Building Demanding Hard Real-Time Systems with Software Thread Integration

Shobhit Kanaujia, Benjamin Welch, Adarsh Seetharam, Deepaksrivats Thirumalai, Alex Dean
alex_dean@ncsu.edu
Center for Embedded Systems Research
Department of Electrical and Computer Engineering
North Carolina State University
www.cesr.ncsu.edu/agdean/stiglitz
Compile to Eliminate Context Switches Where They Limit Performance

- Create efficient implicitly multithreaded (integrated) functions
- Use a compiler at design time to create the functions (compile for low-cost concurrency)
- Build the task/function scheduling decisions into the scheduler (if used) or the ISRs

Break down the barrier between task scheduling (scheduler and dispatcher) and instruction scheduling (compiler)
Hardware to Software Migration

• What is it?
  - Replacing hardware components with software functions on a conventional CPU (uniprocessor)

• Why do it?
  - Custom HW is too expensive unless production volume is high (1M$ mask set)
  - Cost, size, reliability, weight, power
  - Function availability, time to market, field upgrades

• Who does it?
  - Everyone

• Why do they hate it so passionately?
  - Code in assembler and it takes forever to develop
    • Tools are real-time-ambivalent
  - Code in C and you can’t get decent performance, but you finish coding sooner
    • Slow context switches and scheduling
    • Tools are real-time-oblivious

Cost of Microcontroller Throughput

<table>
<thead>
<tr>
<th>Year</th>
<th>Price per MIPS (US ¢)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>60</td>
</tr>
<tr>
<td>1996</td>
<td>40</td>
</tr>
<tr>
<td>1997</td>
<td>20</td>
</tr>
<tr>
<td>1998</td>
<td>10</td>
</tr>
<tr>
<td>1999</td>
<td>10</td>
</tr>
<tr>
<td>2000</td>
<td>5</td>
</tr>
<tr>
<td>2001</td>
<td>5</td>
</tr>
<tr>
<td>2002</td>
<td>5</td>
</tr>
</tbody>
</table>
Software Thread Integration for Hardware to Software Migration

- Efficiency improved in two dimensions
  - Integrated threads more efficient, can use slower processor
  - Integration process automated, can improve time to market
- Simplifies hardware to software migration (HSM)
• CDG’s hierarchical structure simplifies integration
  - Vertical = conditional nesting, Horizontal = execution order
  - Summary information at each level

• Our Thrint back-end compiler operates on CDGs of host, guest threads
  - Annotates host with execution time predictions.
  - Moves guest code into host, enforcing ctl/data/time dependencies
    • Find gap, or else descend into subgraph
    • Have code transformations to handle conditionals & loops
Thrint

foo.s

Control-flow Analysis

Data-flow Analysis

Static Timing Analysis

Integration Analysis

Integration

foo.int.s

foo.id

XVCG

GnuPlot

Data-flow Analysis

Static Timing Analysis

Integration Analysis

foo.int.s

XVCG

GnuPlot

foo.id

foo.s

Thrint

foo id

foo.s

Control flow Analysis

Data flow Analysis

Static Timing Analysis

Integration Analysis

Integration

foo.int.s

foo.id

XVCG

GnuPlot

foo.id

foo.s

Thrint

foo id

foo.s

Control flow Analysis

Data flow Analysis

Static Timing Analysis

Integration Analysis

Integration

foo.int.s

foo.id

XVCG

GnuPlot

foo.id

foo.s

Thrint

foo id

foo.s

Control flow Analysis

Data flow Analysis

Static Timing Analysis

Integration Analysis

Integration

foo.int.s

foo.id

XVCG

GnuPlot

foo.id

foo.s

Thrint

foo id

foo.s

Control flow Analysis

Data flow Analysis

Static Timing Analysis

Integration Analysis

Integration

foo.int.s

foo.id

XVCG

GnuPlot

foo.id

foo.s

Thrint

foo id

foo.s

Control flow Analysis

Data flow Analysis

Static Timing Analysis

Integration Analysis

Integration

foo.int.s

foo.id

XVCG

GnuPlot

foo.id

foo.s

Thrint

foo id

foo.s

Control flow Analysis

Data flow Analysis

Static Timing Analysis

Integration Analysis

Integration
What About Interrupts And Other Short-Laxity Tasks?

