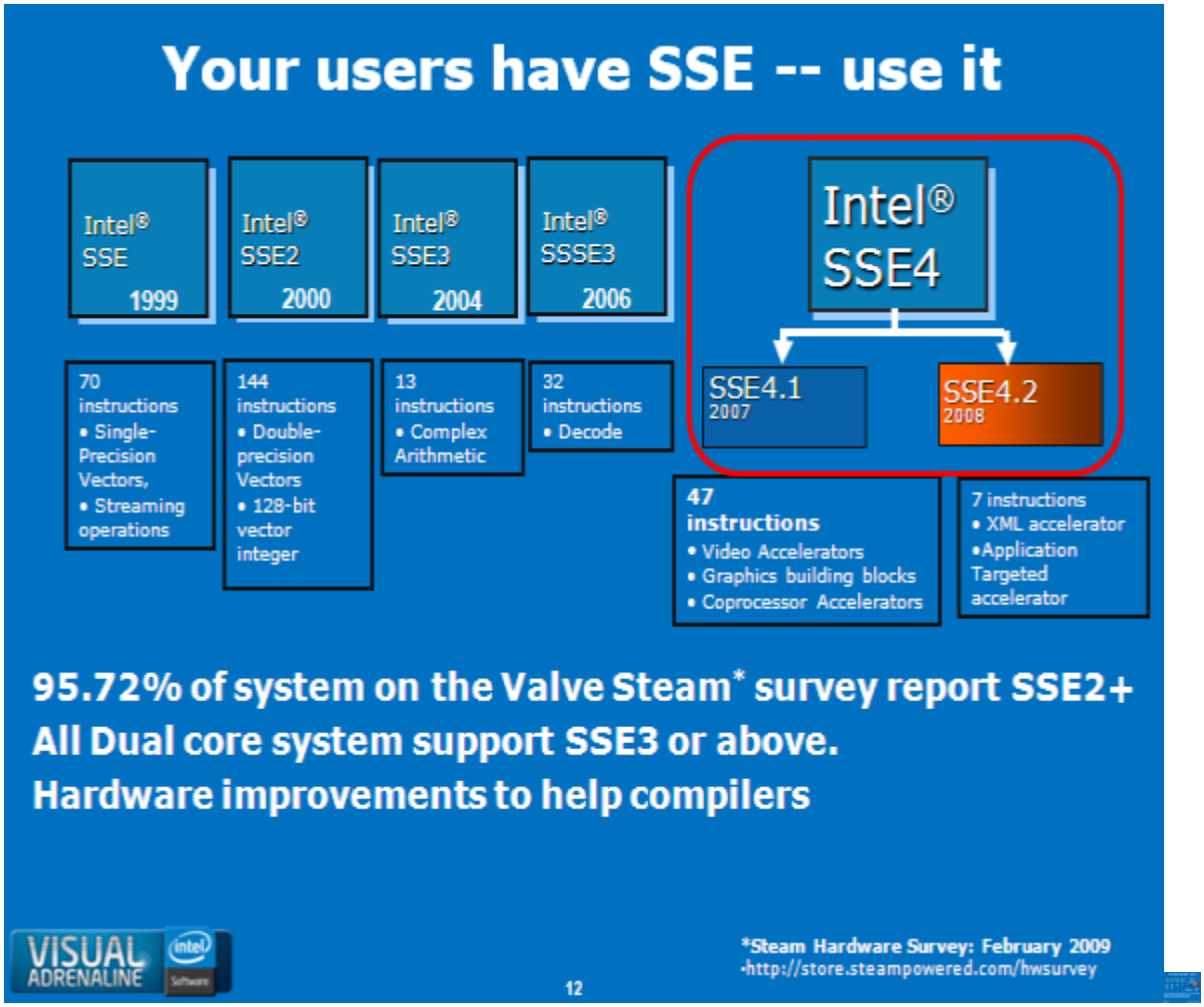


그림 8. 인텔 스트리밍 SIMD 명령어의 진화

직접 코딩 방식의 최적화 외에, 새로운 아키텍처 각각의 목표 가운데 하나는 컴파일러가 자동으로 생성할 수 있는 최적화의 유연성을 높이는 것이다. 정렬되지 않은 로드를 개선시키고 유연 있는 SSE 로드 명령어를 증가시키는 개선 조치를 통해 컴파일 시기에 컨텍스트 정보가 가용한 가운데에서도 종전에는 안전하게 끝날 수 없었던 코드를 컴파일러는 최적화할 수 있다. 따라서 특정 하드웨어를 목표로 하기 위해 설정된 컴파일러 플래그로 성능을 시험해보는 것이 언제나 권장된다. 또한 일부 컴파일러는 다수의 코드 경로가 동시에 생성될 수 있도록 해 목표의 하드웨어에 이르는 신속한 경로와 함께 하위 호환성을 제공한다.

쓰레딩 성능의 고려사항

게임 개발에 있어 보다 중요한 쓰레딩의 측면은 캐시와 다양한 쓰레드와의 상호작용이다. 앞서 언급한 것처럼 인텔 펜티엄 4 프로세서, 인텔 코어 프로세서 계열 및 인텔 코어 i7 프로세서에서의 캐시 구성은 차이가 난다(그림 9.)

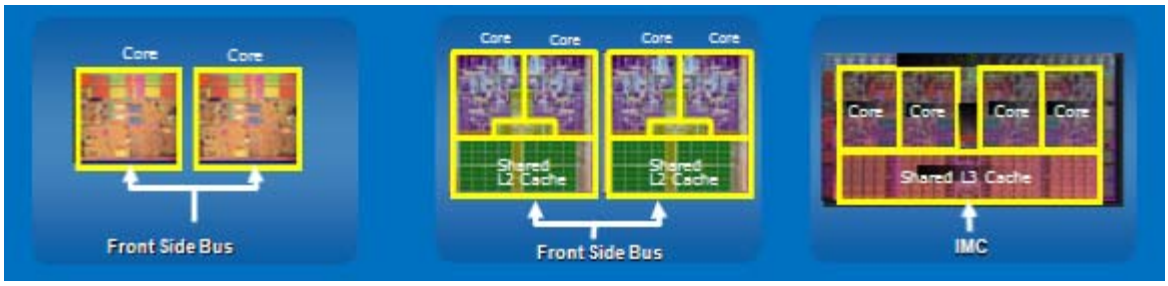


그림 9. 상이한 마이크로아키텍처이서의 캐시 레이아웃

인텔 펜티엄 4 프로세서는 프론트 사이드 버스에 걸쳐 쓰레드 간에 통신해야 하며 따라서 적어도 400-500 사이클 지연을 필요로 하는 반면, 인텔 코어 프로세서 계열에서는 공유된 L2 캐시에 걸쳐 쿼드 코어 설계를 기반으로 한 많은 쌍 사이의 프론트 사이드 버스 및 코어 쌍 간의 20 번 사이클 지연으로 통신이 가능했다. 인텔 코어 i7 프로세서에서 공유 L3 캐시를 사용한다는 것은 다수의 소켓 시스템이 사용되고 있는 것이 아닌 한, 또 다른 코어와의 동기화를 위해 버스를 거치는 것이 불필요하다는 것을 의미한다.

그리고 캐시 크기가 코어의 수만큼이나 빠르게 증가하고 있는 것이 아니기 때문에 개발자들은 가용한 유한 자원을 어떻게 최대한 사용할 것인가에 대해 생각해야 한다. 내부 루프를 더 작은 세트의 데이터와 스트리밍 스토어로 제한하는 캐시 블로킹과 같은 기법은 캐시 라인 오염을 피하는데 도움을 줄 수 있어 캐시 쓰레싱을 줄이고 공유된 캐시의 최종 수준에 걸쳐 요구된 통신을 최소화할 수 있다.

거짓 공유

발생 가능한 중요한 캐시 문제 한가지는 두 개의 쓰레드가 동일한 캐시라인으로 분류되는 데이터에 접근할 때 일어나는 거짓 공유다.

거짓 공유
2 개의 쓰레드가 동일한 캐시 라인으로 분류되는 데이터에 접근할 때 일어날 수 있다.
64-byte aligned global and static
- ___declspec(align(64))
When each thread needs to use its own copy variables
- Use modifier___declspec(thread) - Or use stack variables where reasonable
For dynamic allocation, avoid using malloc and new
- Use routines that allocate from separate 64byte aligned per thread pools

