SDK 코드의 최적화 접근법

빠르고 단순한 고도의 최적화

우리에게 최근 차세대 SDK 를 현 세대 및 소형 플랫폼에 이식시키는 작업은 다소 버거웠다. 그러면서 나타난 다양한 난제는 성능 최적화가 가장 보상이 높은 것 중의 하나가 되어가고 있기 때문이다. 수 주에 걸쳐 우리는 이 기사에서 자주 간과했던 기법을 사용해 성능의 대략 4 배가 증가했고 라이브러리 크기는 적당히 감소했다. 어셈블리 언어는 거의 필요치 않았지만 전반적으로 대부분의 최적화는 높은 수준이었다.

첫 단계- 프로파일 프로파일 프로파일

최적화 실행을 수행하기 전에 최적화 변경 내용을 벤치마크 할 수 있는 탄탄한 토대를 설립해야 한다. 이를 위해 우리 게임에서 단순하게 반복할 수 있는 시나리오를 설정했다. 이 시나리오는 전형적이지만 SDK 치고는 큰 노력을 요하는 상황을 나타내었다. 우리의 경우 매 실행마다 프로파일링 데이터는 약 30 초인 것으로 포착되었다. 이 때의 결과를 저장해 다음의 프로파일링 실행과 비교했다.

우리는 기본적으로 서로 다른 두 종류의 실행을 하는데 있어 프로파일러에서 많은 시간을 소요했다. 첫 번째 종류는 프로파일러가 고주파수로 어플리케이션을 방해하는 샘플 기반의 프로파일링이었다. 방해할 때마다 이 프로그램 카운터는 쿼리되고 튜닝 소프트웨어는 어떤 기능이 실행되는지를 기록한다. 이 프로파일링 기법은 정확도 측면에서는 형편없지만(결과를 방해하거나 로그하는 오버헤드 때문) 대부분의 주기가 소요되는 곳을 신속하게 발견할 때는 유용하다. 일단 잠재적 방해자를 정했으니 이제 소스코드를 검토해 보다 집중된, 함수 전용의 프로파일을 또 한번 실행할 수 있다. 이 때 시간이 소요되었던 곳을 정확하게 짚어내는 일을 빨리 할 수 있다.

두 번째 단계-코드 조사

때로 운이 좋으면 고차원 함수에서 일어나는 병목현상을 재빨리 발견할 수 있다. 또 어떤 때는 확실치 않은 때가 있는데 코드가 이미 차세대 플랫폼용으로 최적화되었던 우리의 경우가 정말 그랬다. 이를 염두에 두고 우리는 생성되었던 실제의 어셈블리 언어와 교차 참조하는 쪽을 택했다. 대부분의 디버거에는 이를 더욱 쉽게 만드는 '소스코드 주석'이 있다. 이것을 함으로써 우리는 코드가 불필요하게 확대되게 한 것으로 여겨진 영역을 빨리 발견할 수 있었다. 이렇게 하기 위해 어셈블러 프로그래밍을 많이 알 필요가 없는 사람도 있다는

점을 말해두어야겠다. 물론 도움이야 되겠지만 개발자로써 우리는 뭔가가 얼마나 많은 코드를 필요로 할 지에 대한 감이 있다.

다음 페이지에서 우리가 했던 최적화보다 더욱 유익한 것들을 일부 소개해보겠다.

명령 캐시 사용법의 향상

먼저 캐시의 동작 방법에 익숙치 않다면 이 기사의 끝에 있는 리소스를 권하고 싶다.

명령 캐시는 당신의 친구나 마찬가지니 부디 죽이지는 말아달라. 성능 최적화를 위한 곳으로 간과되곤 하지만 캐시가 더 작은 콘솔에서 이 영역을 최적화하는데 소요된 시간은 큰 수확일 수 있다.