- STI disables interrupts while integrated threads run
- What if some host threads (e.g. ISRs) can’t be delayed until the integrated thread finishes?
  - In STIGLitz example: interrupts are disabled for one field of video (16.167 ms)
- Solution
  - Use polling servers to service each non-deferrable thread (e.g. UART ISR)
- Polling Servers:
  - Created from the original ISR/routine by copying the body.
  - Insert guard code which checks whether the ISR/routine needs to execute.
What about Frequent Guests and Long Hosts?

- What if much of the idle time comes from short guest threads which run often?
- Amount of idle time in one instance of guest is not sufficient for the entire host thread to complete.
- Guest triggers before the previous integrated version finished.
- Solution
  - Break host thread into segments which fit into idle time and integrate guest in multiple times
  - Execute host one segment at a time

Previously STI couldn’t handle this region

WCET for Host Thread

Laxity for Host Thread (max. latency allowed)

Minimum guest thread period minus maximum guest thread work

Currently STI couldn’t handle this region
Resulting Expansion of STI Design Space

- Loosened the constraints on response time and idle time characteristics
- Now able to handle more demanding real-time applications
HSM Application Target: Video Signal Generation with STIGLitz

Goals

- Generate monochrome NTSC video refresh signal very cheaply
- Also provide high-speed (115 kbps) serial I/O
- Use software on a low-cost 20 MHz processor assisted by simple, cheap hardware
  - $3 20 MHz 8 bit Atmel AVR MCU (128kB ROM, 4kB RAM)
  - 64k x 8 SRAM
  - Two 4-bit shift registers
  - Two 4-bit clock dividers
  - One hex inverter
  - Four-resistor DAC
NTSC Signal: Idle Time Distribution

Idle-time between pixel rendering reclaimed to run integrated host work.

Idle-time between signal transitions is reclaimed to run dispatcher and do co-routine calls.

Idle-time yielded to run unintegrated host threads.
- Display refresh (pump out byte of video every 800 ns) has a hard real-time requirement
- Idle time between pixel-banging of video refresh accounts for 75% of CPU time during video data portion, 59% overall
- Reclaim that idle time by running integrated threads which refresh display and render graphics primitives simultaneously (time too short for context switch)
- **STIGLitz Graphics library**
  - APIs for rendering lines, circles, sprites, text, polygons, GIF decoder.
  - Fixed point math
- **NTSC video driver**
  - Implemented as Timer/Counter ISR.
  - ISR does one field of the NTSC frame.
  - 16 and 20 MHz versions.
  - 2 bpp 256x240 frame buffer
- **115.2 kbaud serial port**
  - one character per 87 us
Steps in STI: Source Code Preparation

• Structure program (C) to accumulate work to perform in integrated functions
• Write functions (C) to be integrated
• Compile to assembly code, partitioning register file for functions to be integrated (-ffixed)
Steps in STI: Analysis and Integration Planning

- Parse assembly code to form CFG and then CDG
- Perform tree-based static timing analysis
- Pad away timing variations from conditionals with nops or nop loops
- Perform basic data-flow analysis to identify loop-control variables and possibly iteration counts
- Compare duration of guest functions with maximum allowed latency for ISRs and other short-laxity host tasks
  - Create polling servers to handle these as needed
- Compare duration of host functions with amount of idle time time in guest functions, considering minimum period for guest
  - Break long host functions into segments which fit into guest functions’ idle time minus polling servers minus two context switch times.
- Define target times for regions in guest code which are time-critical
Steps in STI: Integration