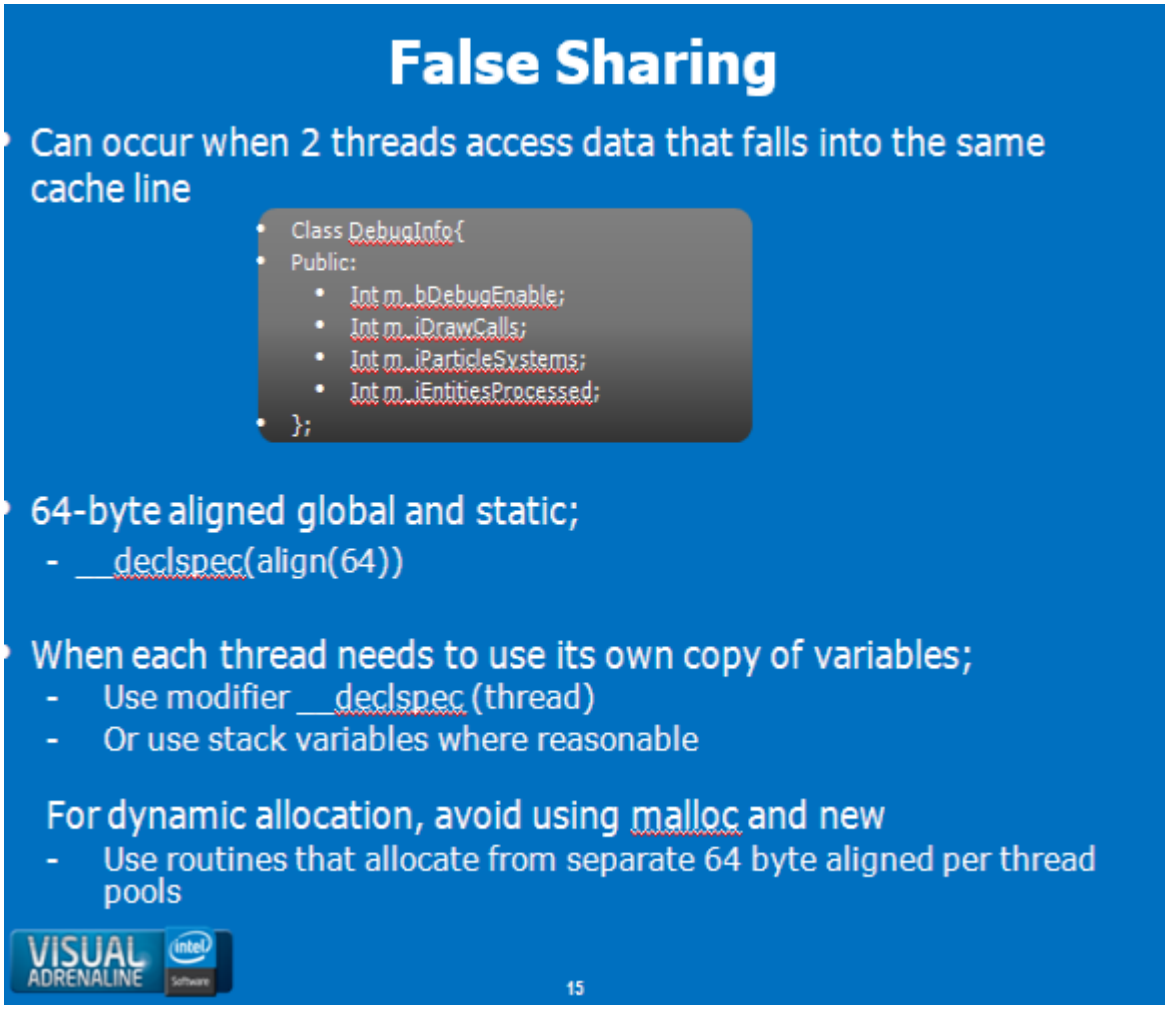


그림 10. 거짓 공유의 예

그림 10 은 거짓 공유에 관한 예시로, 이 경우 해당 개발자는 디버그 데이터를 저장할 수 있는 간단한 전역 객체를 생성했다. 생성된 4 개의 모든 정수가 단 하나의 캐시 라인에 들어맞는다. 그러나 다중 쓰레드는 구조의 서로 다른 부분에 접근할 수 있다. 새로운 드로우 콜을 제시하는 메인 렌더 쓰레드처럼 어떤 쓰레드든 그것이 관련 변수를 갱신할 때, 그것은 그 객체에 있는 정보를 접근하거나 갱신하는 다른 모든 쓰레드에 있어 전체 캐시 라인을 오염시킨다. 이 캐시 라인은 모든 캐시에 걸쳐 다시 동기화되어야 하고 이것이 성능을 떨어뜨릴 수 있다.

그러한 2 개의 쓰레드가 인텔 코어 프로세서 또는 인텔 펜티엄 4 프로세서의 프론트 사이드 버스를 통해 통신 중이라면 해당 사용자는 그 간단한 디버그 코드 업데이트에 대해 무려 400-500 번의 사이클 지연을 경험할 수 있다. 변수의 배치가 일반적인 사용 패턴을 설명할 수 있도록 변수를 배치할 때 주의를 기울여야 한다. 인텔 VTune 퍼포먼스 애널라이저 같은 툴이 거짓 오류 같은 캐시 문제를 찾는데 사용될 수 있다.

기본적인 작업 큐

캐시 설계가 알고리즘의 성능에 얼마나 크게 영향을 미칠 수 있는가에 대한 예시로 기본적인 작업 관리자를 사용할 수 있는데 이것은 종전의 작업을 완료한 이후 워커 쓰레드가 가용할 때 그것에 작업들을 분배한다. 이런 접근법은 작업부하를 병렬로 실행될 수 있는 분리된 작업들로

분해함으로써 확장 가능한 쓰레딩을 어플리케이션에 추가하는 흔한 방식이다. 그러나 그림 11 에서 보는 것처럼 이런 접근법을 사용함으로 인해 여러 쟁점이 불거질 수 있다. 먼저 양 쓰레드 모두 해당 큐에서 정보를 잡아 끌어 동일한 데이터 구조에 접근해야 할 것이다.

기본적인 작업 큐

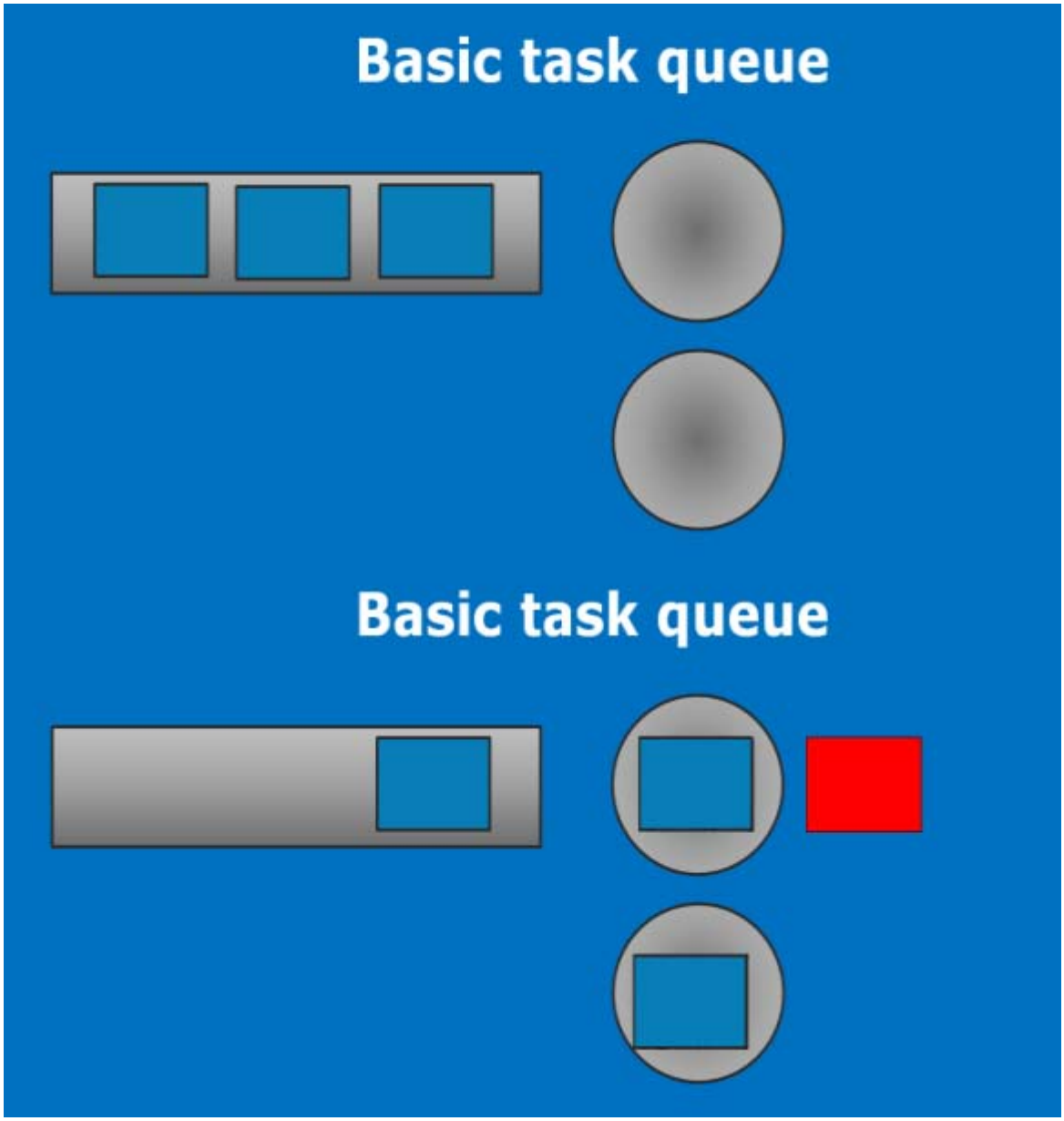


그림 11a. 간단한 작업 큐. 그림 11b. 쓰레드에 할당된 작업

프로그래밍이 데이터를 어떻게 동기화하느냐에 따라 이런 지연은 공유된 캐시 상에서의 최소 단위 연산을 갖는 몇 개의 사이클에서부터 mutex 또는 컨텍스트 스위치를 필요로 하는 다른 모든 연산을 사용하는 경우 몇 천 개의 사이클에 이르는 프론트 사이드 버스를 사용할 때의 몇

백 개의 사이클까지 어디에든 있을 수 있다. 기본 작업 큐에서 작은 작업들을 잡아 끄는 오버헤드는 꽤 커질 수 있고 그들의 병렬 처리를 통해 얻어지는 이점을 능가할 수도 있다.

