More Useful Computations in the Same Duration:

Optimizing Embedded Hard Real-Time Code in C++

By Scott Schurr for C++ Now! 2012

A Travelogue of Sorts

What is Real-Time Software?

- Operations have a specific time deadline

What if I miss my deadline?

A continuum between…

- Hard real-time: missile guidance system
- Soft real-time: streaming audio

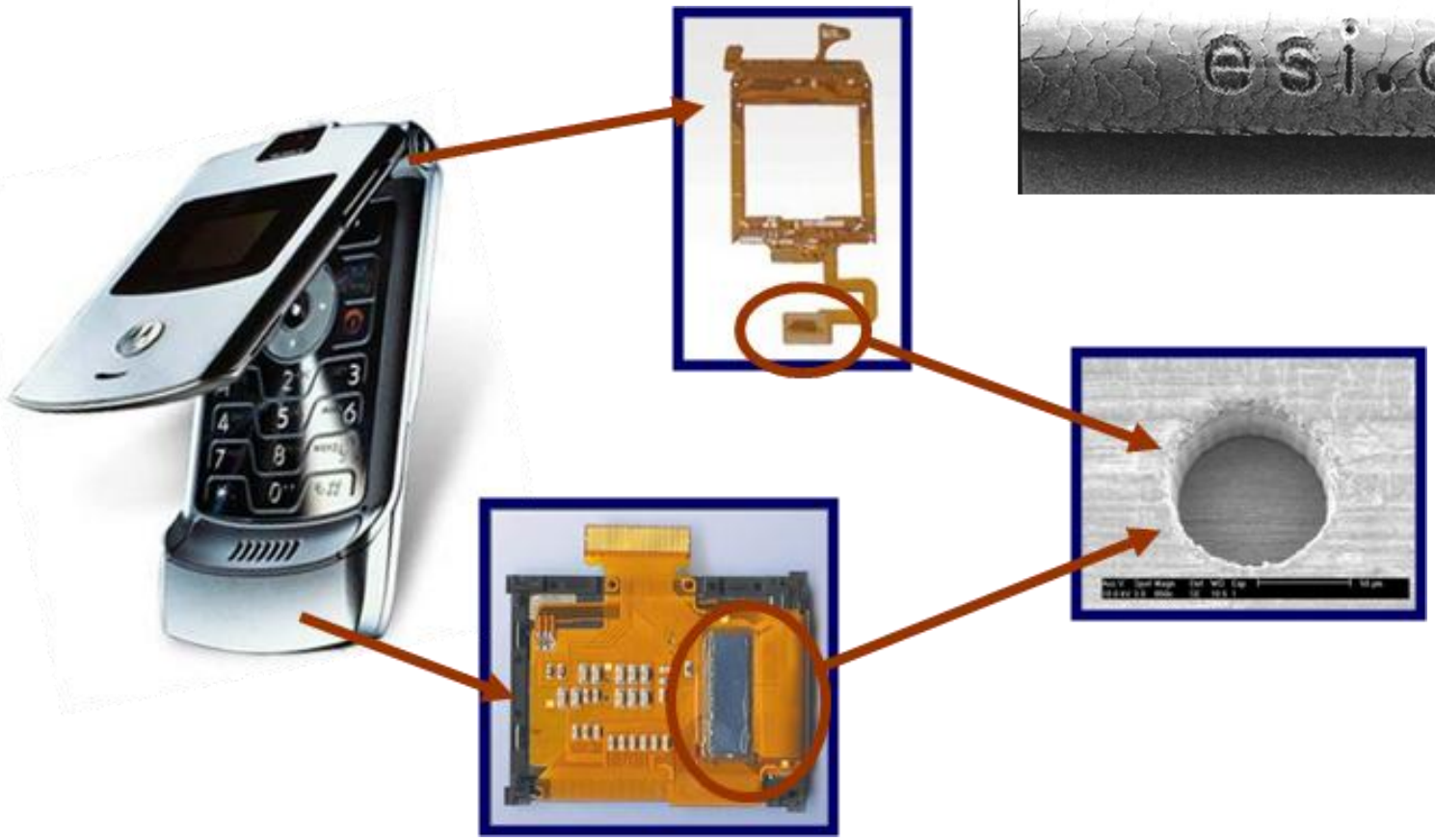
What is an Embedded System?

- Purpose-built: not a general-purpose computer
- Often constrained by
  - Heat dissipation
  - I/O (e.g., no monitor)
  - Power
  - Size
  - Memory

