높은 성능의 C++ 트레이딩 시스템

wsong0101 · 2020년 9월 17일

C 트레이딩

https://www.youtube.com/watch?v=7FQwAjELMek
위 영상의 내용을 정리했습니다.

낮은 레이턴시를 위한 프로그래밍 기술

느린 부분을 제거

이런 코드는 피하고

```
if (checkForErrorA())
    handleErrorA();
else if (checkForErrorB())
    handleErrorB();
else if (checkForErrorC())
    handleErrorC();
else
    sendOrderToExchange();
```

이런 코드를 작성하라

```
int64_t errorFlags;
    ...
if (!errorFlags)
    sendOrderToExchange();
else
    HandleError(errorFlags);
```

아래 코드가 더 빠른 이유는 뭘까? 캐시 예측(cache prediction)과 분기 예측(branch prediction) 때문이다. 아래 코드에는 하드웨어의 분기 예측이 다룰 분기 개수가 더 적다. 또한 아래 코드는 더 적은 명령어를 생산하기 때문에 명령어 캐시(instruction cache)에 가해지는 부담이 적다. 위

코드의 경우 에러를 핸들링하는 함수가 안에서 무슨 일을 할지 모르기 때문에 데이터 캐시(data cache)가 더 사용될 수도 있다. 반면 아래 코드는 int 하나만 사용한다.

템플릿 기반의 설정

설정 파일에 기반해 동작을 구분해야 할 일이 있다면, 가상 함수를 많이 사용한다. 하지만 분기의 개수를 미리 알고 있다면 템플릿을 사용할 수 있다.

```
struct OrderSenderA {
    void SendOrder() {
        ...
    }
};
struct OrderSenderB {
    void SendOrder() {
        ...
    }
};

template <typename T>
struct OrderManager : public IOrderManager {
    void MainLoop() final {
        ...
        mOrderSender.SendOrder();
    }
    T mOrderSender;
};

std::unique_ptr<IOrderManager> Factory(const Config& config) {
    if (config.UseOrderSenderA())
        return std::make_unique<OrderManager<OrderSenderA>>();
    else
        return std::make_unique<OrderManager<OrderSenderA>>();
}

int main(int argc, char *argv[]) {
    auto manager = Factory(config);
    manager->MainLoop();
}
```

컴파일 시간에 알 수 있는 정보를 활용해 코드 속도를 빠르게 하자. C++17에서는 if constexpr을 사용해 같은 구현을 좀 더 간단하게 할 수 있다.

람다 함수는 빠르고 편하다

람다를 사용한다고 더 빨라지는 것은 아니지만 성능 저하가 없고 인라인화도 잘 되기 때문에 잘 사용하면 좋다.

메모리 할당

메모리 할당은 비싸기 때문에 미리 할당된 오브젝트를 사용하는 것이 좋다.

delete는 시스템 함수를 호출하지는 않지만 (OS로 메모리를 돌려주지는 않는다) free 호출의 코드가 400라인 정도 되기 때문에 재사용 하는 것이 좋다. 메모리 파편화 방지에도 도움을 준다.

만약 큰 오브젝트를 반드시 delete 해야 한다면 다른 스레드에서 하는 것을 고려해보라.

C++에서의 exception

exception을 사용하는 것 자체에 비용이 발생하지는 않는다. throw가 발생했을 때에만 비용이 발생하기 때문에 사용을 너무 무서워할 필요는 없다.

분기 보단 템플릿을 선호하라

```
template <Side T>
void Strategy<T>::RunStrategy() {
    const float orderPrice = CalcPrice(fairValue, credit);
    CheckRiskLimit(orderPrice);
    SendOrder(orderPrice);
}
template<>
float Strategy<Side::Buy>::CalcPrice(float value, float credit) {
    return value - credit;
}
template<>
float Strategy<Side::Sell>::CalcPrice(float value, float credit) {
    return value + credit;
}
```

모든 if를 이렇게 제거할 필요는 없지만 hotpath 에서는 유의미하다.

멀티 스레딩

피하는 것이 최선이다.

- 데이터를 lock을 통해 동기화 하는 것은 비싸다.
- lock free 코드라고 해도 하드웨어 레벨에서는 lock이 필요하다.
- 제대로 병렬 처리를 구현하는 것은 매우 어렵다.
- producer가 consumer를 앞서는 경우가 생기기 쉽다.

만약 반드시 써야 한다면..