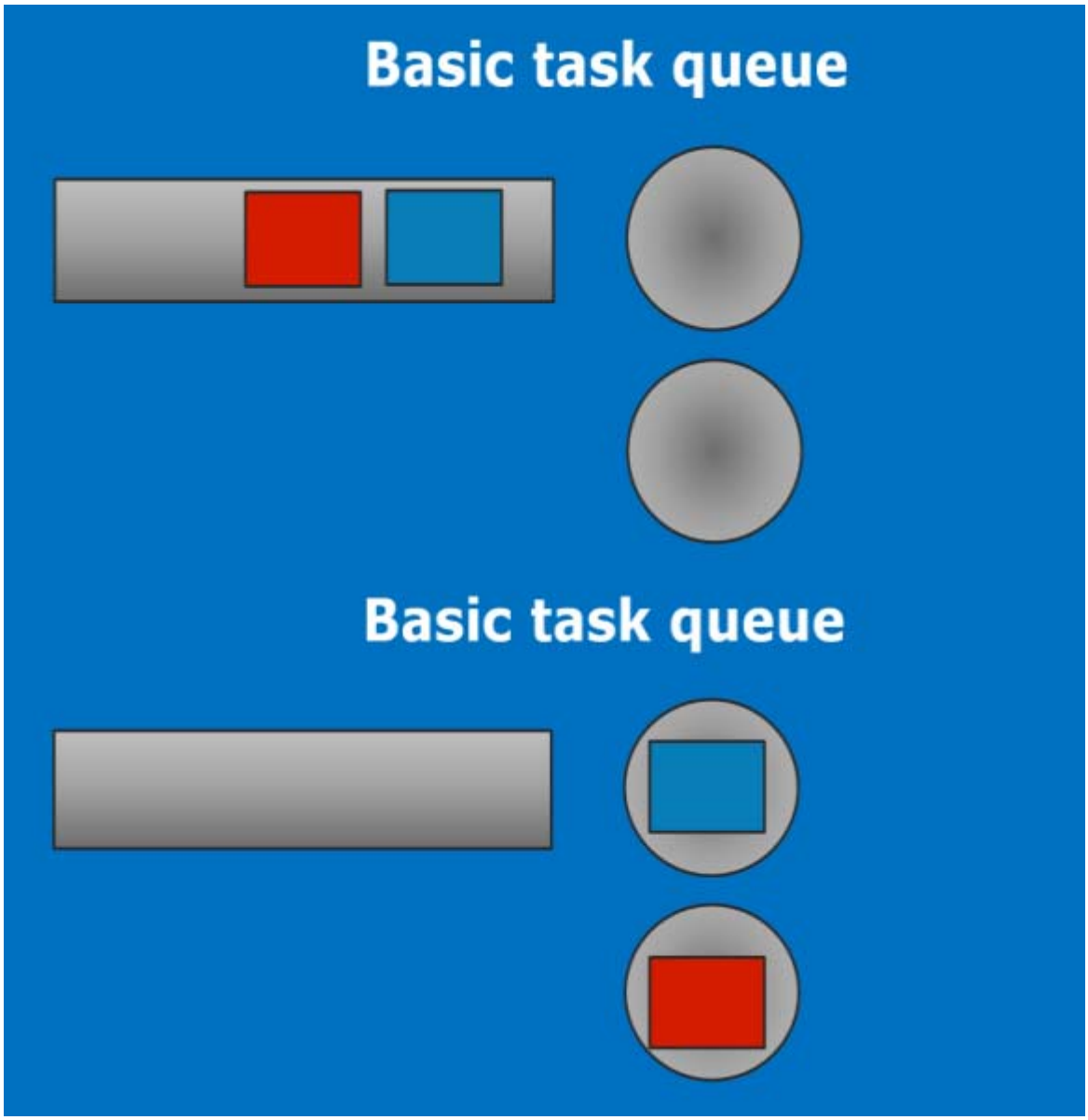


그림 11c. 큐에 추가되는 절차상의 작업. 그림 11d. 상이한 코어에서 실행되는 새로운 작업

이런 큐 설계의 또 다른 문제는 쓰레드와 그 위에서 처리되는 데이터 간의 상호 관계가 없다는 점이다. 동일한 데이터를 필요로 하는 작업은 서로 다른 쓰레드에서 실행될 수 있어 메모리 대역폭을 증가시키고 잠재적으로는 위에서 대강 설명된 경우와 유사한 공유 문제를 일으킬 수 있다. 부모 작업이 원래 실행된 쓰레드에 그것이 실행한 토대가 되는 쓰레드를 상관시키지 않고 개발자가 절차에 따라 생성된 작업(그림 11b 및 11c)을 다시 해당 큐로 놓을 때 또한 문제가 발생할 수 있다.

그 작업이 어쩌다 동일한 코어와 쓰레드를 히트하는 경우라면 개발자는 운이 좋은 것이고 그것이 필요로 하는 대부분의 데이터는 여전히 종전의 작업에서 온 캐시에 상존할 수 있다. 그렇지 않으면 그 작업은 위에 언급된(그림 11d) 지연과 다시 완벽하게 재동기화되어야 한다. 아울러 가용 쓰레드의 수가 증가하면서 분명 그러한 시스템의 복잡성도 극적으로 증가되며 "운 좋은" 캐시 히트의 가능성은 감소한다.

캐시 인식적인 작업 큐

호응을 얻고 있는 접근법 한 가지는 바로 쓰레드 당 개별 작업 큐를 구성하는 것으로(그림 12), 그럼으로써 쓰레드 간의 동기화 지점을 제한하고 인텔 쓰레딩 빌딩 블록에서 사용된 것과 유사한 테스크 스틸링 접근법을 구현한다.

절차에 따라 생성된 작업을 싱글 큐의 끝에 배치하는 대신 이 기법은 후입선출의 순서를 사용하는데 이 경우 새로운 작업은 부모 작업이 도출되었던 바로 그동일한 큐의 프론트로 밀려나간다. 현재 개발자는 이 캐시들을 활용할 수 있는데 왜냐하면 쓰레드가 새로운 작업을 마치는데 필요한 대부분의 데이터로 캐시를 준비시켰을 가능성이 높기 때문이다.

이 개별적 큐는 또한 공유된 데이터별로 작업들이 분류될 수 있음을 의미한다.

작업 큐는 캐시 인식적인 것임을 분명히 하도록 한다.

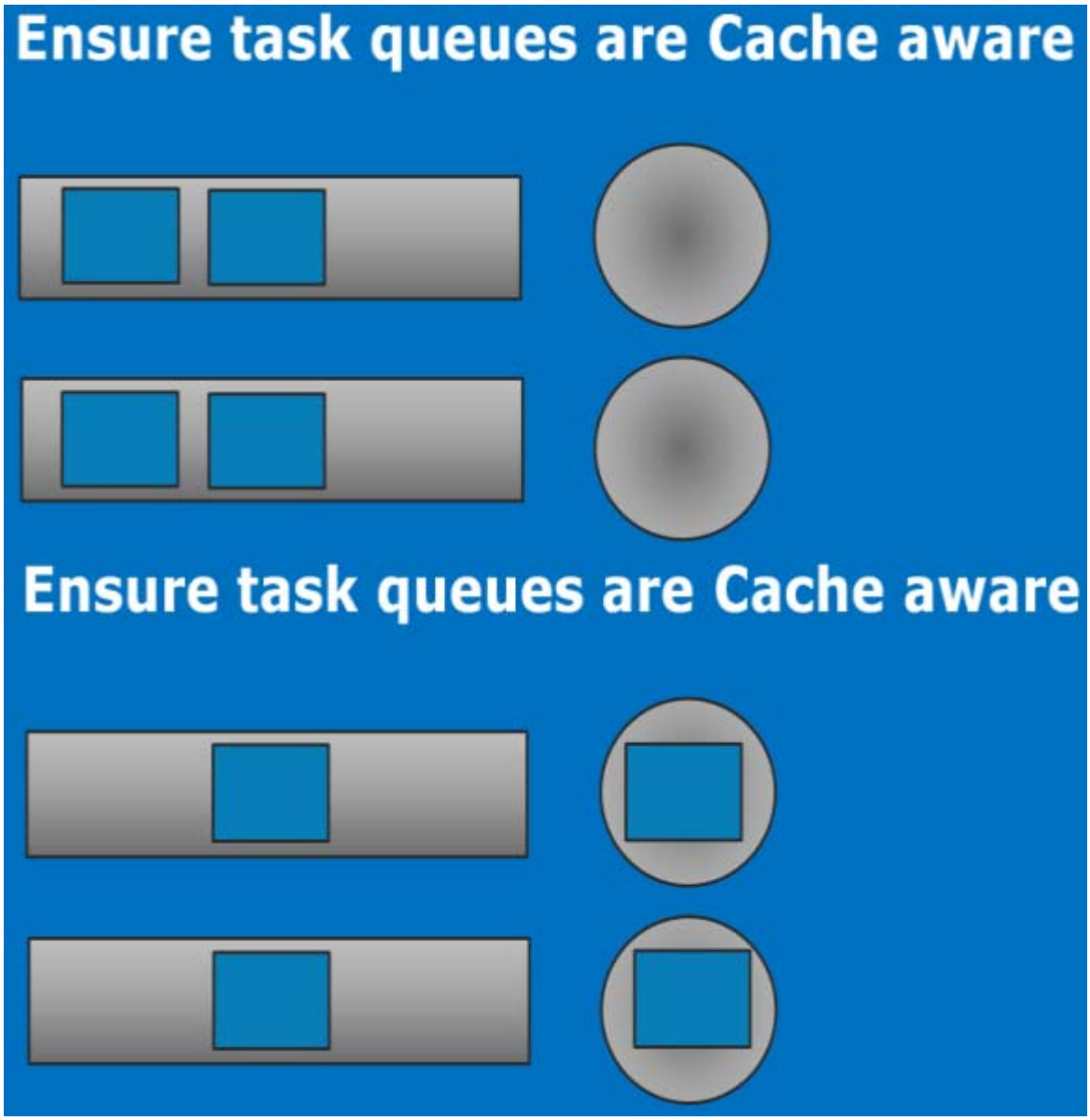


그림 12a. 쓰레드 당 그림 12a. 쓰레드는 큐에서 데이터를 독립적으로 끌어당긴다

작업 큐는 캐시 인식적인 것임을 분명히 하도록 한다.