- Note: conditionals have been padded away previously
- Single guest events
  - Move guest code to execute at proper times within host code
    - Replicate guest code into conditionals
    - Split and peel loops and insert guest code
    - Guard guest code within loop to trigger on given iteration
- Looping guest events
  - Peel off guest function loop iterations which don’t overlap with host loops
    - Integrate as single guest events
  - Fuse loop iterations which do overlap
    - Fuse loop control tests
      - Unroll loop to match idle time in guest loop with work in host loop
    - Create clean-up loops to perform remaining iterations
- Redo static timing analysis and verify correct timing
- Recreate assembly file
- Compile, link, download and run!
Threads and Integration

- **Threads**
  - UART Transmit polling server shifts data from tx queue to UART
  - UART Receive polling server transfers data from UART to rx queue
  - PumpPixel outputs a byte of packed video data every 16 cycles

- **Timing Analysis**
  - 63.5 us per scan line: 1270 cycles
  - Line counting and other overhead: -200 cycles
  - Dispatcher and context switching: -275 cycles
  - Time for UART polling servers: -132 cycles
  - Remaining time per line refreshed: ~650 cycles
    - Integrate 650 cycles of a host function per segment
New Utilization of Processor

MIPS Used

- Wasted capacity
- Integrated rendering
- UART polling servers
- Dispatcher w/context switching
- Display refresh & sync
- Foreground processing

No Integration
Int. - Horizontal Line
Int. - Vertical Line
Int. - Diagonal Line
Int. - X-Major Line
Performance Improvement and Code Expansion

- H-Line: 11.8
- V-Line: 13.5
- D-Line: 12.0
- X-Major-Line: 4.0

Normalized Performance (1/time)

- Discrete
- Integrated

Size (bytes)

- ROM
- RAM

Original

Integrated
Conclusions

• Enlarging the application space for STI
  – Low-laxity threads (and ISRs)
  – Frequent, short guest threads

• Demonstrated by integrating STIGLitz with video generation:
  – In use in NCSU’s Embedded System Design course.
  – For more information look at our upcoming December 2003 Circuit Cellar magazine.
  – Can download source code and PCB design from www.cesr.ncsu.edu/agdean/stiglitz

• Thank you
Appendices
Register File Partitioning vs. Performance

• Problem: STI requires that integrated threads share the register file

• Trade-off:
  • Code compiled to fit into fewer registers switches contexts faster
    - Dispatcher switches contexts roughly every 900 cycles
    - Two context switches for one register take 12 cycles
  • Code compiled to fit into fewer registers runs slower
    - More variables must remain in memory

• Goal: Squeeze pre-integrated threads into as few registers as practical

• Method: Determine sensitivity of the host threads’ execution time to the number of registers available
  - Divide AVR registers into three classes:
    • Pointer registers (r26-r31)
    • Immediate-operand capable registers (r16-r25)
    • Other registers (r0-r15)
  - Analyze DrawSprite, DrawLine, DrawCircle functions
  - Limit registers available to the register allocator through gcc’s –ffixed option.
  - Measure execution time using an on-chip timer/counter
Results

• Measurements
  - DrawLine and DrawCircle not very sensitive
  - DrawSprite very sensitive
  - Strange speed-up when excluding one pointer register

• Design decisions
  - DrawLine and DrawCircle
    • Exclude eight “other” registers and two pointer registers
    • Use 22 registers
    • Each context switch: 132 cycles
  - DrawSprite
    • Exclude only one “other” register and two pointer registers
    • Use 29 registers
    • Each context switch: 174 cycles
Code Size
Software Thread Integration Publications

- So, W. and Dean, A. “Procedure Cloning and Integration for Converting Parallelism from Coarse to Fine Grain,” Seventh Workshop on Interactions between Compilers and Architectures (INTERACT 7), February 1, 2003
- Dean, A. "Compiling for Concurrency: Planning and Performing Software Thread Integration," 23rd IEEE Real-Time Systems Symposium, Austin, TX, December 3-5, 2002

- More information and soft copies of above: http://www.cesr.ncsu.edu/agdean