특집: 게임 프로그래밍의 성능 문제

(Sponsored Feature: Common Performance Issues in Game Programming)

베키 하이네만(Becky Heineman)
가마수트라 등록일(2008. 6. 18)

http://www.gamasutra.com/view/feature/3687/sponsored_feature_common_.php

본문 텍스트

[마이크로소프트의 XNA 관련 가마수트라 마이크로사이트에 실린 이번 기사에서 XNA 디벨로퍼 커넥션의 기자이자 인터플레이의 공동설립자인 베키 하이네만(Becky Heineman)은 게임 제작 시 '로드-힛-스토어(Load-Hit-Store)'라는 성능 저하 요인을 피할 수 있는 팁을 제공한다.]

"90%의 시간이 10%의 코드에 할애된다. 그러므로 그 10%를 최대한 빠른 코드로 만들어라."

컴퓨터 게임 제작 시 겪게 되는 가장 흔한 문제 중 하나가 바로 성능의 문제이다. 디스크 액세스, GPU 성능, CPU 성능, 레이스 컨디션, 메모리 대역폭(내지 이의 부족) 등의 문제는 정지 내지 지연을 야기해 초당 30 프레임 짜리 게임을 초당 9 프레임 게임으로 만들어버린다.

본 기사에서는 가장 흔한 CPU 성능 저하 요인 중 하나인, 로드-힛-스토어를 설명하는 한편 이를 피할 수 있는 팁과 기법을 소개한다.

로드-힛-스토어

Xbox 360 의 성능 엔지니어에게 로드-힛-스토어에 관해 물어보면 대개 진저리를 치며 그에 관해 열변을 쏟아낼 것이다. 일반적으로 순차적인 메모리 읽기 연산(The load), 레지스터에 값 할당(The Hit), 레지스터에 값을 실제로 쓰기(The Store)는 파이프라인의 여러 단계에 숨겨져 있으므로 이들 연산으로 인해 정지(stall)가 유발되지는 않는다. 그러나 읽어 들이는 메모리 위치가 이전 쓰기 연산에 의해 작성된 것일 경우 "저장(Store)' 작업을 완료하기까지 무려 40 사이클이나 필요하다.

예:

stfs fr3,0(r3) ; 부동 소수형 저장
lwz r9,0(r3) ; 정수 레지스터로 돌아가 데이터 읽기
oris r9,r9,0x8000 ; 부정으로 강제

첫 번째 명령은 32 비트 부동 소수 값을 메모리에 기록하고, 다음 명령은 이를 다시 읽는 것이다. 흥미로운 사실은 정지가 발생하는 곳에는 로드 명령, 즉 'oris' 명령이 없다는 것이다. r9 으로 '저장'이 끝나야 명령이 완료되며, 그 전까지는 L1 캐시가 업데이트되기를 기다려야 한다.

그렇다면 어떤 일이 발생할까? 첫 번째 명령은 데이터를 저장하고 L1 캐시를 'dirty'로 표시한다. 데이터를 L1 캐시에 쓰고 CPU 에서 이용할 수 있으려면 약 40 사이클이 소요된다. 그 동안 한 명령이 해당 캐시로부터 데이터를 요청하며 '저장'을 위해 R9 을 '할당'한다. 저장이 끝날 때까지는 마지막 명령이 실행될 수 없으므로 정지가 발생한다.

Microsoft 의 PIX 툴로 이 문제를 파악할 수 있다. 'oris' 명령을 정지의 원인으로 규정하는 작업은 복잡하기 때문에 PIX 로 일련의 이벤트를 시작하는 로드 명령을 플래그하여 프로그래머가 문제를 해결할 수 있다.

스레드 하나에 CPU 가 3 개

파워 PC(PowerPC)를 3 개의 별도 CPU 로 생각해보자. 각 CPU 는 각각의 명령 세트, 레지스터 세트, 데이터 연산을 수행하는 방식을 보유하고 있다. 그 중 첫 번째는 32 개의 정수 레지스터를 가진 정수 유닛인데 여기서 연산의 많은 부분이 이루어진다.

두 번째는 32 개의 부동 소수 레지스터로 구성된 부동 소수 유닛으로서 여기서 단순 계산을 모두 처리한다. 마지막은 복잡한 벡터 연산을 처리하는 128 개의 레지스터로 구성된 VMX 유닛이다.

이들 유닛을 공통 명령 스트림을 공유하는 3 개의 CPU 라고 생각해보자. 이들 유닛에서는 데이터를 상호간 직접 이전할 수 없다. 정수 레지스터의 데이터를 실수 레지스터로 이동시키는 명령이 없기 때문에 CPU 에서는 정수 값을 메모리에 쓴 후 메모리 읽기 명령을 이용하여 이를 실수 레지스터로 읽어와야 한다. 이러한 작업 패턴을 '로드-힛-스토어'라고 한다.

정수 유닛에서 데이터를 실수 유닛으로 이동시키는 것은 그리 어려운 작업이 아니다…..