I Work On…

It Does Laser Cutting

human hair



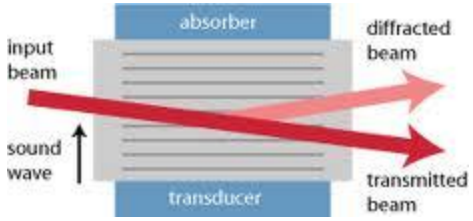
The Hardware

Laser

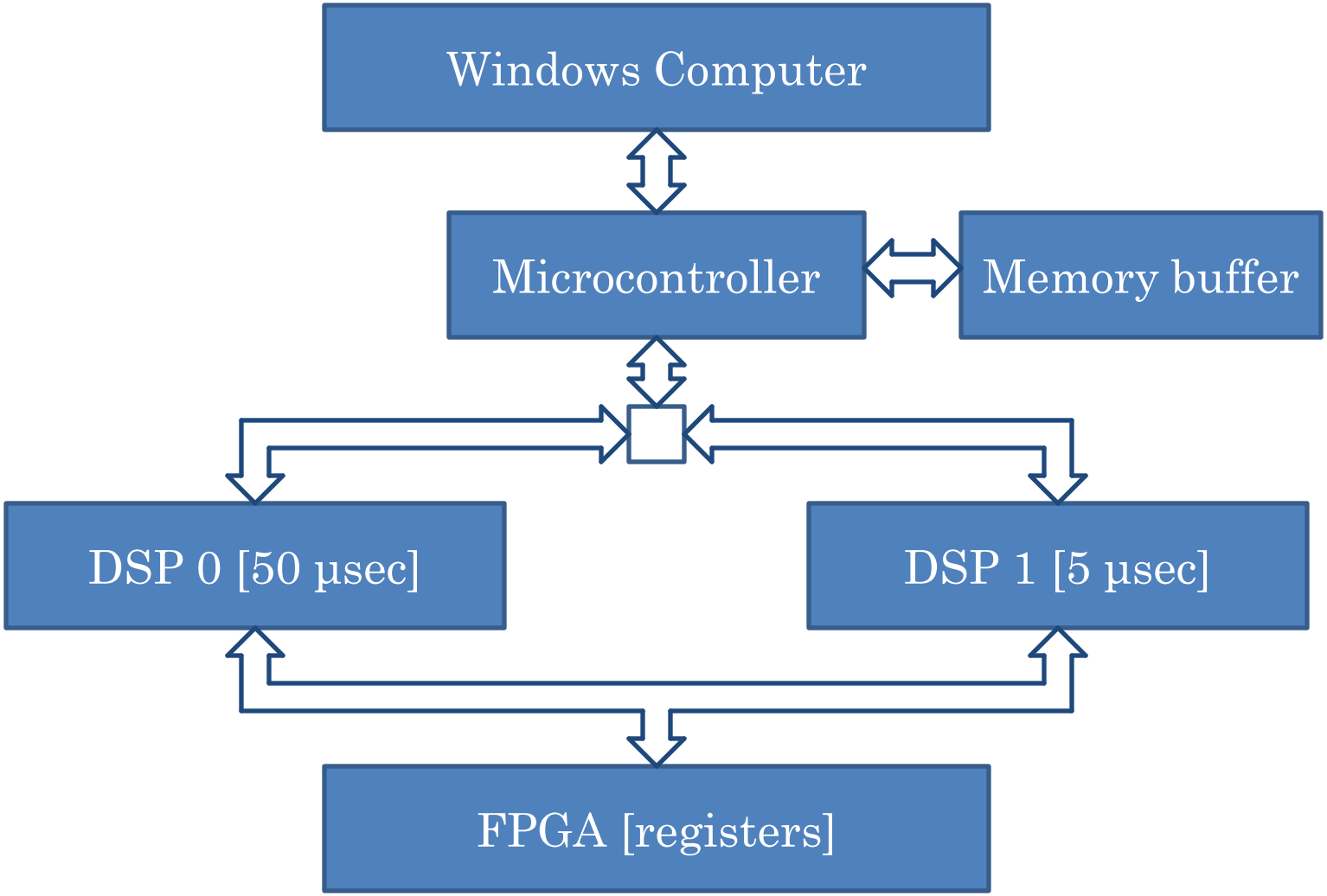
X-Y linear stage

X-Y mirror galvanometer

X-Y acousto-optic deflector



The Compute Platform



The Compute Platform

- Windows control computer
  - C# and .NET (mostly)
  - Very non-real-time
- Analog Devices Blackfin Microcontroller
- Two Analog Devices TigerSHARC DSPs
  - 600 MHz static superscalar
  - Harvard architecture
  - 3 MB internal memory each

Two Embedded Build Targets

- It's all C++ (and a hair of assembly)
- Windows (about 95%)
  - For a software-only simulation
  - Allows breakpoints in "real-time" code
- Analog Devices Processors
  - Part of VisualDSP++ tool kit
  - Surprisingly compliant with C++98
  - Includes a cycle-accurate processor simulation

The Embedded Software Platform

- The simulation environment runs entirely under Windows. No hardware.
- Requires about 95% of the code to compile for Windows and TigerSHARC
- Only the TigerSHARC compiled code needs to meet timing constraints
- Building for two compilers is a short-term pain, but results in better code

Observation: Good Tools

If you use a great debugger…

…you'll write code that can only be debugged with a great debugger

This is a serious problem when something goes wrong, but only on the hardware platform

Unusual Considerations

- No exceptions
- No operating system
- Limited memory [3 Mbytes per DSP]
- No breakpoints or debugger (except in simulation)

No Exceptions

Understand the costs of exception handling

Scott Meyers *More Effective C++* Item 15

- Execution time predictability
- Size of code and tables for exceptions
- Handling exceptions is slow

- No RTTI (often a good thing)

No Operating System, um, mostly

- No file I/O needed
- Heap management by C++ runtime
- No scheduler
  - Improves interrupt response time predictability
  - Reduces interrupt overhead
- No built-in inter-thread communications
  - Must be hand-built

Division of Labor in DSPs

- Both DSPs have a non-interrupt ‘thread’
  - Non-real-time commands
  - Non-real-time responses
  - Error notifications to Windows
- 50 µsec servo interrupt [DSP 0]
  - Dispatch of real-time commands
  - Servo code for linear stages
- 5 µsec servo interrupt [DSP 1]
  - Servo code for galvanometers

5 µsec Is Not Very Much Time

- 3000 DSP1 clocks between interrupts
- Each interrupt must…
  - Save processor state on the stack
  - Read several sensors
  - Handle the position commands
  - Execute several filters
  - Write control registers
  - Restore processor state from stack, and
- Leave time for some non-real-time stuff

Goals for the Embedded Code

- Handle more hardware – acousto-optic deflectors
- Increased coordination between hardware – simultaneous use of stages, galvos, and acousto-optic deflectors
- Increased feature resolution at high beam velocity
- Increased reliability
- No additional embedded compute power