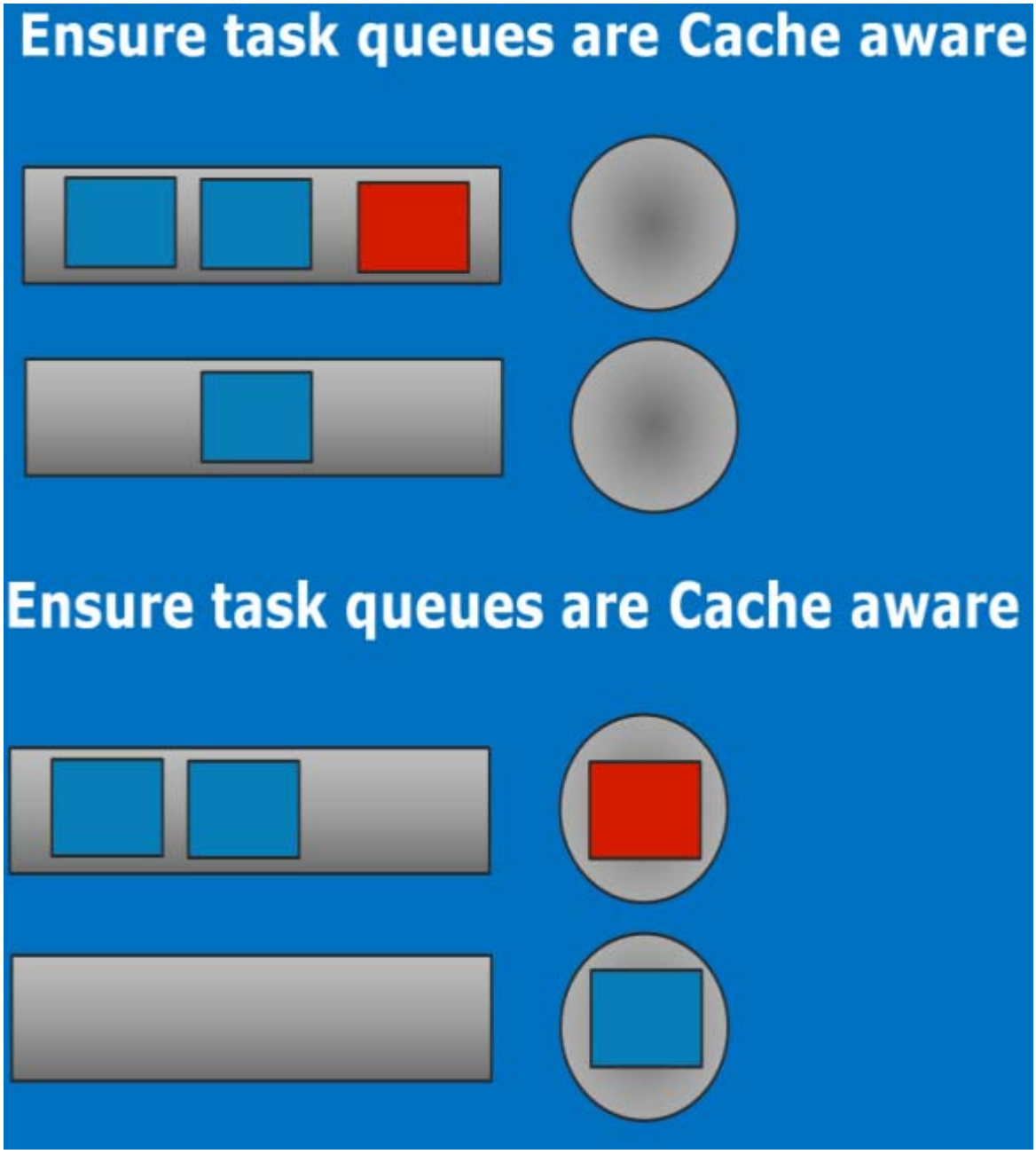


그림 12c. 절차에 따른 작업에 대한 LIFO 그림 12d. 작업으로 사용된 핫 캐시

쓰레드에 국한된 큐가 빈 경우에만 또 다른 큐에서 작업을 취하는 경우가 일어난다. 이것이 대체적으로 캐시를 효율적으로 사용하는 것은 아니지만, 가만히 앉아있는 것과 비교할 때 그것은 두 개의 나쁜 것 중 덜한 쪽일 수 있고 보다 균형 잡힌 접근법을 제공한다.

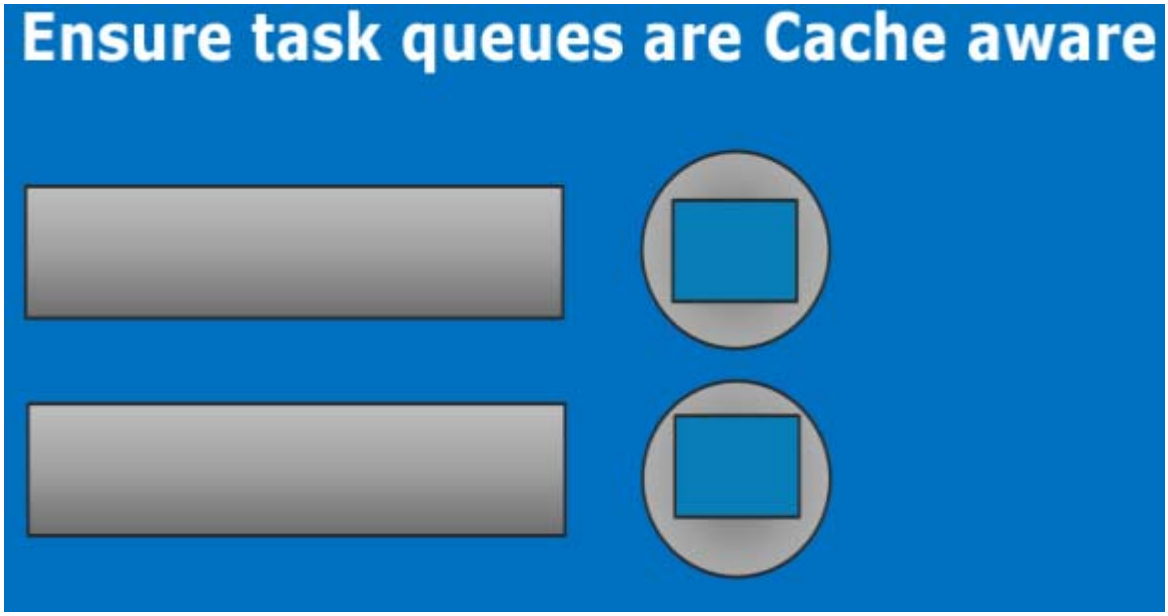


그림 12e. 큐가 빈 경우의 쓰레드 테스크 스틸

지능적인 작업 큐 외에 병렬 처리되고 있는 알고리즘 자체는 또한 기저의 하드웨어, 작업 할당의 지능적 사용을 인식하고 있어야 하며 작업이 실행될 때의 범주화는 캐시 활용을 크게 개선시킬 수 있어 그로 인해 전반적 성능도 개선되고 최고의 최적화 툴은 언제나 프로그래머의 뇌다.

인텔 하이퍼 쓰레딩 기술은 성능에 이득이 될 수 있다

인텔 펜티엄 4 프로세서와 함께 도입된 인텔 HT Technology 덕분에 적은 비용의 트랜지스터로 단일 코어 프로세서 상에서 또 한번의 컴퓨팅 쓰레드가 가능하다(그림 13). 이론적으로 성능을 증가시키는 것은 전력 효율적인 방식인데, 왜냐하면 두 개의 쓰레드라면 동일한 실행 코드 및 자원을 공유할 수 있을 것이기 때문이다. 많은 경우 코어에는 “버블”이 있고 이 경우 단일 쓰레드는 의존성 때문에 클럭 당 최대 4 개의 명령어를 실행할 수 없다. 이들 버블은 명령어로 채워질 수 있고 또 다른 쓰레드로부터 동작될 수 있어 그렇지 않은 경우보다 더 이르게 완료될 수 있다.

여기서의 함의는 바로 명령어 당 코어 클럭 신호가 나아진 반면, 개별 쓰레드의 성능은 프로세서가 각 쓰레드에 대한 데이터와 명령어를 이리저리 바꾸면서 다소 떨어질 수 있다는 점이다. 이는 보통 무시될 수 있는 부분이지만, SMT 가 그들 사이를 구분하지 않을 것이기 때문에 성능이 결정적인 쓰레드가 낮은 우선순위의 작업을 하는 쓰레드와 함께 실행될 때 그리고 마찬가지로 양 쓰레드 모두 취급할 때 종종 이것이 문제를 야기시킬 수 있다. 해당 OS 에 따라서 쓰레드가 실행되는 곳에 보다 새로운 OS 는 다중 쓰레드가 분배되는 방식을 개선시켜 하드웨어 성능을 최대화한다.

Intel® 하이퍼 쓰레딩은 성능에 이점을 제공할 수 있다				
SMT 없이	시간(proc. cycles)	주 : 각 박스는 프로세서 실행		

		유닛을 대표한다		
SMT				
아키텍처 상태	아키텍처 상태	동시적 멀티쓰레딩(SMT)으로도 알라ㅓ짐 - 코어 당 동시에 두 개의 쓰레드를 실행한다		
데이터 캐시		자원 공유 (캐시, 프론티드, 실행 유닛)		
CPU 프론트 엔드	비순차적 실행 엔진	코어 CPI (명령어 당 클럭 신호)를 개선한다		
		잠재적으로 쓰레드 CPI 를 저하시킨다		