예:

```
int iTime;
float fTime;
fTime = static_cast<float>(iTime);
```

이는 단순하면서도 매우 흔한 코드이다. 그러나 파워 PC 에서는 메모리로부터 실수 레지스터로 로드하는데 부동 소수 명령이 실행될 수 있도록 정수 값을 메모리에 저장하는 명령이 생성된다. 'fix-up' 명령이 이어져 정수 표시를 부동 소수 표시로 변환하면 시퀀스가 완료된다.

'로드-힛-스토어'는 타이트 루프에서 멤버 값이나 레퍼런스 포인터를 반복자로 이용할 때 흔히 발생한다.

예:

```
for (int i=0;i<100;++i)
{
m_iData++;
}
```

컴퍼일러는 위 루프를 m_iData+=100 으로 변화시켜 단일 연산으로 최적화할 수 있는 지 파악할 수 있는 정도로 똑똑하지는 않다. 대개 런타임 시 m_iData 를 로드하여 인크레멘트(increment)한 후 이를 해당 포인터가 지시하는 메모리에 되돌려 저장한다. 루프의 첫 번째 패스는 최대 속도로 실행되지만, 반복 실행되면 m_iData 값은 해당 루프의 이전 패스의 쓰기 연산으로부터 '로드-힛-스토어'에 빠진다.

레지스터는 페널티를 행사하지 않으므로 코드가 아래처럼 보이도록 다시 쓰여질 경우,

```
int iData = m_iData;
for (int i=0;i<100;++i) {
iData++;
}
m_iData = iData;
```

연산이 레지스터 내에서 이루어지기 때문에 코드가 훨씬 더 빨리 실행될 뿐 아니라 컴파일러가 이를 iData+=100 로 변형하여 '로드-힛-스토어'에 빠지지 않도록 한다.

참고

'로드-힛-스토어'는 코드 내에서 발생할 수 있는데, 심지어는 전혀 그래서는 안 되는 상황에서도 발생할 수 있다.

```
void foo(int &count)
{
count = 0;
for (int i=0;i<100;++i) {
if (Test(i)) {
++count;
}
}
}
```

이 코드는 로드-힛-스토어를 발생시킨다. 어떻게 발생시키는 지 확인해보자.

변수 'count'는 메모리 바운드이다. 모든 것이 여기에 쓰여지고 대부분의 경우 읽히고 메모리를 통과한다. 어떤 변수가 메모리 바운드이고 타이트 루프 안에 있다면 이는 로드-힛-스토어를 유발할 수 있다. 이를 수정하는 법은 이전의 코드 예와 유사하다.

```
void foo(int &output)
{
int count = 0;
for (int i=0;i<100;++i) {
if (Test(i)) {
++count;
}
}
output = count; // Write the result
}
```

벡터로드-힛-스토어

이전에 제시한 사례들을 통해 실수 및 정수 트랜잭션에서 로드-힛-스토어가 얼마나 생성되기 쉬운지를 알 수 있었다. VMX 레지스터 세트 역시 동일한 문제에 당면한다. 일반적으로 일부 수학 연산은 VMX 연산에서 보다 능률적으로 처리될 수 있다고 알려져 있지만, 여기에 비-벡터(non-vector) 데이터가 포함되면 어떨까?

Xbox 360 에서 VMX 레지스터 intrinsic __vector4 는 하나의 구조 위에서 매핑된다. 아래 이유로 인하여 이러한 구조 요소들에 대한 런-타임 액세스가 중단된다.

```
XMVECTOR Radius = CalcBounds();
pOut->fRadius = Radius.x;
```

VMX 레지스터가 하나의 구조로서 이용되기 때문에 2 번째 열이 로드-힛-스토어를 발생시킨다. 그 결과 컴파일러는 전체 레지스터의 콘텐츠를 로컬 메모리에 기록해야만 한다. 이후 첫 번째 요소가 실수 레지스터를 통해 읽어지고 그런 다음에야 pOut->fRadius 로 값이 쓰여진다.

숨겨진 로드-힛-스토어를 유발하지 않으면서 이와 동일한 코드를 작성하는 방법은 다음과 같다.

```
XMVECTOR Radius = CalcBounds();
__stvewx(&pout->fRadius,__vspltw(Radius,0),0);
```

VMX에는 어떤 엔트리든 단일 부동 소수로 쓰는 기능이 있다. vspltw() 연산은 요청된 엔트리를 임시 벡터 레지스터로 복사하고 stvewx() 연산은 부동 소수 쓰기를 처리한다. 값에 액세스하는 컴파일러 기능은 권하고 싶지 않다.

정수-부동 소수 변환 제거

변환된 데이터가 프레임 타임 할당량(frame time quantum)이나 화면 너비처럼 반-정적(semi-static)이라면 이 데이터는 복제될 수 있고 정수형이나 부동 소수형을 읽는 함수는 이를 페널티 없이 읽어올 수 있다.