- 공유 데이터를 정말 최소화 하라. 멀티 스레드가 같은 캐시라인에 쓰는 것은 비싸다.

- 데이터를 공유하는 대신 복사할 수 있는 방법을 고려하라.
- 데이터를 공유해야 한다면 동기화를 하지 않는 것을 고려하라.

데이터 접근

```
struct Market {
    int32_t id;
    char shortName[4];
    int16_t quantityMultiplier;
    ...
};
struct Instrument {
    float price;
    int16_t quantityMultiplier; // 중복 데이터
    ...
    int32_t marketId;
};
```

만약 Instrument의 marketId를 통해 Maket을 찾고, 그 Market의 quantityMultiplier를 가져와야 한다면, 위 처럼 Instrument에 중복 데이터를 넣어 속도를 높히는 방법을 고려해볼 수 있다. 어차피 Instrument에 접근해 price를 가져와야 한다면 같은 캐시라인(64바이트)에 넣어서 추가 비용 없이 데이터에 접근할 수 있다.

빠른 연관 컨테이너(std::unordered_map)

std::unordered_map을 사용할 경우 해시 키가 중복되면 linked list를 만든다. 해시 키가 중복될 경우 정말 worst case에는 O(N)을 기대하게 될 수도 있다.

이 경우 구글의 dense_hash_map 같은 컨테이너를 고려해볼 수 있다. key-value가 포인터 대신 연속된 메모리에 올라가있는 자료구조이다.

Optiver(발표자의 회사)에서는 이 둘을 섞은 하이브리드 구조를 사용한다. key에 이미 hash가 된 key를 사용하고, value에는 포인터가 들어간다. 이 둘의 크기가 16 바이트이기 때문에 한 캐시라인에 연속되는 4개의 key-value가 한 번에 들어간다. 따라서 충돌이 일어날 경우에도 메모리를 건너뛰지 않고 바로 연속되는 key와 비교하며 캐시를 효율적으로 사용할 수 있다.

((always_inline)) 과 ((noinline))

attribute((noinline)) 을 사용해 hotpath에 원하지 않는 분기문 등이 들어가는 것을 방지할 수 있다. 다만 이 속성들을 주의해서 사용하고 꼭 성능을 측정해보자.

캐시를 hot 하게 유지하기

이 부분에서 5ms 정도의 성능 차이가 나기 때문에 가장 중요한 부분이다.

실제로 hotpath가 실행되는 경우는 많지 않다. 그렇기 때문에 캐시를 hot 하게 유지하기 위해서 실제 hotpath가 실행되는 것과 동일한 흐름을 매번 실행하는 것으로 데이터 캐시와 명령어 캐시의 우선 순위를 유지할 수 있다. 또한 하드웨어 분기 예측도 원하는 방법으로 학습시킬 수 있다.

인텔 제논 E5 프로세서

인텔 엔지니어를 존중하는 방법은 아니지만.. 하나를 제외한 나머지 코어에서 L3 캐시 사용을 꺼버릴 수 있다.

작은 부분의 최적화

Placement new는 약간 비효율적이다

```
#incldue <new>
Object* object = new(buffer)Object;
```

많은 컴파일러에서 placement new를 호출하면 내부에서 널체크를 한다. 이 한 번의 널체크로 인해 inline 여부가 결정될 수도 있다.

작은 문자열 최적화 지원

```
std::unordered_map<std::string, Instrument> instruments;
return instruments.find({"IBM"}) != instruments.end();
```

gcc 5.1 이상에서 15 캐릭터 이하의 문자열은 할당을 하지 않는다. 하지만 Redhat/Centos/Ubuntu/Fedora에서 문자열을 사용한다면 이전 버전과의 호환을 위해 아직도 예전 std::string 구현을 사용하기 때문에 느리다.

C++11에서 static 지역 변수 초기화를 사용했을 때의 오버헤드

```
struct Random {
    int get() {
        static int i = rand();
        return i;
    }
};
```

아주 작긴 하지만 어셈블리 레벨에서 변수 i에 접근할 때마다 값이 초기화 되었는지 여부를 확인하기 때문에 오버헤드가 존재한다.

std::function에서 할당이 일어날 수 있다

컴파일러에 따라 사용하지 않는 값을 캡쳐할 경우에도 new를 호출하는 경우가 있다.
std::function을 대체할 수 있는 inplace_function을 사용한다면 function이 스택에 머물 경우 클로저를 위한 버퍼도 스택에 생성해 new를 방지할 수 있다. (std 구현은 아님)

std::pow는 느릴 수 있다

```
auto base = 1.0000000000001, exp1 = 1.4, exp2 = 1.5;
std::pow(base, exp1) = 1.0000000000140 // 53 ns
std::pow(base, exp2) = 1.0000000000151 // 479195 ns
```

std::pow는 초월함수(transcendental function)이기 때문에 첫 번째 페이즈에서 충분히 정확한 값을 구할 수 없다면 두 번째 페이즈로 넘어간다.