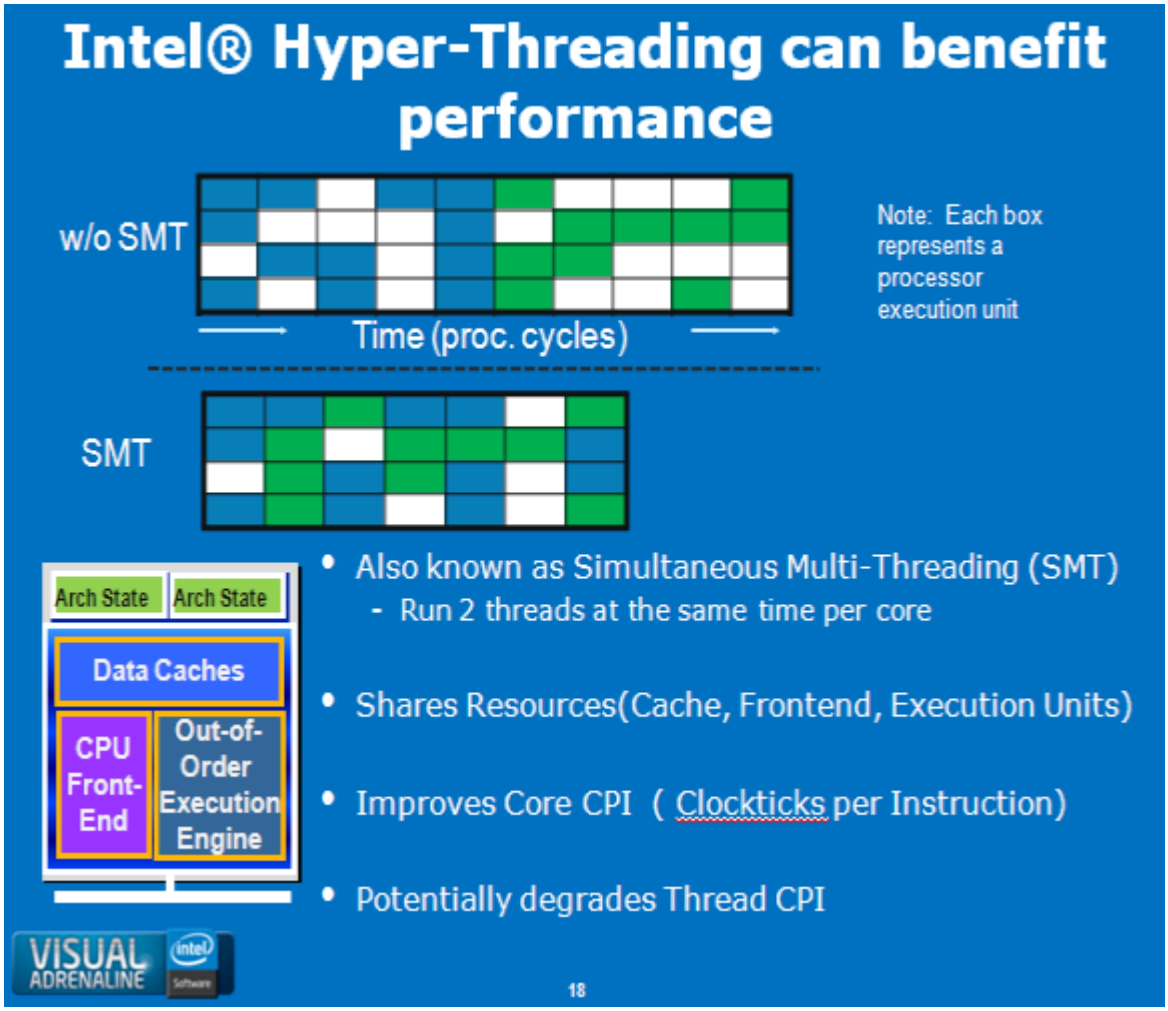


그림 13. 인텔 하이퍼 쓰레딩 기술은 명령어 당 코어 클럭 신호를 개선시키는 프로세스 쓰레드에서 “버블”을 제거한다.

SMT 지원은 초기의 아키텍처 상에서 그 효율성을 제한했던 많은 병목현상을 제거함으로써 인텔 코어 i7 프로세서 상에서의 구현이 개선되었다. 메모리 대역폭이 세 배로 증가되면서 이 프로세서는 대역폭에 제한될 가능성이 훨씬 떨어지며 데이터와 함께 다중 쓰레드 피드를 유지할

수 있다. 인텔 HT Technology 가 작동될 때 인텔 펜티엄 4 프로세서의 경우처럼 자원은 더 이상 부트에 분할되지 않는다. 인텔 코어 i7 프로세서 논리는 총체적으로 동적일 수 있도록 설계된다.

프로세서에서 사용자가 단지 하나의 쓰레드만을 실행하고 있음을 알아채는 경우, 해당 어플리케이션은 모든 전산 자원에 접근할 수 있으며 이들 자원은 소프트웨어와 OS 가 그 물리적 코어에서 한 개 이상의 쓰레드를 실행할 때만 공유된다. 쓰레드에 가용한 자원의 실제적 수 또한 종전 세대에 비교해 증가되었다(그림 14).

<table>
<tr><td colspan="4">SMT는 Intel® Core i7™에서 더 나아진다</td></tr>
<tr><td colspan="2" rowspan="2">Intel® Core i7™ u-아키텍처 이점

- 더 커진 캐시
- 대량 메모리 BW
- 4-급의 실행 엔진
- 더 많은 실행 자원</td><td colspan="2">성능 이득 SMT 가능 대 억제</td></tr>
<tr><td colspan="2">플로팅 포인트 om SPECfp_rate_base 2006* 추산에 기초한다

Interger 는 SPECint_rate_base 2006* 추산에 기초한다</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>Intel ® Core™ u-아키텍처</td><td>Intel ® Core™ i7 u-아키텍처</td><td></td></tr>
<tr><td>명령어 대기열</td><td></td><td></td><td></td></tr>
<tr><td>로드 버퍼</td><td></td><td></td><td></td></tr>
<tr><td>스토어 버퍼</td><td></td><td></td><td></td></tr>
<tr><td>동시 발생의 uOos</td><td></td><td></td><td></td></tr>
</table>

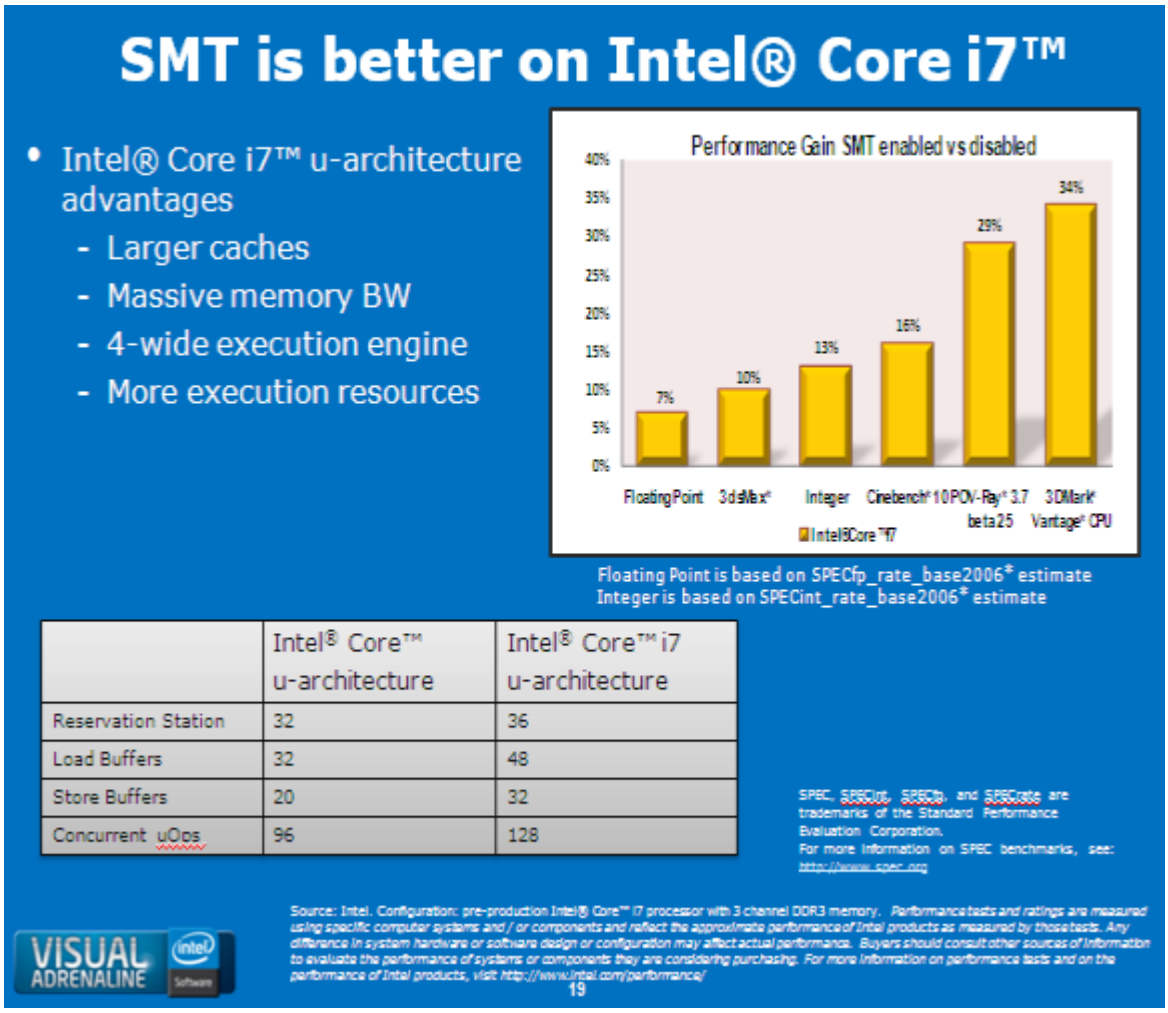


그림 14. 인텔 코어 i7 프로세서 마이크로아키텍처에서의 동시 멀티 쓰레딩 개선사항

방어적 목적의 SMT 용 설계
작업 / 코어
개별 작업의 지속기간은 비결정적이다
보다 세분화된 작업의 병렬 처리는 스케줄링 문제를 감소시킨다/없앤다
불필요하게 쓰레드를 스핀하지 않는다. 공유된 논리적 프로세서에 자원을 소비한다
● 필요 시, SMT 인식 스핀-대기 루프를 사용한다 (MWAIT 사용)
캐시 알고리즘이 공유된 자원을 차지하도록 한다
● VTune 에서의 L1 및 L2 캐시 적중률 여부를 점검한다