예:

```
typedef struct ScreenSize_t {
int m_iWidth;
int m_iHeight;
float m_fWidth;
float m_fHeight;

inline Void SetHeight(int iWidth) {
m_iWidth = iWidth;
m_fWidth = static_cast(iWidth);
```

```
};
} ScreenSize_t;
```

결과적으로 정수 값을 필요로 하는 함수는 멤버의 'I'폼으로부터 데이터를 로드하며 부동 소수 입력값을 필요로 하는 함수는 'f' 폼으로부터 데이터를 읽어 들이게 된다. 이 값들은 정수형 및 부동 소수형을 동시에 업데이트하는 인라인 함수에 의해 업데이트될 수 있다.

반복자를 이용하고 변환할 경우, 또 다른 형식의 정수에서 부동소수로 변환된다. 자체적인 문제들로 인하여 부동 소수 비교는 최소화하도록 한다. 이 예에서 정수에서 부동 소수로의 변환에 의해 각각의 루프와 함께 'angle'이 생성되면서 로드-힛-스토어가 발생한다.

```
VOIDDebugDraw::DrawRing( const XMFLOAT3 &Origin,
const XMFLOAT3 &MajorAxis, const XMFLOAT3 &MinorAxis, D3DCOLOR Color )
{
static const DWORD dwRingSegments = 32;
MeshVertexP verts[ dwRingSegments + 1 ];

XMVECTOR vOrigin = XMLoadFloat3( &Origin );
XMVECTOR vMajor = XMLoadFloat3( &MajorAxis );
XMVECTOR vMinor = XMLoadFloat3( &MinorAxis );

FLOAT fAngleDelta = XM_2PI / (float)dwRingSegments;
for( DWORD i = 0; i<dwRingSegments; i++)
{
FLOAT fAngle = (FLOAT)i * fAngleDelta;
XMVECTOR Pos;
Pos = XMVectorAdd( vOrigin, XMVectorScale( vMajor, cosf( fAngle ) ) );
Pos = XMVectorAdd( Pos, XMVectorScale( vMinor, sinf( fAngle ) ) );
XMStoreFloat3( (XMFLOAT3*)&verts[i], Pos );
}

verts[ dwRingSegments ] = verts[0];

SimpleShaders::SetDeclPos();
SimpleShaders::BeginShader_Transformed_ConstantColor
( g_matViewProjection, Color );
g_pd3dDevice->DrawPrimitiveUP( D3DPT_LINESTRIP, dwRingSegments, (const
```

```
VOID*)verts, sizeof( MeshVertexP ) );
SimpleShaders::EndShader();
}
```

3 개의 열을 변경하면 로드-힛-스토어가 제거되고 무결 상태가 된다.

```
VOID DebugDraw::DrawRing( const XMFLOAT3 &Origin,
const XMFLOAT3 &MajorAxis, const XMFLOAT3 &MinorAxis, D3DCOLOR Color )
{
static const DWORD dwRingSegments = 32;
MeshVertexP verts[ dwRingSegments + 1 ];

XMVECTOR vOrigin = XMLoadFloat3( &Origin );
XMVECTOR vMajor = XMLoadFloat3( &MajorAxis );
XMVECTOR vMinor = XMLoadFloat3( &MinorAxis );

FLOAT fAngleDelta = XM_2PI / (float)dwRingSegments;
FLOAT fi = 0.0f; // Added a copy of i as a float
for( DWORD i = 0; i<dwRingSegments; i++, fi+=1.0f ) // Inc fi
{
FLOAT fAngle = fi * fAngleDelta; // NO int to float conversion
XMVECTOR Pos;
Pos = XMVectorAdd( vOrigin, XMVectorScale( vMajor, cosf( fAngle ) ) );
Pos = XMVectorAdd( Pos, XMVectorScale( vMinor, sinf( fAngle ) ) );
XMStoreFloat3( (XMFLOAT3*)&verts[i], Pos );
}

verts[ dwRingSegments ] = verts[0];

SimpleShaders::SetDeclPos();
SimpleShaders::BeginShader_Transformed_ConstantColor
( g_matViewProjection, Color );
g_pd3dDevice->DrawPrimitiveUP( D3DPT_LINESTRIP, dwRingSegments, (const
VOID*)verts, sizeof( MeshVertexP ) );
SimpleShaders::EndShader();
}
```

결론

조금만 연습하면 별 무리 없이 무결한 코드를 사용할 수 있을 것이다. 그러나 주의를 기울이지 않으면 성능 저하를 초래하는 코드를 사용하기가 쉽다. MS PIX 툴 등을 사용하면 이를 파악하는데 유용할 것이나 성능 저하를 피하려면 이들이 어떻게 존재하는지를 파악하고 처음부터 코드를 사용하지 않는 것이 가장 바람직하다.

현대 게임 프로그래밍에서 하드웨어를 완벽하게 이해할 필요는 없으나 프로그래머가 CPU 와 메모리 하위시스템 간의 상호작용하는 및 CPU 가 작용 원리에 대해 확실한 기본 지식을 가지고 있다면 성능이 극대화된 SW 를 만들 수 있을 것이다.