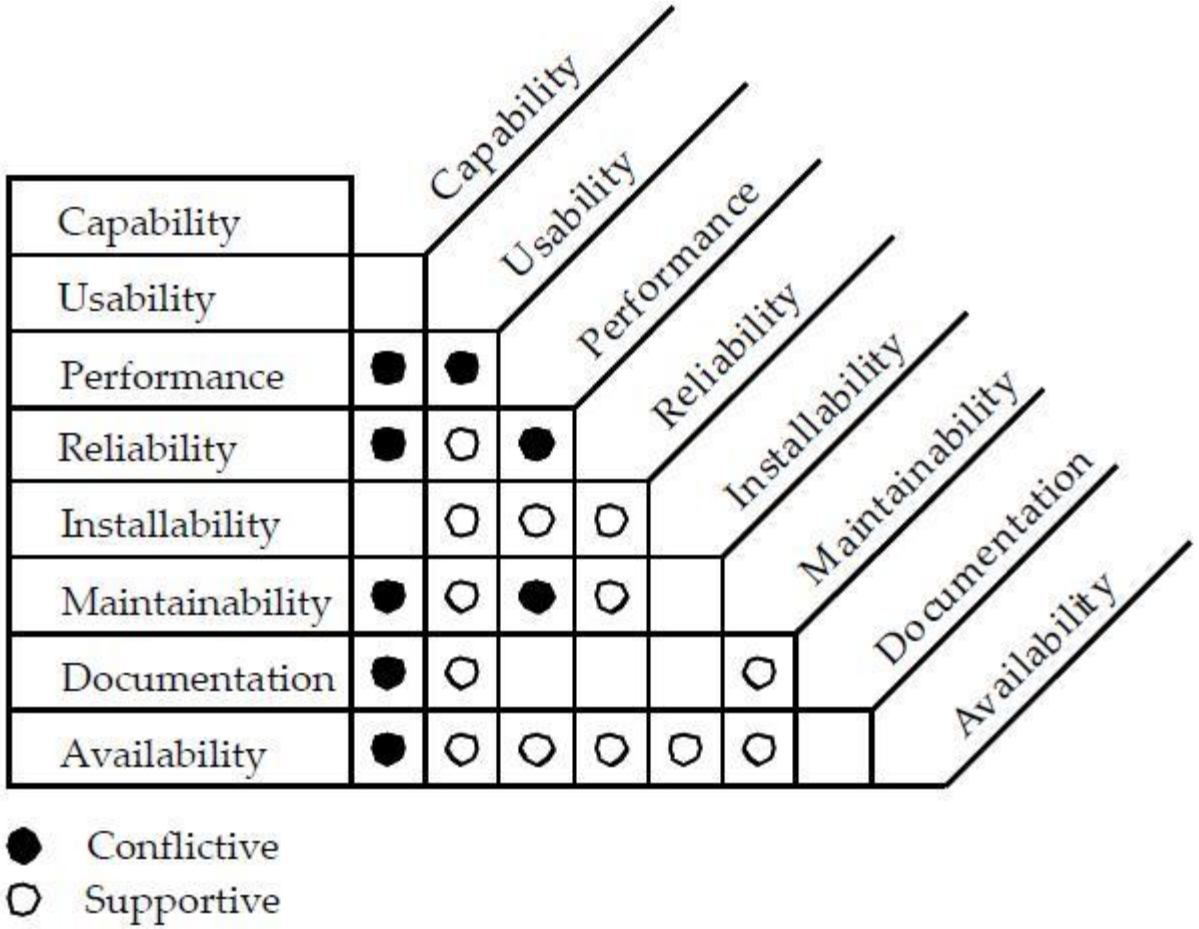
Observation: Start Slow

If you want to optimize a system for speed…

…start with a system that was not optimized for speed

- MATLAB generated servo code
- Commands streamed in XML
- Automatic recordability of computed values in the control path

Kinds of Software Goodness



Kan *Metrics and Models in Software Quality Engineering* 2003 Figure 1.1

Early Questions

1. How to monitor servo interrupt health?
   a. Time spent inside the interrupt?
   b. Are there servo violations?
   c. What is the servo jitter?

2. Why can we only get a command into the embedded system every 200 µsec?

Observations

- Make tools
- Time is not the only important thing to measure
- Consider Using Lazy Evaluation

Meyers *More Effective C++* Item 17

DspHealthGauge

```
class DspHealthGauge {                  // Base class
public:
    static void StreamAllGauges(ResultStream& rs);
    static void ResetAllGauges();
    virtual void Stream(ResultStream& rs) const = 0;
    virtual void Reset() = 0;
protected:
    void PushBack();
    void Remove();
private:
    DspHealthGauge* link_;
    static DspHealthGauge* children_;
};
```

DspHealthGauge

- Construct, stream and reset from non-interrupt “thread”
- Constructor adds to intrusive list
- Judicious use of virtual
- Low overhead setting in interrupt code
- Most processing postponed until streaming
- Similar effects possible in plain C, but with much more maintenance.