우리의 경우 가장 큰 성공은 과도한 인라인 사용을 제거함으로써 가능했다. 우리가 최적화했던 라이브러리는 거대한 캐시 시스템을 염두에 두고 생성되었다. 프로그래머는 코드를 빠르게 실행할 수 있는 가장 쉬운 방법은 그것을 인라인 하는 것이라는 일반적 가정을 증명할 수 있다. 이는 함수호출 오버헤드가 함수 코드의 실행 값과 비교해 큰 영역에서 사실일 수 있다. 그러나 우리는 지극히 큰 함수가 인라인되는 것을 보았다. 우리의 목적 플랫폼은 캐시가 작고 캐시라인 크기가 32 바이트다. 각 명령 코드는 4 바이트를 필요로 하기 때문에 8 개의 명령마다 캐시 되지 않은 코드를 실행할 때 캐시 실패를 예상할 수 있다는 의미가 된다. 극단적인 경우 과도한 인라이닝은 동일하게 인라인 된 서브 함수를 반복적으로 호출하는 코드는 결코 캐시를 hit 하지 않았음을 의미했다. 서브 함수가 인라인 되지 않았다면 초기의 실패는 제쳐두고 서브 함수 코드가 캐시에 있을 것이다. 이것과 캐시라인을 채우면 빠르지 않다는 사실을 합하면 엄청난 성과를 볼 수 있다.

우리가 인라인 했던 것에 대해 선택적이었음을 뒤로 한 채 우리는 이 영역에서 성능을 향상시키기 위해 또 한 쌍을 최적화했다. 알다시피 캐시는 한 번에 다수의 인라인을 담을 수 있다. 게임의 영역이나 모듈을 집단화함으로써 그것들이 동시에 캐시에 들어맞는다. 우리가 다뤘던 코드는 확실하게 정의되었던 SDK 부분이 연속적으로 실행되었다는 점에서 이미 모듈이었다. 이들 영역에 있는 명령 캐시 성능을 보다 쉽게 하기 위해 우리는 두 가지 다른 접근법을 사용했다. 먼저 각 모듈 내에서 서로 가까이서 실행되었던 함수들을 확인했다. 이들 함수는 이 때 소스코드 내에서 함께 집단을 이루었다. 이것의 목적은 이미 캐시에 있던 코드의 가능성을 늘리고 라인 경쟁 문제의 가능성을 줄이는 것이었다. 둘째 우리의 컴파일러에는 링크 프로세스의 일부분인 함수 스트라이핑 옵션이 있지만 우리의 경우 이것은 죽은 코드를 제거할 때 비효과적인 것으로 증명되었다. 이를 염두에 두고 우리는 실행이 되지 않는 걸로 여겨진 코드를 수동으로 제거하고 필요치 않았던 함수성 또한 제거하는 지루한 작업에 착수했다. 시간 소모적이었지만 일단 우리의 코드 풋프린트가

제거된 이상 성능 향상에서 상당한 성과가 있었다. 결과적으로 명령 캐시를 미조정하는데 드는 시간이 꽤나 소모되었다. 성능 영향은 상당했고 노력도 거의 들지 않습니다.

데이터 캐시 사용법의 개선

우리가 두 번째로 주목하는 부분은 우리의 데이터 캐시 사용법을 향상시키는 것이었다. 다시 말하지만 우리의 경우 캐시 라인 크기는 32 바이트였다. 따라서 우리가 비슷한 때에 동일한 캐시 라인으로 접근되었던 데이터를 그룹화하는 것은 합리적이었다. 생각해보면 알겠지만 이것이 소스 코드를 통해 나아가는 것과 관련이 있기 때문에 빠른 과정은 아니다. 그러나 성능 이득이 가치가 있다고 여겼기 때문에 괜찮은 시간 소모였다.

이 영역에서 우리가 했던 또 다른 최적화는 데이터 풋프린터를 줄이는 것이었다. *bool* (컴파일러가 많은 32 비트)을 사용하다가 대신에 비트 플래그를 사용했다. 우리는 정수값을 살펴보고 가능한 곳에 이들을 char 로 저장했다. 다시 말하지만 이 과정은 느리지만 꽤나 가치가 있다.