시스템 함수 호출을 피하라

커널 레벨로 전혀 접근하지 않는 것이 좋다.

성능 측정

샘플링 프로파일러 (e.g. gprof)

- hotpath가 실행되는 순간은 짧기 때문에 전체적으로 보면 아무것도 하지 않는 것처럼 보인다.

명령어 프로파일러 (e.g. callgrind)

- I/O 적인 부분은 찾아낼 수 없기 때문에 정확하지 않다.

마이크로 벤치마크 (e.g. Google benchmark)

- 실제 환경을 반영하지 않는다.
- 테스트 코드를 최적화해서 없애지 않도록 신경써야 한다.
- 힙 메모리 조각화 상태가 성능에 영향을 줄 수 있다.

실제와 거의 동일한 환경을 구축하고, 서버에 오고가는 네트워크 패킷을 감지하여 성능을 측정한다. 매우 만들기 어렵지만 실제와 가장 흡사하다.

요약

- 컴파일러와 C++에 대한 지식이 있어야 한다.
- 머신 아키텍쳐에 대한 기본 지식이 있어야 한다. 이게 코드에 어떤 영향을 주는지도 알아야 한다.
- 런타임 로직은 아주 간단하게 가져가는 것을 목표로 한다.
  - 컴파일러는 간단한 코드를 가장 잘 최적화한다.
  - 필요하다면 완벽한 정확도 보다 유사치를 선호하라.
  - 비용이 큰 작업은 시간이 남을 때 한다.
- 정확하게 측정하라 (가장 중요)

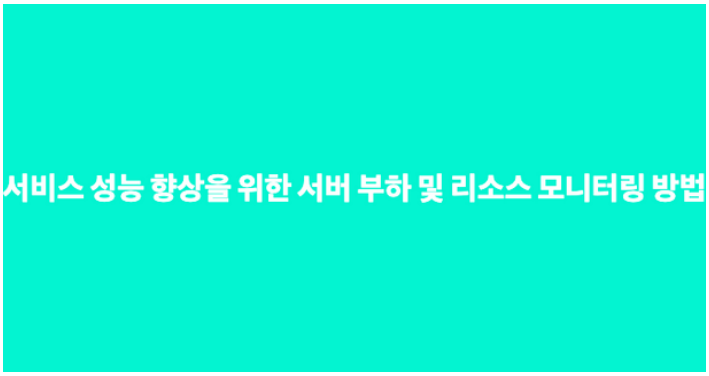
WSong

개발새발

다음 포스트
[NDC2021] 정리 - <쿠키런: 킹덤> 서버 아키텍처 뜯어먹기!-천만 왕국을 지탱하...

이전 포스트
이펙티브 모던 C++ 정리

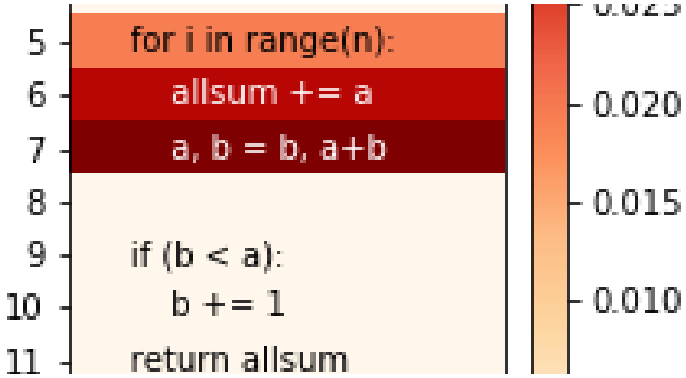
관심 있을 만한 포스트



서비스 성능 향상을 위한 서버 부하 및 리소스 모

개발자의 기본 "인프라" 란 ?

친절한 SQL 튜닝 - 1장



Jupyter Notebook에서 Magic Command로

0개의 댓글

댓글을 작성하세요

댓글 작성