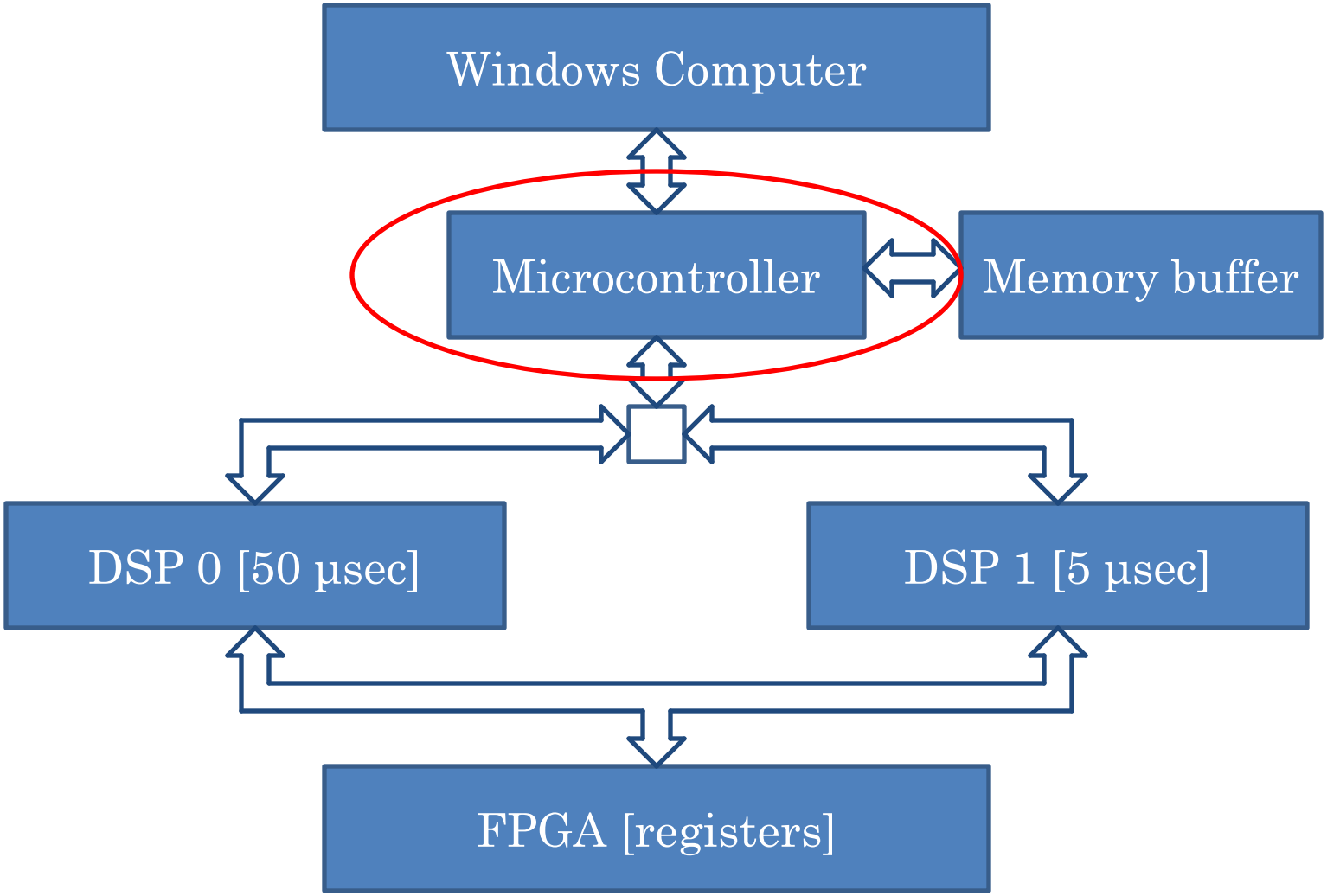
DspHealthGauge

- Permanent reports
  - Servo interrupt characterization
  - Galvo response
- Temporary measurements
  - For focused characterization
- Add new health gauge types easily
  - Policy based derived classes
  - 17 types currently defined in system

Observation: Make Measurements

- Why can we only get a command into the embedded system every 200 µsec?
- I **knew** where the problem was
  - A piece of code that annoyed me
- I measured to prove my assertion
- I was wrong
- Further measurements showed the real problem

The Culprit


Windows Computer
Microcontroller
Memory buffer
DSP 0 [50 µsec]
DSP 1 [5 µsec]
FPGA [registers]

The Culprit

- Message from microcontroller to DSP0 took 280 µsec – more than 200 µsec.
- Microcontroller was supposed to be so simple it **couldn't** be slow
- But, inside…
  - A microkernel scheduler
  - Three threads
  - Code running in external memory

Observation: Memory Matters

- Step 1. Moving code to internal memory in microcontroller removed 100 µsec.
- Lesson for non-embedded folks: remember your processor caches.
- Consider reading *What Every Programmer Should Know About Memory* by Ulrich Drepper, 2007

Observation: Consider Removing Threads

- Turned three threads into a single polling loop
- Delivery times dropped from 150 µsec to 20 µsec.
- Removed microkernel. Now all code fits in internal memory.
- Delivery time dropped under 5 µsec.

The First Goal

- Delivery times improved by more than an order of magnitude. From 280 µsec to under 5 µsec.
- (Much) less code
- Simpler code. No threads.

- Minimum command time is now 75 µsec

Collateral Damage

- Timing changes can reveal additional problems
- Faster microcontroller exposed a timing window in communication throttling
- Required an FPGA code change from level sensitive to edge sensitive

The Next Question

- How can we execute several real-time commands inside one 50 µsec interrupt?
  - Several commands had big initial compute spikes. Some compute spikes almost consumed the interrupt. No way to afford 2.
  - Command format was streamed. Each command took 4 µsec to unstream.

Design for the Worst Case

- You can't design for **every** case
- Identify the most important **worst** case
- Get agreement from the team
- Don't lose focus

Simplify Algorithms

Compute spikes came from re-computing stuff the control computer already knew

1. Convert algorithms from complicated to simple
2. Pass all necessary data from control computer

Turned the embedded system from slightly autonomous into a pure slave

Use Old Fashioned Code

- Replaced stream with POD structs
  - Unpacking time fell from 4 µsec to 0 ns
  - Data size fell to 1/3 original
- Don't reformat data en route
  - Taught the control computer to format for the final destination
- Identify commands with a single enum
  - Changed command dispatch from two run-time searches through vectors to a single dispatch through a switch

Ahem, POD Structs in C#?

- Um, well, no.
- But it can be done in C++/CLI
- We forced C++/CLI in safe mode to produce something close to POD structs
- We used macros so C++98 and C++/CLI could share the header files
- We got both speed and message type safety between the DSPs and the control computer

The Macros

```
#ifdef MANAGED_BUILD_FOR_SCC // For consumption by Control Computer/.NET

#define MIXED_USE_REF_STRUCT         ref struct
#define MIXED_USE_VALUE_STRUCT       value struct
#define MIXED_USE_REF_CLASS          ref class
#define MIXED_USE_VALUE_CLASS        value class
#define MIXED_USE_ENUM               enum class
#define MIXED_HANDLE_OR_PTR          ^
#define MIXED_USE_CONST
#define MIXED_USE_ABSTRACT           abstract

#else                        // For consumption by DSP/DSP simulation

#define MIXED_USE_REF_STRUCT         struct
#define MIXED_USE_VALUE_STRUCT       struct
#define MIXED_USE_REF_CLASS          class
#define MIXED_USE_VALUE_CLASS        class
#define MIXED_USE_ENUM               enum
#define MIXED_HANDLE_OR_PTR          *
#define MIXED_USE_CONST              const
#define MIXED_USE_ABSTRACT

#endif
```

Using the Macros

```
MIXED_USE_VALUE_STRUCT CcncyId
{
public:
    MIXED_USE_ENUM AtlasCcncyEmbeddedId {
        ccncyNone = 0,
        ccncyLine,
        ccncyArc
    };
private:
    AtlasCcncyEmbeddedId ccncyId_;
public:
    inline AtlasCcncyEmbeddedId GetCcncyId() MIXED_USE_CONST {
        return ccncyId_;
    }
    ...
```

Fast Loops

- DSP0 produces 50 ‘points’ in 50 µsec
- So each ‘point’ must take significantly fewer than 600 DSP clocks to compute
- I need fast loops
- How to get them?