결국 코드에 한 쌍의 핵심적 데이터 정렬이 있었는데 다음과 같이 표시되었다.

```
for ( i=0; i < num; ++i)
        if ( data[i].m_active )
                data[i].doSomething();
```

대부분의 요소가 정지되었지만 이 무고해 보이는 코드는 캐시 실패와 루프 반복을 할 때마다 상당한 사이클 패널티를 생성하고 있었다. 그 이유는 해당 데이터가 캐시 라인에 맞지 않았기 때문에 캐시라인 충전은 반복할 때마다 필요했다. 이 겨우 데이터 구조가 하나의 캐시 라인에 맞을 수 있는 방법이 없었다. 확실한 해결책은 데이터 구조에서 m_active 플래그를 제거하고 이를 개별 구조에서 비트 플래그로써 저장하는 것이었다. 개별 구조가 캐시 경계에서 정렬되게 하면서 동일한 캐시라인 내에서 256 배열 요소에 맞게 플래그를 저장할 수 있었다.

많은 프로파일러가 명령/데이터 개시 실패를 모니터한다는 점을 언급해야 할 것 같다. 두 말하면 잔소리지만 이렇게 하면 핫스팟을 확인할 때 시간을 많이 절약할 수 있다.

메모리 관리의 최적화

우리가 찾아낸 최대의 히트는 SDK 에서 시스템 malloc 과 free 연산으로의 호출이었다. 알다시피 new 와 delete 또한 동일한 메모리 할당 과정을 겪는다. 호출의 전단 볼륨은 대대적인 성능유출을 야기하고 있었기 때문에 우리는 이 영역에 집중했다. 우리가 했던 첫

번째 최적화는 우리가 할당하는 일반적 크기에 맞게 설계된 사용자 정의 메모리 매니저와 목적 플랫폼을 사용하는 것이었다. 우리는 로깅 정보를 메모리 매니저에 추가했고 그럼으로써 대부분의 활동이 발생하고 있던 곳을 확증할 수 있게 하였다. 일단 이것을 알았을 때 처음부터 그것을 호출하지 않는 메모리 매니저를 호출할 수 있는 최고의 방법에 착수했다. 클래스 할당 당 크기가 더 적고 고정된 경우 우리는 클래스 구조 그 자체 내에서 단순히 이 메모리를 저장했다. 왜냐하면 병목현상은 할당의 크기가 아니고 호출 주파수였기 때문이다.

우리가 했던 또 다른 최적화는 함수가 시작할 때 할당되어 끝에서 제거되는 가변크기 메모리용이었다. 이를 위해 우리는 alloca 함수를 사용했다. 이것은 CPU 오버헤드가 너무 낮은 스택 상에 메모리를 할당했다. alloca 를 사용할 때 클린업 함수가 없었는데 이 함수가 존재할 때 이 메모리/스택이 자동으로 클린업 되었기 때문이다. alloca 를 사용하는 또 다른 이점은 malloc 과 달리 메모리 단편화의 가능성이 없다는 점이다.

new 및 *delete* 연산자의 최적화는 우리만의 연산자를 생성하는 간단한 사례였는데 이 때의 연산자는 우리의 자체적인 메모리 매니저를 통해 메모리를 할당했다. 각 요소는 우리가 획득했던 메모리 어드레스에서 적절하게 'new'ed 되었다.