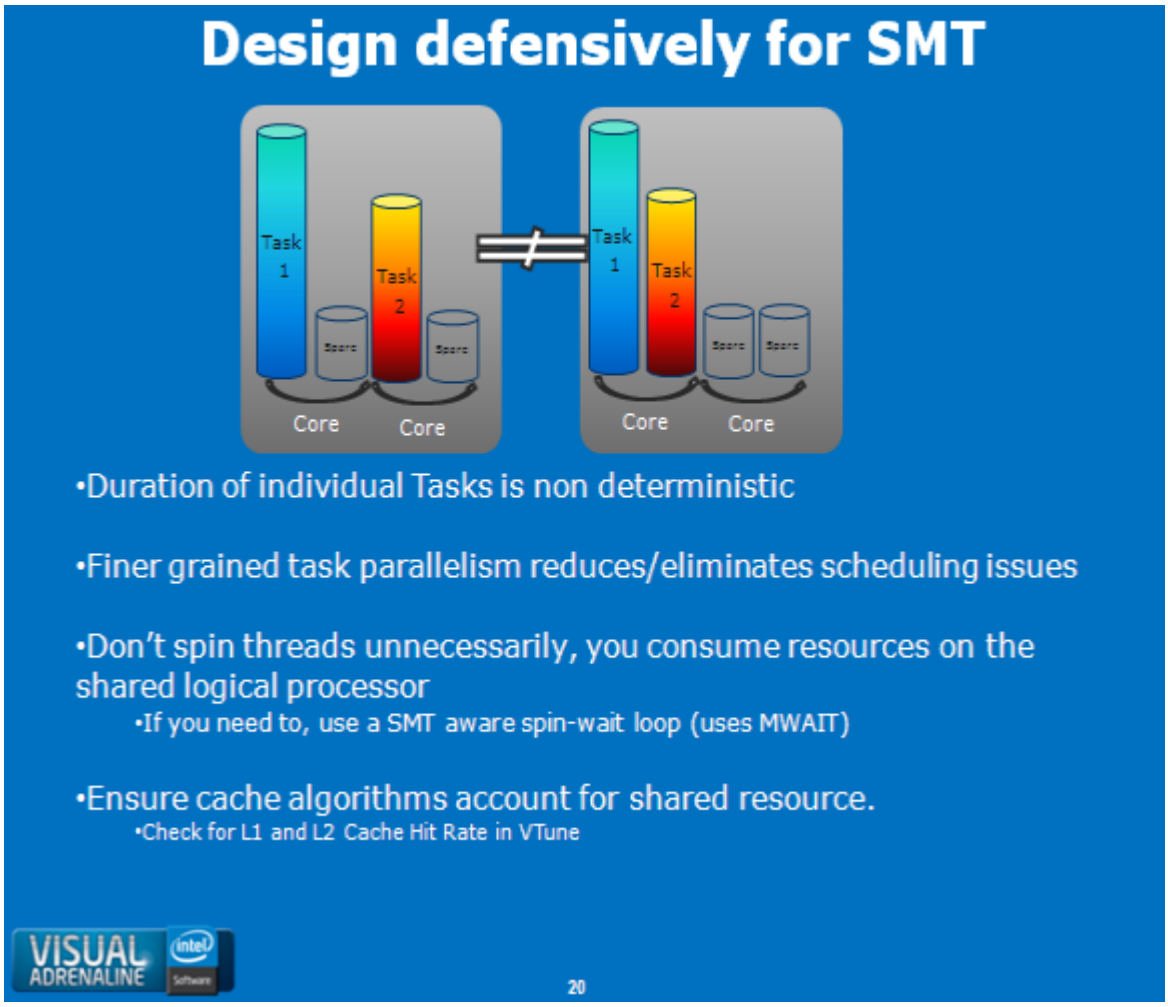


그림 15. SMT 용의 프로그래밍은 성능 패널티를 피할 수 있다.

또한 논리적 코어 및 물리적 코어 전반에 걸쳐 작업이 고르지 않게 분포되는 상황에서 SMT 가 성능에 도움을 주거나 적어도 잠재적 문제(그림 15)를 방지할 수 있도록 할 어플리케이션 코딩에 대한 간단한 접근법이 일부 존재하기도 한다. 첫 번째 해결법은 논리적 프로세서 사이에서 보다 자주 뒤섞일 수 있는 세분한 조각으로 작업들을 분해하는 것이다. 이는 해당 프로그램이 결국 동일한 물리적 코어에서 실행되는 가장 큰 두 개의 작업으로 끝나 버릴 가능성을 효과적으로 낮출 수 있다.

또 다른 문제는 개발자가 스케줄링을 목적으로 쓰레드를 스핀하려고 할 경우 불거진다. 논리적 프로세서에서 스핀하는 쓰레드는 실행자원을 이용하는데 이것은 코어의 나머지 쓰레드에 의해 더 잘 활용될 수 있다. 여기에 대한 해결책은 바로 EnterCriticalSectionSpinwait 같은 SMT 인식 기법을 사용하는 것으로써, 이것은 재점검 이전에 25 번의 사이클 동안 실제의 쓰레드를 슬립으로 놓을 수 있거나 실제로 쓰레드가 슬립을 이용하기보다는 실행 가능할 때 이벤트를 이용해 신호할 수 있다.

개발자가 배경 작업이 어떤 어플리케이션의 주 쓰레드에 양보할 수 있도록 "슬립 제로"를 사용하고 있고 그 양보로 인해 배경 작업의 실행이 정지될 수 없는 시스템 상에서 논리적 프로세서보다 쓰레드가 더 적은 경우 유사한 영향력이 발생할 수 있는데 왜냐하면 그 OS 에서는 그것이 완벽하게 빈 프로세서를 가졌다고 생각하고 있기 때문이다. 스핀은 기본적으로 무시되고

가끔씩만 실행되기로 되어 있던 싱글 쓰레드는 계속해서 쓰레싱되어 동일한 물리적 코어를 공유하는 모든 쓰레드에 있어 성능 감속을 야기한다. 이 경우 쓰레드만이 한 번에 그리고 적절한 때에 실행될 수 있도록 하기 위한 윈도우 이벤트 같은 보다 명시적인 동기화가 요구된다.

프로세서 토폴로지 함정

이러한 상이한 캐시 계층과 인텔 HT Technology 시나리오를 프로그램하고 계획하기 위한 직접적으로 보이는 방식 한가지는 바로 사용자의 시스템에서 특정 프로세서의 토폴로지를 찾아 정확한 하드웨어를 목표로 삼기 위해 알고리즘을 맞추는 것이다. 문제는 (CPU IDentification 에서 도출된) CPUID 규격에 대한 지식만으로 프로세서 토폴로지를 정확하게 정의하는 데에 있다.

CPUID 의 사용은 여분의 리프가 추가되었기 때문에 시간이 지나면서 변경되었다. 가용 프로세서의 수를 계산하기 위해 과거에 사용되었던 정확하지 않지만 일반적인 방식은 CPUID 리프 4 를 사용하는 것이었다. 이것은 뜻하지 않게 종전의 아키텍처에서 동작했는데 왜냐하면 그것이 물리적 패키지에서 주소지정 가능한 최대의 ID 수를 제공했고, 초기의 하드웨어에서 이것은 실제의 프로세서 수와 언제나 동일했기 때문이었다. 그러나 이런 가정이 인텔 코어 i7 프로세서의 경우에는 해당되지 않는다.

CPUID (쉬운 방법을 취하지 않는다!)
패키지 당 코어의 수를 탐지할 수 있는 제대로 된 방법
CPU 토폴로지 열거 백서를 정확하게 복사한다
http://softwarecommunity.intel.com/articles/eng/3887.htm
- CPUID 는 논리적 프로세서를 탐지할 수 있다 - CPUID 는 공유된 캐시를 탐지할 수 있다 - CPUID 는 캐시 크기를 보고한다
잘못 되기 쉽다 – 정확하게 복사한다

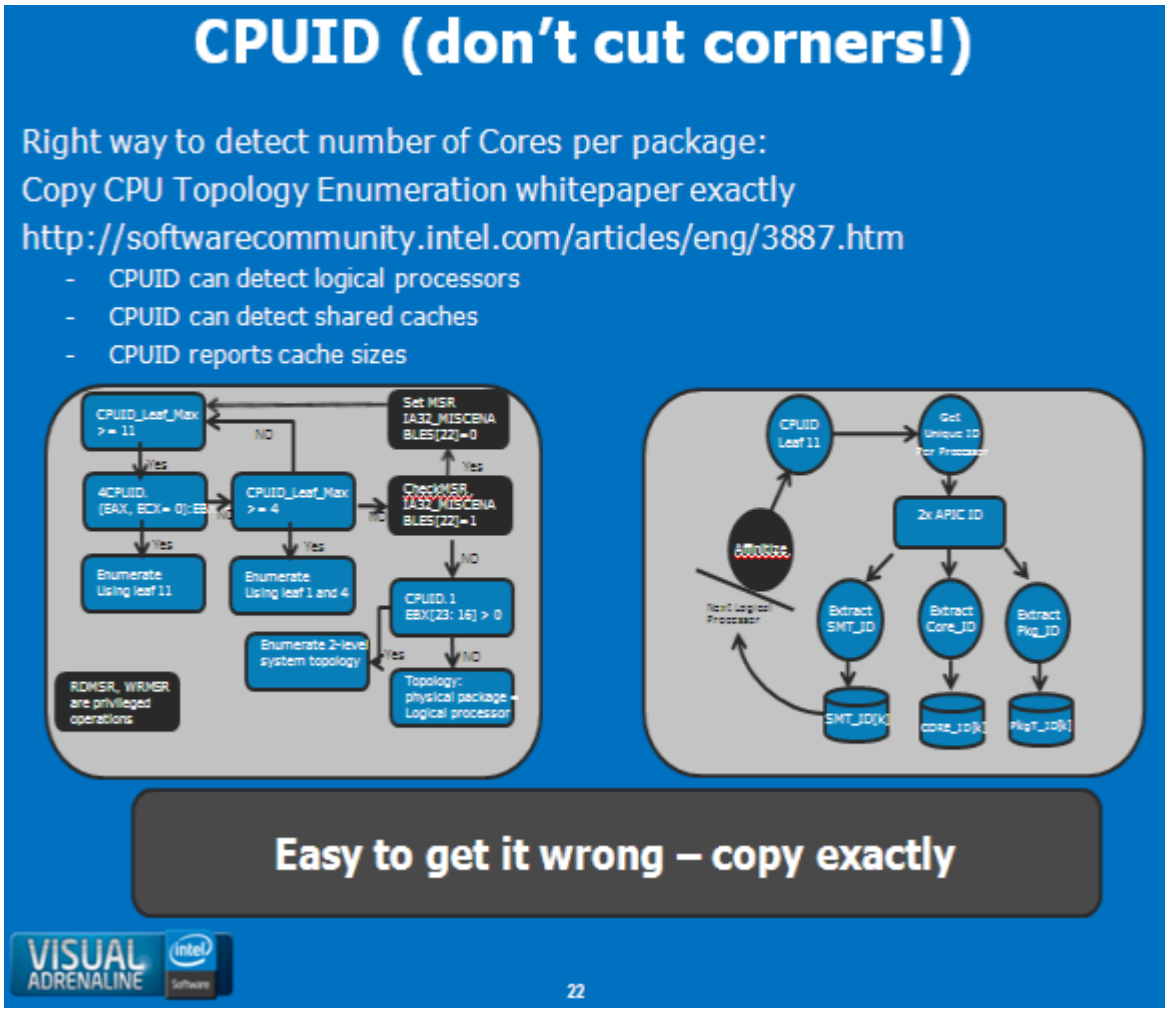


그림 16. 정확한 프로세서 토폴로지를 위한 테스트 과정은 매우 복잡하다.