Observation: Disassemble

- The TigerSHARC compiler manual encourages looking at disassembly.
- It helped me. A lot.
- It's likely the best way to understand your optimizer.

TigerSHARC Assembly Notes

```
for (int i = 0; i < pointCount; ++i)
{
   SegmentPoint& nomPointOut  = pointsOut->Next();
   nomPointOut.aod = pointsIn[i].first;
   interpolator.ExtractNominalGalvo(
       i, pointsIn, &nomPointOut.x, &nomPointOut.y);
}
//-----------------------------------------------
// Loop at "AtlasBiLinearInterpCal.cpp" line 580
//-----------------------------------------------
// This loop executes 2 iterations of the original
// loop in estimated 32 cycles.
//-----------------------------------------------
// Trip Count  = 24
//-----------------------------------------------
```

Const Loop Lengths

- Encourage hardware loop counters…
- Try this

```
const int pointCount = pointDest->Remaining();
for (int i = 0; i < pointCount; ++i) {
    ...
}
```

- Not this

```
while (pointDest->Full() == false) {
    ...
}
```

Cascade Small Loops

This

```
for (...) {
    ...
}
for (...) {
    ...
}
for (...) {
    ...
}
```

Not this

```
for (...) {
    ...
    ...
    ...
    ...
    ...
    ...
    ...
}
```

No Function Calls Inside Loop

- Non-linear flow puts a bubble in the processor pipeline
- The compiler doesn't know what happened to non-local variables
- There are function calls that don't look like function calls
  - Integer division
  - Modulus
- **Inlined** function calls are okay

#pragma no_alias

```
#pragma no_alias
void myLoop (const int* in, int* out) {
    while (*in) {
        *out = *in;
    }
}
```

- The optimizer **must** assume in* and out* overlap…
- …unless **you** tell it they don't!
- Improves register scheduling and loop unrolling

From the Loyal Opposition

"Noalias is an abomination"

"Noalias must go. This is non-negotiable."

Dennis Ritchie 1988

http://www.lysator.liu.se/c/dmr-on-noalias.html

Copy members to locals

- Declare constant locals of members that will be constant for the loop
- Declare local non-consts for member variables that will change in the loop.
- Use only locals inside the loop
- Remember to write back changed local copies of member variables

Use Intrinsics

- A compiler Intrinsic is specific to that compiler
- It looks like a function call…
- But (usually) the compiler turns it into a single assembly instruction
- Check your compiler documentation
- Every cycle saved inside a loop is multiplied by the loop count

Intrinsics for Multiple Targets

```
#ifdef _WIN32
static inline int builtin_min(int a, int b) { return a < b ? a : b; }
static inline int builtin_max(int a, int b) { return a > b ? a : b; }
static inline float fminf(float a, float b) { return a < b ? a : b; }
static inline float fmaxf(float a, float b) { return a > b ? a : b; }
static inline float copysignf(float dest, float signProvider)
{
   const float sign = signProvider >= 0.0f ? 1.0f : -1.0f;
   return (fabsf(dest) * sign);
}
#else
// TigerSHARC
static inline int builtin_min(int a, int b)
    { return(__builtin_min(a, b)); }
static inline int builtin_max(int a, int b)
    { return(__builtin_max(a, b)); }

// Note: fminf, fmaxf, and copysignf don’t require aliases
#endif // _WIN32
```

Results

For a specific for loop:

- Started with a non-hardware loop
  - Changed to const loop length
- Got a hardware loop with 52 cycles per iteration
  - Applied other techniques
- Finished with a hardware loop with 18 cycles per iteration

Tell The Compiler **Everything**

- The compiler can't see the **value** of a const member.

- But it can see a template parameter

Use Template Parameters…

Turn this

```
Class TCB {
    const int chan_;
public:
    inline void status()
    {
        switch (chan_)
        {
        case 0:
        ...
        }
    }
};
```

To this

```
template <int chan_>
class TCB {
public:
    inline void status()
    {
        switch (chan_)
        {
        case 0:
        ...
        }
    }
};
```

…but, Use Templates Wisely

Factor parameter-independent code out of templates

Scott Meyers *Effective C++ Third Edition* Item 44

Templates Without Code Bloat

Dave Gottner *Dr. Dobb's* August 1, 1995

http://www.drdobbs.com/184403053

Managing Code Bloat

```
Class TCB_impl {
    template<int chan> friend class TCB;

    inline void status(int chan) { ... }
    void source(int chan, int st) { ... }
};

template <int chan>
class TCB {
public:
    inline void status()       { impl_.status(chan); }
    inline void source(int st) { impl_.start(chan, st); }
private:
    TCB_impl impl_;
};
```

Inlining

Good:

- Reduces branching
- Informs the compiler

Bad:

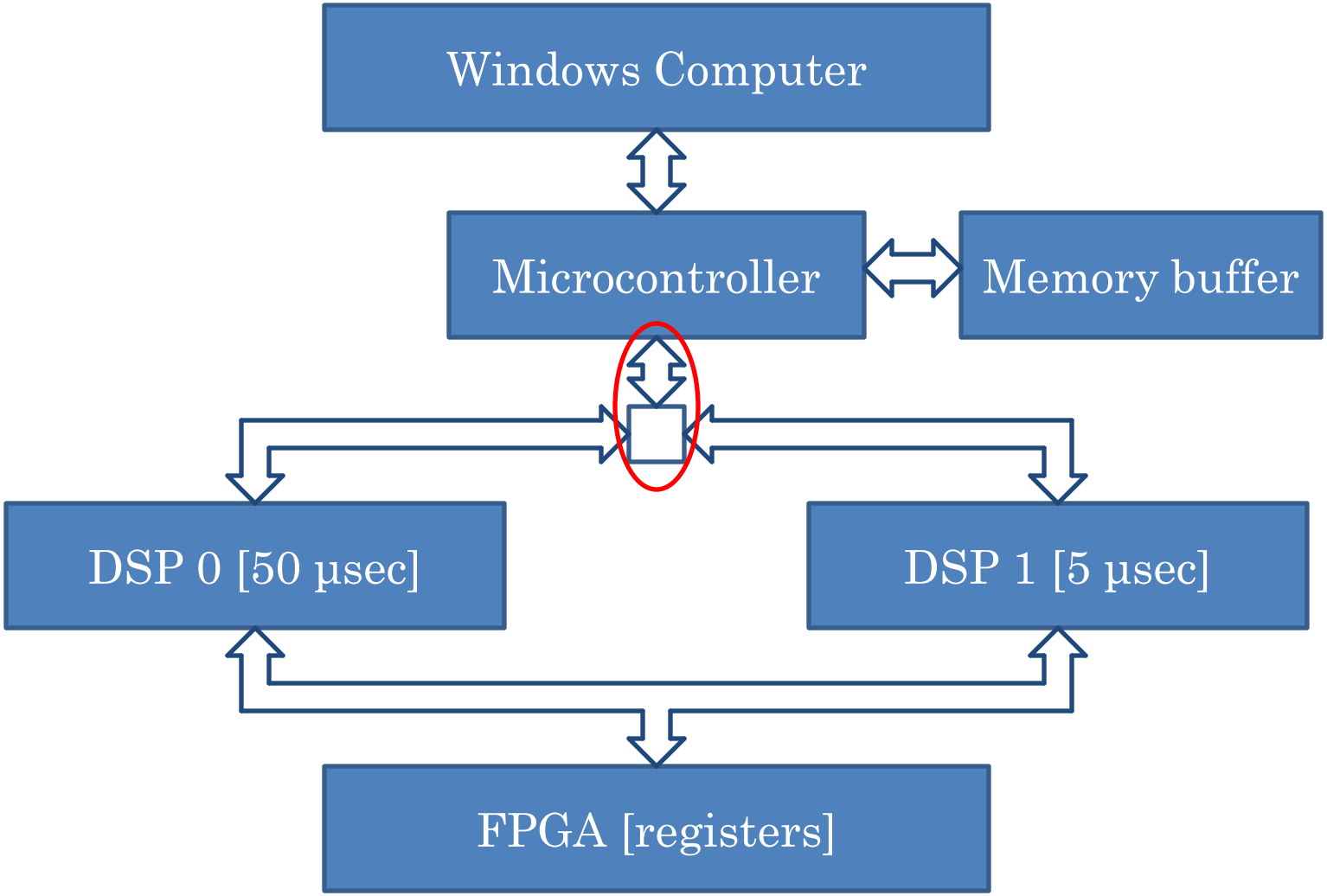
- Increases code size
- Increases local complexity

So Measure!