명령 세트를 사용하라

우리가 최적화했던 SDK 는 플랫폼 전용 명령을 사용하지 않았다. 예를 들어 *cosf, sinf, atanf* 와 같은 함수에 일치하는 호출은 모두 기본 구현을 사용하고 있었다. 이들 기본 함수들은 너무 느려서 우리는 이 함수들을 플랫폼 전용 변종으로 교체했다. 우리 대부분은 쉽게 획득할 수 있는 근사함수의 어셈블러 버전을 사용했다. 그러나 하나의 플랫폼은 실제로 코프로세서 명령 세트의 부분인 *sin* 및 *cos* 를 갖고 있다.
다른 플랫폼 전용 최적화는 모든 매트릭스/벡터 연산에 SIMD 명령의 사용을 포함했다. 이러한 최적화는 특히 어셈블리 언어를 거의 필요치 않고 어떤 경우엔 컴파일러가 이 작업을 더 쉽게 만드는 intrinsic 을 지원한다. 거대한 코드 기반을 변경할 때 우리가 당면했던 유일한 어려움이라면 SIMD 명령문이 16 바이트를 필요로 했다는 점과 이 어쩔 수 없는 사실로 인해 거대한 소스기반을 변경해야 했다는 점이다. 다시 말하면 SIMD 를 사용한 것과 성능 면에서의 차이가 놀랄만하기 때문에 이러한 최적화는 가치가 있다.

결국 목적 플랫폼이 8 또는 16 바이트 로드/스토어 연산을 지원하면 이 때 당신이 할 일은 이들을 사용하면 되는 것이다. 기본 *memcpy* 및 *memset* 함수를 무척 빠르게 대체하는 것들을 코드 하는데 오래 걸리지 않는다.

누구나 실수한다

디셈블러를 통해 컴파일된 코드를 살펴보면 C++코드에서 단순한 실수들이 쉽게 발견되기도 한다. 예를 들어 코드 확대 같은 실수들이 쉽게 발견되곤 하는데 이것은 프로그래머가 객체를 반환하거나 임시 객체를 사용해서 야기된 것이다. 예를 들어 우리가 최적화했던 매트릭스/벡터 클래스의 경우 원래의 프로그래머는 두 버전의 일반 매트릭스/벡터 연산을 제공했다. 다음과 같이 하나는 클래스 내에서 표준 함수다.

```
Matrix3x3 Multiply( const Matrix3x3 &matA, const Matrix3x3 &matB )
```

그리고 다른 하나는 오버로드 된 연산자다.

```
Matrix3x3 operator*( const Matrix3x3 &mat )
```

대부분의 코드 작성은 예를 들어 result = matrixa * matrixb 는 오버로드 된 연산자를 사용했다. result.Multiply(matrixa, matrixb)보다는 확실히 단순해 보이지만 이러한 단순성에는 대가가 따른다. 이미 추측했겠지만 더 단순해진 오버로드된 연산자는 스택에서 반환된 후 지정 객체로 복사되는 임시 객체를 수반한다. 대부분의 C++ 프로그래머는 이를 알고 있지만 잊는 사람이 많은 것 같다. 두말할 필요 없이 우리의 해결책은 오버로드된 연산자를 제거하고 프로그래가 보다 빠르고 확실한 버전을 사용하도록 하는 것이다.

디버거에서 쉽게 발견할 수 있는 또 다른 간단한 실수는 배정도 수학의 사용이다. 이것은 프로그래머가 항상 가변적인 포인트값에 f 의 추가를 잊어버리거나 *sqrtf* 대신 *sqrt* 를 사용하는 것 만큼이나 단순하다. 이들 함수 호출의 발견은 중단점 설정만큼이나 쉽다.

결국 명령 세트의 효과적 사용을 언급하는 것은 의미가 있다. 값에 의해 객체를 반환해야 하고 당신의 명령 세트가 8/16 바이트 피연산자를 지원한다면 클래스 할당 연산자를 오버로드하고 복사 명령을 빠르게 하기 위해 이 피연산자들을 사용한다.

컴파일러에서 최상을 얻는다

가장 단순하게 최적화할 수 있는 것 중 일부는 컴파일러 옵션에 있는 스위치의 조작을 이용한 것이다. 당신의 컴파일러에 따라 기본으로 활성화 될 수 있는 몇 개의 일반 옵션을 포함할 예정이다. 이들 옵션은 코드 성능을 저해할 뿐 아니라 당신의 코드 크기를 늘릴 수 있다.

