Unsafe at any Speed

High Speed Low Latency Systems in C#

Using Targeted Memory Management

Chris Condron
Solaris

In vivo open air fluorescence imaging system with real time video for fluorescence guided surgery from PerkinElmer.
Core Workflow

Applying the 80/20 rule most of the application is Hardware Control, UI, and Business Logic.

The 20% that cares about high performance can be isolated to the image processing pipeline.
static unsafe void Main() {
    const int arraySize = 20;
    int* fib = stackalloc int[arraySize];
    int* p = fib;

    // The sequence begins with 1, 1.
    *p++ = *p++ = 1;
    for (int i = 2; i < arraySize; ++i, ++p) {
        *p = p[-1] + p[-2];
    }
    for (int i = 0; i < arraySize; ++i) {
        Console.WriteLine(fib[i]);
    }
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}

The Epiphany
How do I organize my code and data to minimize memory usage, cache misses, and so on? My first-order answer is
- don't store data unnecessarily,
- keep data compact, and
- access memory in a predictable manner.

Bjarne Stroustrup

Source C# stackalloc Documentation MSDN
Pipeline Architecture

Responsibility segregated into 3 primary components

- Memory management
- Flexible Processing Pipeline
- Algorithms
Approach

• Most of the App will use regular GC
• The unmanaged memory should be wrapped in managed classes with IDisposable
• Unsafe algorithms should only deal in pointers and structs
• No-one else should see them
• Always think in RAII, be deterministic with your finalizers
• Nail up your memory as soon as possible and hold it for the lifetime of the app
• Use allocation sizes are large enough to cause the allocation in the Large Object heap.
• Don’t bother with Fixed or other short term pinning options or marshalling
Cache and Release

The lifetime of an unmanaged resource received from and outside caller cannot last beyond the end of the function call

- Key Design constraint
- To retain locally, a copy must be made
  - memcpy is really fast
- This allows the caller to dispose/return the object at the end of the call
- Key to maintaining sanity in the code!
Block Buffer Memory Manager

- Simple implementation
- Manager owns private buffers
- Callers can request wrapped buffer
- Check-Out Check-In Via Wrapper
- RAII via IDisposable at each level
- Single use case, keep it simple
**Block Buffer Memory Manager**

**Key Points**

- GCHandle.Alloc copies memory!
- Allow the original value to go out of scope on the stack to avoid excess memory consumption
- Don’t try to use the original
- Create a copy of the pointer to allow write access
- Keep it simple, each Manager type is for a single use case.
- Each layer implements IDisposable
- Call dispose explicitly, do not allow GC to clean up.
  - Keep things as deterministic as possible
```csharp
public unsafe void Subtract(ushort* fore, ushort* back, ushort* counts) {
    for (int row = 0; row < height; row++){
        for (int c = 0; c < width; c++){
            if (fore[c] <= (back[c] + Threshold))
                *counts = 0;
            else{
                diff = fore[c] - back[c];
                if (diff > maxShift)
                    *counts = ushort.MaxValue;
                else
                    *counts = (ushort)(diff << ScaleFactor);
            }
            counts++;
        }
        fore += stride;
        back += stride;
    }
}
```
Algorithms

Key points

• C++ style implementation
• Can use all of the best practices for performance from C++
• Easy to port C++ algorithms directly, just copy and fix syntax.
• Use pointers as function parameters to clarify correct usage and maintain separation of concerns.
• Often managed in a dedicated project
• Experts only
• Only used for targeted areas in the application
• Most pointers mapped to structs
//Declare
private BufferBlock<MonoImageSet> _frameQueue;
private TransformBlock<MonoImageSet, MonoImageSet> _postToStorage;
private TransformBlock<MonoImageSet, MonoImageSet> _subtract;

//Define
_postToStorage = new TransformBlock<MonoImageSet, MonoImageSet>(p =>{
_currentStorageAction?.Invoke(p);

_startDateTime = DateTime.UtcNow;

return p;}, SingleProducer);

//Link
_frameQueue.LinkTo(_videoRoi);
_videoRoi.LinkTo(_postToStorage);
_postToStorage.LinkTo(_subtract, i => _engageAccumulation);
Data Flow

Key points

- Import via NuGet package
- Set options to single producer
- Allows conditional linking
- Pass custom message type with pointers to the data objects
- Define blocks with lambda functions
- Supports the basic messaging patterns in *Enterprise Integration Patterns*
Unsafe Structs

```csharp
[StructLayout(LayoutKind.Sequential)]
public unsafe struct VideoFrameHeader
{
    public long FrameNumber;
    public fixed byte FrameId[16];
    public fixed byte VideoId[16];
    public double OffsetMilliseconds;
}

[StructLayout(LayoutKind.Sequential)]
public unsafe struct MonoVideoFrame
{
    public long FrameNumber;
    public fixed byte FrameId[16];
    public fixed byte VideoId[16];
    public double OffsetMilliseconds;
    public fixed byte pixels[MonoImage.SourceBytes];
}
```
• Use StructLayout.Sequential
• Blittable types only!
  • No unfixed arrays
• Allows the alignment of memory to optimize CPU utilization
• Can be mapped to match other structs
  • Header implementation
  • Pointers can be cast to any type
• Can be mapped to match C++ classes
  • Avoid marshalling overhead
  • Mixed mode applications
• Create wrapper classes for interaction with managed code
Object Pooling Memory Manager

Second use case for Targeted Memory Management

Ideal for small items like Messages

- Rolling set of lockless forward allocating buffers
  - Keep a simple count of allocations deallocation
  - When allocation reaches the end grab the next free buffer

- Matching memory structs on messages for common header with message type and size information

- Handlers take a header pointer and cast to correct type based on the type indicated in the header

- Messages subscription requires a explicit type to be specified rather than by overloads or type inference

- Messages containing large data payloads can use a combination of this and a Block Buffer Memory Manager
Other Best Practices

For High Performance Sections

n.b. A call to main memory is 200-500 instruction cycles

To the hardware a 1d array is the only data structure

• Avoid
  • Task.Run: use a handful of dedicated Threads
  • Linq
  • Foreach
  • Non-Blittable Types
  • Division: Use bitwise operations if possible
  • Doubles: use integer types where possible
  • Matrixes: Use vectors instead (row major)

• Use
  • Structs
  • Arrays
  • For loops
  • Bit-shift operations
  • Linear and predictable memory access: Pre-fetchers are your friend, (so is a shared L2 cache)

• Test performance, and redesign your algorithms based on what you find
Targeted Memory Management

Summary

• Get to have your cake and eat it too
• GC for most of the app
• Near native speed for critical sections
• Access to a large body of work and best practices on high performance code in C++
• Avoid excessive GC thrashing via either memory size or object creation volume
• Keep things simple and strongly separated