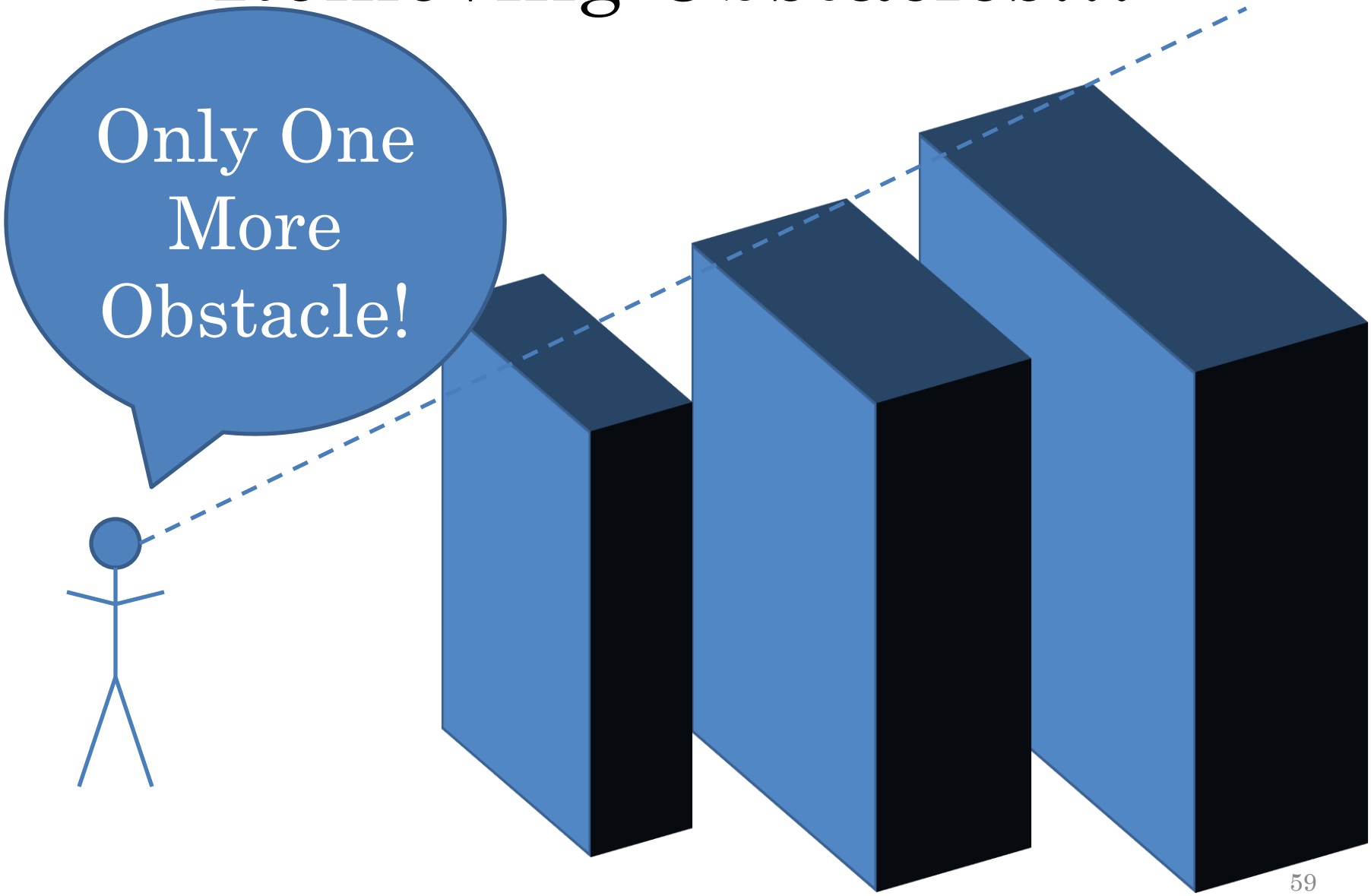
Results

- At the start of our travels we could execute one real-time command every 250 µsec
- At the end we can execute an arbitrarily long stream of 25 µsec commands. Limited by hardware implementation.
- The servo interrupt would support an arbitrarily long stream of 4 µsec commands

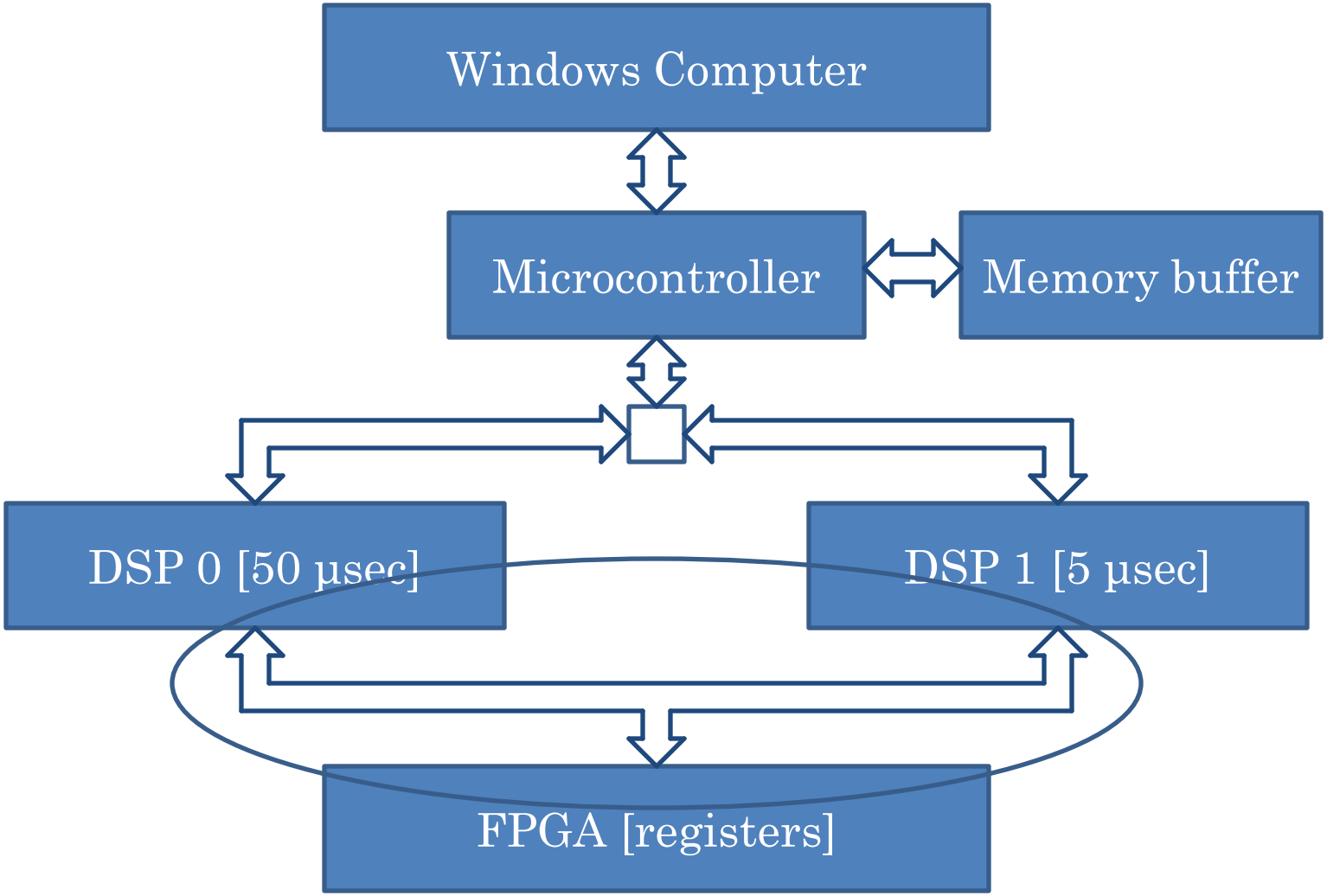
The 25 µsec Limit



Removing Obstacles…



Effects of I/O



I/O and Compute Time

- Internal memory access takes 1.66 ns
- External register write takes 60 ns
- External register read takes 160 ns
- Bus ownership change takes 140 ns
- External reads by the core stall the core
- A contended read by one core can stall…

  140 + 160 + 140 + 160 = 600 ns

- That's over 1/10th a DSP1 servo cycle

I/O and Servo Jitter

- Register reads cause servo jitter
- The core must complete a read
- Interrupts are stalled until the read completes
- That's 600 ns of jitter on the servo interrupt (worst case)
- The worst case doesn't happen often, but it **does** happen sometimes

What to Do About I/O?

- Allow only one DSP on the shared bus
- Keep the cores from doing I/O

- Keep the cores from register reads?