RTTI -런타임 타입 정보

이 옵션으로 당신은 클래스/객체에 대한 정보를 실행시간으로 질의할 수 있고 그런 만큼 추가적 메모리 오버헤드가 따라온다. *dynamic_cast* 및 *typeid* 연산자는 이러한 정보에 의존하나 당신의 프로그램이나 라이브러리에서 이것을 사용하지 않으면 분명 RTTI 를 무효화할 만하다. 이 옵션을 무효화 해 둔 채 상당히 감소된 코드 크기를 보아야 한다. 당신이 이것을 사용하는지의 여부가 확실하지 않다면 옵션을 무효화 하고 당신의 코드를 재구축한다. 이 때의 컴파일러는 RTTI 를 필요로 하는 코드를 발견하면 오류를 생성할 것이다.

예외 처리

이것은 또 다른 옵션으로써 기본적으로 활성화되지만 성능 영향 및 코드 크기 증가 모두를 동반한다. 예외 처리를 사용하지 않으면 이 옵션을 무효화하는 게 안전하다. 예외 처리를 피한다는 말은 당신의 코드에 있는 *catch, try* 또는 *throw* 를 사용하지 않는다는 것이다.

new 를 점검하라

종종 이렇게 지원되는 옵션에는 적은 메모리와 실행처리 부담이 따라온다. 이 옵션은 new 연산자로 반환되는 포인터가 무효가 아니라는 점을 확인하기 위해 추가 코드를 생성한다. 당신이 자신의 메모리 관리를 확실히 제어하고 이것을 우연히 발견하리라고 예상하지 않는다면 이 옵션을 무효화하는 것이 좋을 것 같다.

최적화 설정

상이한 최적화 설정의 영향을 살펴보는데 꽤 많은 시간을 소요하라고 권하고 싶다. 최적화의 수준을 최고로 설정하고 싶겠지만 경험으로 볼 때 그렇게 한다고 반드시 코드가 가장 빨라지는 것은 아니다. 최고의 최적화 설정으로 인해 컴파일러가 무엇을 인라인 할지에 대해 완벽하게 자유롭다는 것을 깨달았다. 이러한 설정을 통해 알게 된 것은 성능의 증가가 눈에 띄지 않는 경우 코드 크기가 상당히 증가했다는 점이다. 개인적으로 자동 인라이닝을 포함하지 않는 최고의 최적화 설정을 선호한다. 나라면 어떤 함수가 인라인 되는지를 완벽하게 제어하겠다.

당신의 프로그램 크기를 줄이고자 할 때 내가 흥미롭게 관찰했던 것은 '크기에 맞게 최적화하라' 옵션이 있는 컴파일러에 대한 부분이었다. 필자가 이 옵션을 시도했던 플랫폼에서 예상된 바대로 생성된 코드는 최적화 설정이 더 높은 상태에서 생성된 코드보다 더 느리게 실행되었다. 그러나 프로그램 크기의 차이는 미미했다. (우리의 경우 0.3%)

결론적으로 요약하자면 크기에 대한 여러 가지 컴파일러 옵션으로 해 본 일부 결과는 다음과 같다.

최적화 없음, RTTI on, 예외 on, new 점검 on - **11,674,296 bytes**
최적화 없음, RTTI on, 예외 on - **11,587,128 bytes**
최적화 없음, RTTI on - **7,579,672 bytes**
최적화 없음 - **7,139,128 bytes**
완전 최적화 - **4,623,800 bytes**
자동 인라이닝 없는 완전 최적화 - **4,574,392 bytes**
크기에 맞는 최적화 - **4,559,928 bytes**

결론

우리는 실제로 코드 기반의 최적화 수준을 낮게 했지만 대부분의 성능은 본인이 여기서 다뤘던 단순한 고차원적 기법에서 나온 것이었다. 코드의 최적화는 매우 보상적인 경험이며 여러분도 나만큼 즐기길 바란다.

참고

http://en.wikipedia.org/wiki/CPU_cache
http://en.wikipedia.org/wiki/SIMD
인텔 아키텍처를 위한 고성능 레시피 ISBN: 0971288712

연락처

NathanWhitaker@blueyonder.co.uk