CPUID 가 각자 유리하게 동작하게 만들 수 있는 방법은 있다. 그러나 그 방법은 매우 복잡하고 여러 단계를 거치는 기나긴 과정이다(그림 16). CPUID 검출 알고리즘에 대한 구체적인 예시가 담긴 여러 건의 양질의 백서를 인텔의 웹사이트에서 찾아볼 수 있다. 일단 제대로 된 자리에 있다면 그 CPUID 정보로 사용자의 프로세서에 얼마나 많은 코어가 있는지 뿐 아니라 어떤 코어가 어떤 캐시를 공유하는지도 알 수 있다. 이 과정 동안 쉬운 길을 택한다면 해당 애플리케이션의 성능 및 신뢰성이 쉽게 강등될 가능성은 있지만 이렇게 되면 특정 알고리즘에 맞는 알고리즘을 맞춤으로 작성할 수 있는 가능성이 열린다.

쓰레드 친화도

게임 엔진에서 쓰레드 친화도 사용에 관해서는 보다 복잡한 상황이 발생한다. 토폴로지 열거로 생성된 마스크는 OS, OS 버전(x64 대 x86)에 따라, 심지어 서비스 팩 간에도 다를 수 있다. 이것이 쓰레드 AFFINITY 의 설정을 프로그램에서 하기에 매우 위험한 것으로 만들 수 있다. 코어가 인텔 펜티엄 4 프로세서 상의 CPUID 열거에서 등장하는 순서는 인텔 코어 i7 프로세서에서의 등장 방법과 완전히 다르며, 인텔 코어 i7 프로세서는 OS 사이에서 역순으로 나타날 수 있다. 코드의 친화도는 이들 차이점이 세심한 열거와 알고리즘 설계를 통해 설명되지 않는 경우 그들을 방지하기 보단 문제를 일으킬 가능성이 있다.

또한 개발자라면 게임과 그 OS 의 전반적 소프트웨어 환경을 고려해야 한다. 어떤 어플리케이션 내의 미들웨어는 그 자체의 쓰레드를 생성할 수 있고 그들이 사용자의 코드와 잘 스케쥴링될 수 있도록 하는 API 를 제공하지 않을 수 있다. 그래픽 드라이버 또한 해당 OS 에 의해 프로세서 코어로 할당되는 그 자체의 산출하고 따라서 게임 쓰레드에 맞게 친화력이 있는 것을 적절히 코딩하는데 관계되었던 작업이 오염될 수 있다. 어플리케이션에서 친화도를 정하는 것은 향후 개발을 위해 기본적으로 잠재적인 골치거리는 접어두고 단기간의 이득에 거는 것이다.

부정확한 CPUID 정보와의 기계 내장형 쓰레드 친화도가 사용된다면 어플리케이션 성능은 실제로 두 번 적중될 것이다. 최악의 경우 어플리케이션은 종전의 아키텍처를 토대로 한 가정 때문에 보다 새로운 하드웨어에서 실행되지 않을 수 있다. 게임의 경우 쓰레드 친화도 사용을 피할 것이 권장되는데 왜냐하면 그것이 단기간의 솔루션을 제공하고 이는 일이 잘못 되는 경우 모든 이득을 능가해 장기간 문제를 일으킬 수 있다 대신 하드 바인딩보다는 SetIdealProcessor 같은 프로세서 힌트를 사용한다.

개발자가 이런 방향으로 나아가고자 결정한다면 향후의 프로세서 아키텍처와 하드웨어가 성능 또는 안정성에 크게 영향을 미치는 경우 사용자가 친화도를 억제할 수 있도록 하는 것과 관련해 안전 기능이 마련되어야 한다. 소프트웨어에서 친화도의 실행을 필요로 하는 경우, 결합된 하드웨어와 소프트웨어 환경에서 난처한 버그 발생이 기다리고 있을지도 모른다.

OS s 스케줄링

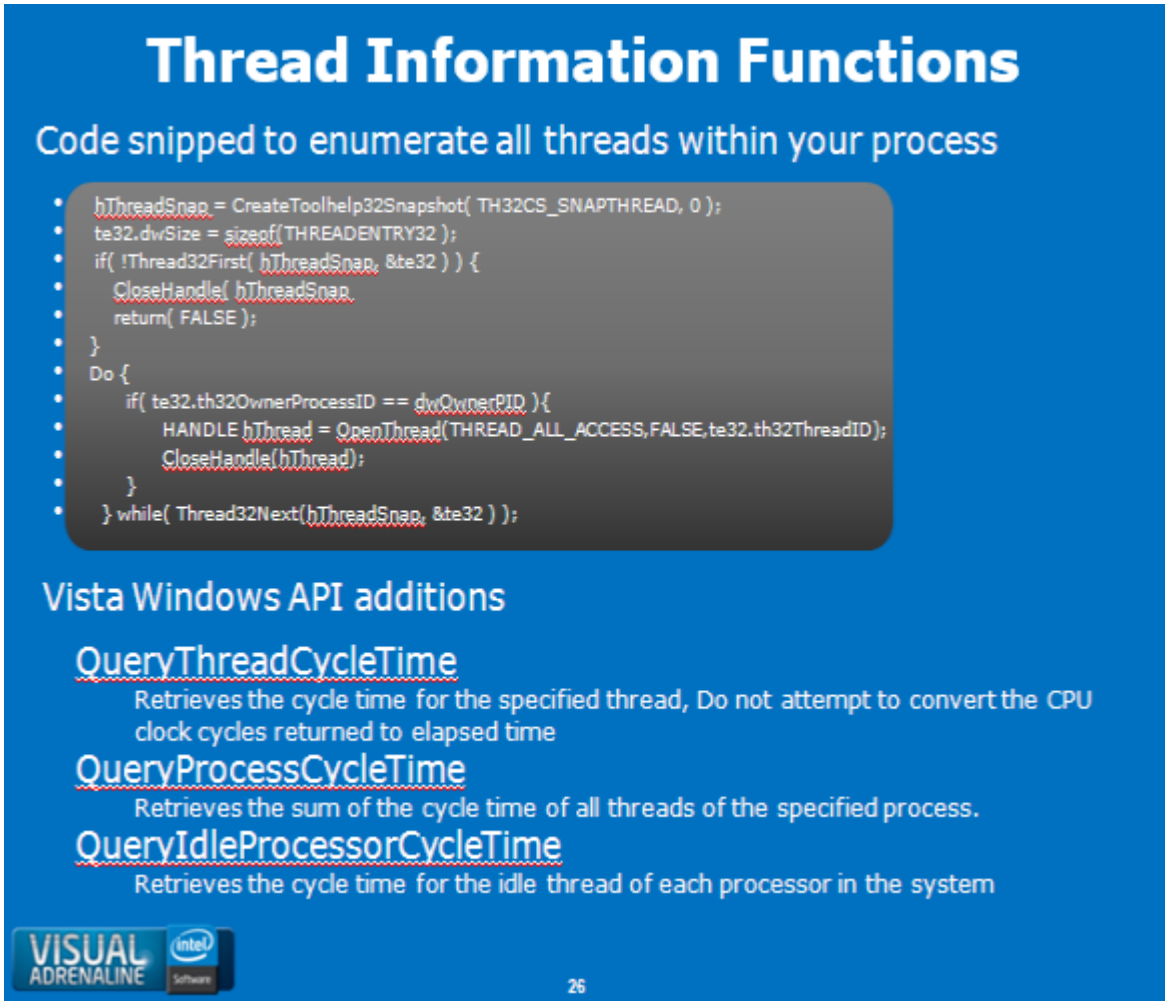
마이크로소프트 윈도우 OS 버전마다 쓰레드 스케줄링을 약간 다르게 처리한다. 심지어 윈도우 7 는 윈도우 비스타의 처리 방법과 비교되는 개선사항을 갖추고 있다. 스케줄러는 우선순위 기반의 라운드 로빈 시스템으로써 해당 시스템에서 현재 실행되는 모든 작업에 공정하도록 설계된다. 쓰레드가 밀려나기 전 실행될 수 있도록 윈도우가 부여하는 시간 할당은 실제로 꽤 길어 이벤트 자발적 전환을 유발하지 않는 한, 약 20 밀리초-30 밀리초 정도다. 게임의 경우, 이것은 매우 큰 시간 길이로써 한 프레임 이상 지속될 수 있다.

OS 스케줄링의 코스 게인을 고려할 때 작업이 반드시 적당한 시간 동안 실행되도록 하기 위해 게임이 그 자체의 쓰레드를 정확하게 스케줄링하는 것이 중요하다 부정확한 쓰레드 동기화는 사소한 배경이나 낮은 우선순위의 쓰레드가 예상보다 훨씬 오래 실행되어 게임의 크리티컬 쓰레드에 크게 영향을 미침을 의미할 수 있다. 안됐지만 과도한 동기화 또한 좋지 않은 것일 수 있는데, 왜냐하면 쓰레드 간의 변경은 수 천 개의 사이클을 요할 것이기 때문이다.