- Yes, with DMA [Direct Memory Access]

I/O with DMA

- DMA channels are coprocessors
- Their only purpose is to move data
- Once they are programmed they run independent of the core
- DMA copies data from shared bus to internal memory or vice versa

Downsides:

- They must be programmed (takes core time)
- They add yet another ‘thread’
- Increased latency (time from read to use)

I/O with DMA

- The TigerSHARC has 14 DMA channels
- We use five DMA channels in DSP1:
  1. DMA shared bus reads and writes
  2. DMA data to DSP0
  3. DMA data from DSP0
  4. DMA commands from microcontroller
  5. DMA command response to microcontroller
- All of these DMAs offload the core

The Core and DMA

1. Core starts DMA **before** it needs data
2. Together…
   a) DMA runs in the background while
   b) Core does other useful work
3. When core needs data it checks for DMA completion by…
   a) Polling (what we use) or
   b) Interrupt

DMA Benefits

- Removed 0.5 to 1.5 µsec from DSP1 servo time (which we promptly used elsewhere)
- Significantly reduced servo interrupt jitter

Non-Embedded DMA (maybe)

- Intel I/O Acceleration Technology
- Intel "Virtualization Technology for Directed I/O" (VT-d)
- AMD I/O Virtualization Technology, "AMD-Vi"
- ARM DMA-230 and -330 Controllers
- Windows DMA API (kernel mode)
- Linux DMA API (kernel mode)

Thread Safety Options

Threads, it's never threads apocryphal ESI quote

Threads, on the other hand, are wildly nondeterministic. The job of the programmer is to prune away that nondeterminism.

Edward A. Lee *The Problem With Threads*, Technical Report No. UCB/EECS-2006-1, January 10, 2006

Thread Safety Options

- OS-level locks
  - I have no OS. Spin locks also won't work
- Atomic instructions
  - TigerSHARC has none
- Volatile
  - Can guarantee order but not counts
- Atomic data arrival
  - Use special (large) data types
- Disable interrupts
  - Works for me, not for multi-core systems

Atomic Data Arrival

- Useful when a set of data must arrive simultaneously
- Create a union in a large data type
- TigerSHARC supports 128-bit integers

- No locking of any sort required
- Imposes alignment requirements for atomic arrival

Prefer Native Data Types

- The TigerSHARC does not natively support…
  - 8-bit char
  - 16-bit short
  - 64-bit double
- They are synthesized in software
- Very inefficient
- TigerSHARC defaults to 32-bit chars

Summary

1. Big improvements take new algorithms.
2. Design for the worst case.
3. Make measurements. Make tools.
4. Know your optimizer. Read disassembly.
5. Tell the compiler everything.
6. Don’t let fashion lead you to slow code.
7. Memory and I/O matter, sometimes lots.

Thanks To Coworkers…

- Rick Coates, Mr. DMA
- Alex Myachin, for Windows solutions
- Guang Lu, the controls master
- Serge Ioffe, hardware and FPGA ace
- Mark Unrath, system design
- Mike Tyler, boss and encouragement

Questions?

Thanks for attending

After Thoughts…

The four languages most often reported as the primary language for embedded projects for the years 2005 to 2012, along with linear trendlines.

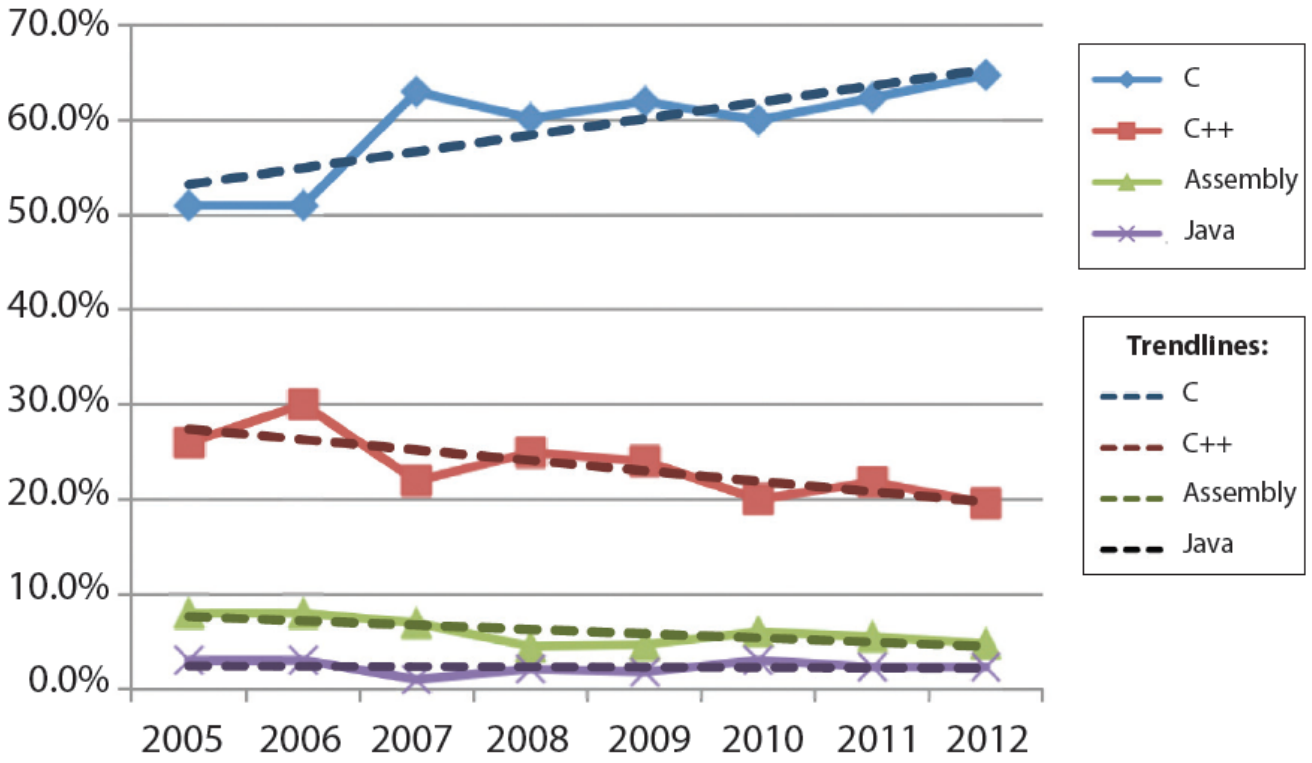


Figure 2

From Dan Saks *Programming Pointers* 5/2/2012
http://www.eetimes.com/discussion/programming-pointers/4372180/Unexpected-trends