데이터를 공유하는 쓰레드 간의 너무 많은 컨텍스트 전환은 성능을 떨어뜨릴 수 있다. 컨텍스트 전환으로 인한 성능 패널티를 피하려면 곧 다시 사용될 가능성이 있는 자원 상에서 짧은 스핀록을 실행하는 것이 도움이 된다. 물론 성능 저하는 대기시간이 너무 긴 경우 발생할 것이고 그래서 쓰레드 록을 면밀히 주시하는 것이 중요하다.

쓰레드 정보 기능
프로세스 내에서 쓰레드를 열거하기 위해 잘린 코드
비스타 윈도우 API 추가분

QueryThreadCycleTime
명시된 쓰레드에 필요한 사이클 시간을 검색한다. 경과된 시간으로 복귀된 CPU 클럭 사이클의 변환을 시도하지 않는다.
QueryProcessCycleTime
특정 프로세서가 갖는 모든 쓰레드의 사이클 시간 총합을 검색한다
QueryIdleProcessCycleTime
시스템에서 각 프로세서의 유휴 쓰레드에 필요한 사이클 시간을 검색한다



그램 17. 쓰레드의 자체 모니터링을 통해 성능 병목현상을 쉽게 찾을 수 있다.

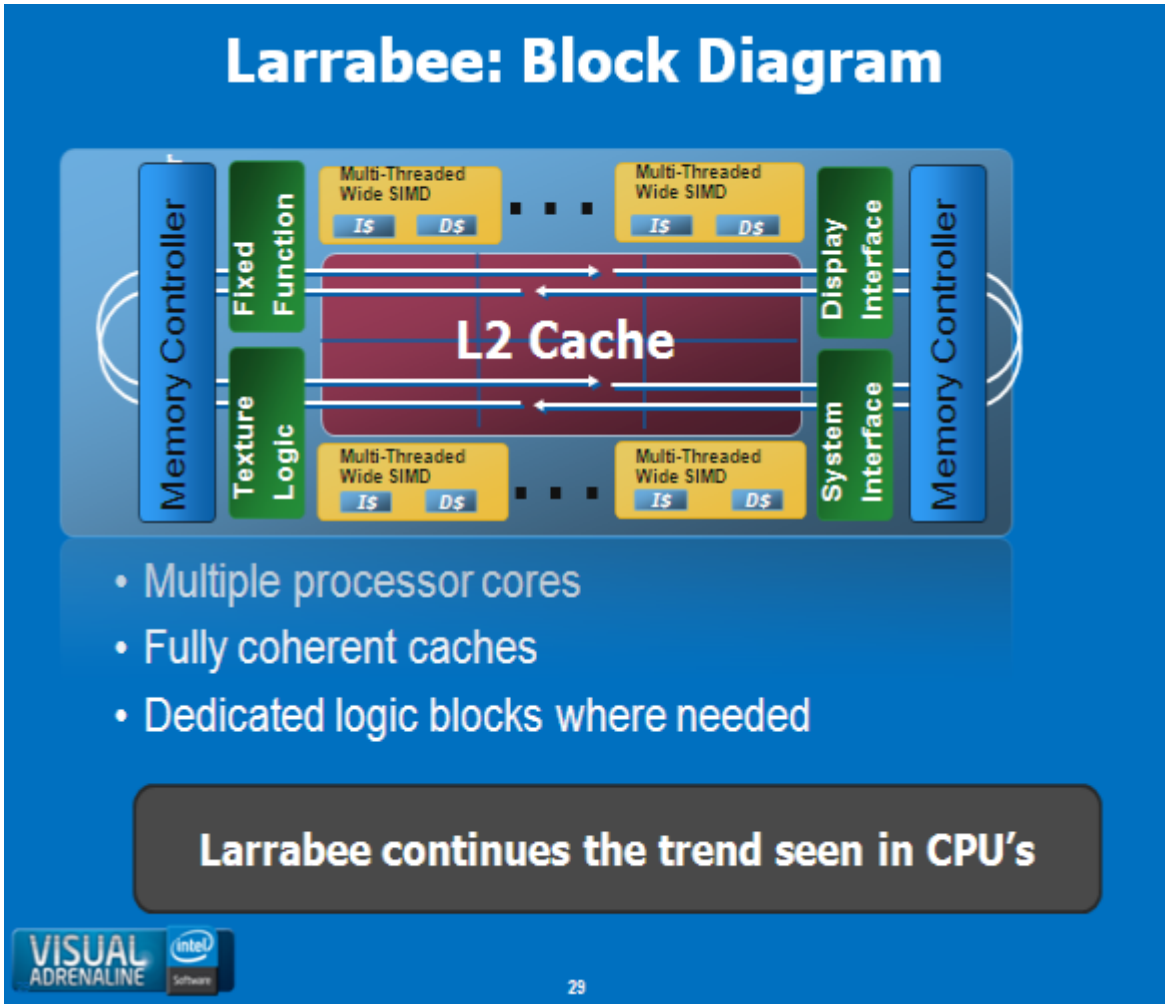
그림 17 은 어플리케이션과 미들웨어가 생성했던 쓰레드를 열거하는 방법을 나타낸다. 비스타는 일단 쓰레드가 열거되면 특정 코어에 얼마나 많은 시간이 소요되었고, 특정 쓰레드가 얼마나 오래 실행되었는지 등을 개발자에게 알려줄 수 있는 몇 개의 새로운 API 추가분을 도입했다. 이런 점을 고려할 때 개발 또는 최종 사용자에게 있어 쓰레딩 성능에 대한 진단적 분석을 온스크린 방식으로 구성하는 것이 쉬울 것이다.

앞으로 남은 문제

멀티 쓰레딩, 마이크로아키텍처 그리고 게이밍의 미래에 대해 생각하자니 2 개의 흥미로운 기술이 떠오른다. 샌다 브리지 아키텍처라 불리는 다음의 "톡"은 인텔 어드밴스드 벡터 익스텐션(Intel AVX)을 선보일 것이다. 오늘날 인텔 SSE4 에서의 128 비트 명령어를 토대로 확장된 인텔 AVX 는 현재의 128 비트 SSE 명령어와 바로 대체 가능한 256-비트 명령어를 제공할 것이다. 64 비트 레지스터를 지원하는 현재의 프로세서가 해당 레지스터 공간의 절반을 사용함으로써 32 비트 코드를 실행할 수 있는 것과 거의 마찬가지로 AVX-가능 프로세서는 절반의 레지스터를 사용해 인텔 SSE 명령어를 지원한다.

벡터는 4-와이드보다는 오늘날 아키텍처가 처리할 수 있는 8-와이드를 실행할 것이고 그럼으로써 낮은 수준의 데이터 레벨 병렬성의 가능성을 증가시키고 게임 성능에 맞게 향상될 것이다. 샌디 브리지는 또한 더 높은 코어 수와 더 나은 메모리 성능을 제공해 세분화된 작업 수준에서 쓰레딩을 더 많이 할 수 있도록 지원할 것이다.

Larrabee: 블록 다이어그램
다중 프로세서 코어
완벽한 일관성의 캐시
필요 시의 전용 논리 블록
Larrabee 는 CPU 에서 보이는 동향을 지속시킨다



17. Larrabee 는 그래픽 커뮤니티에 프로세서 타입 아키텍처를 가져올 것이다

멀티코어 라라비 아키텍처(그림 17)는 앞으로 보게 될 흥미로운 기술이다. GPU 로 규정되기는 했지만, 그 설계는 기본적으로 현재의 멀티코어 프로세서를 한 차원 끌어올리고 있다. 쓰레딩 성능, 프로그래밍 모델 및 친화성에 관련해 언급된 모든 것은, 물론 이것이 훨씬 더 높은 코어 수로 확장되긴 하겠지만 이 아키텍처에 적용될 가능성이 있다. 미래를 계획하는 개발자라면 인텔 코어 i7 프로세서 그리고 심지어 샌디 브리지를 넘어 어떻게 확장되는지를 고려해야 하고 캐시 및 쓰레드 사이의 통신의 측면에서 라라비를 고찰해야 한다.

맺음말

데스크톱 프로세서는 빠른 속도로 진화 중이며, 비록 성능상의 개선점이 개발자들에게 사실성과 충실함을 개선시킬 상당한 기회를 안겨주긴 하지만, 그들은 서로 다른 프로그래밍 모델을 필요로 하는 새로운 문제를 야기하기도 했다. 개발자들은 더 이상 오늘날 그들이 작성한 코드가 자동으로 내년의 하드웨어에서 더 잘 실행될 것이라는 가정을 내릴 수 없는 실정이다. 대신 다가올 하드웨어 아키텍처를 대비해 계획하고 특히 가용 쓰레드의 수에 관련해 변화하는 프로세서 환경에 따라 쉽게 조정되는 코드를 작성해야 한다.

진화하는 하드웨어의 한가지 부작용이라면 성능 시험의 중요성이 증가하고 있다는 사실이다. 정확한 툴을 사용해 작업부하가 명확한 체계적인 성능 시험 세트를 개발하는 것이 어플리케이션이 광범위한 하드웨어에서 최고의 성능을 내는데 있어 매우 중요하다.

다양한 하드웨어에서 게임의 설계 사이클을 조기에 시험하는 것이 아마도 설계 결정에 따라 최장기간의 성능 솔루션이 제공되도록 하는 최상의 방법일 것이다. 본 기사에서 대략적으로 설명된 대다수의 쟁점은 캐시 행위 및 특정 쓰레드에서의 명령어 쓰로풋을 모니터할 수 있는 인텔 VTune 퍼포먼스 애널라이저 같은 툴을 사용해 확인이 가능하다. 또한 쓰레드 동시성을 측정하도록 고안된 인텔 쓰레드 프로파일러 같은 툴도 있다.

본 기사에서 언급된 프로세서에 대한 최적화 가이드라인 및 인텔의 기타 프로세서에 관해서는 http://www.intel.com/products/processor/manuals/을 참조한다.