Software Thread Integration and Synthesis for Real-Time Applications

Alex Dean
Center for Embedded Systems Research
Dept. of Electrical and Computer Engineering
North Carolina State University
Eliminate Context Switches Where They Limit Performance

- Dynamic task dispatching activities (e.g. context switches, ISRs) limit range of “illusion of concurrency” on a uniprocessor
- Instead, 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/dispatcher) and instruction scheduling (compiler)*
Common Problem

- Busy-idle profiles
- Software multithreading
- Interval scheduling
- Context switching or switches used in VULCAN, Esterel, event-driven simulations
- Quasi-static scheduling used in Picasso, Ptolemy
Software Thread Integration

- Efficiency improved in two ways
  - Integrated threads more efficient, so slower processor can be used
  - Integration process automated, so time to market can be improved
- Simplifies hardware to software migration
CDG’s hierarchy simplifies integration
- Vertical = conditional nesting, Horizontal = execution order
- Summary information at each level

Our *Thrinit* 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, transforming code
Thrint*: Thread-Integrating Compiler Back-End

foo.s
profiler
foo.id

Control-flow Analysis → Data-flow Analysis → Static Timing Analysis → Integration Analysis

Integration

XVCG
GnuPlot

* L. Niven
But what about…

- Caches?
- Deep instruction pipelines?
- Branch prediction?
- Superscalar instruction execution?
- Speculative execution?
- The reorder buffer?
- Page faults?
- Forwarding paths?
- Load queues?
- Data prefetching?
- Predicated execution?
- Branch delay slots?
- Instruction prefetching?
- Store forwarding?
- R-ops
- Dynamic optimization
- Et cetera, et cetera
How Much Processing is Needed?

Billions Sold in 2002

MCU/MPU Type

<table>
<thead>
<tr>
<th>Type</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>64</td>
</tr>
</tbody>
</table>

Not much!

Research Platforms/Activity/Funding/System Assumptions

Min. CPU Cycle Time

100 ns  40 ns  20 ns  33 ps

Cache not needed
Small Embedded Systems

• Processors
  • Not practical to design a custom processor
  • Not practical to use fast processor (e.g. raise clock speed by 10x)
  • Can handle some code explosion (e.g. up to 3x)
  • Using a generic microcontroller (e.g. 4, 8 or 16 bit) without memory protection, virtual memory, caches.

• Workload
  • At most a few threads need to make asynchronous progress, others can wait
  • One hard-real-time thread with tight deadlines
  • Other threads may have deadlines which are much longer
  • Interrupts are delayed or handled with polling servers (CASES 2003)
Application: Video Signal Generation with STIGLitz

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

- 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)
**STIGLitzz Software Architecture**

- **STIGLitz Graphics library**
  - APIs for rendering lines, circles, sprites, text, polygons, GIF decoder.
  - Accumulate work in queues
- **NTSC video driver**
  - Implemented as Timer/Counter ISR.
  - 16 and 20 MHz versions.
  - 2 bpp 256x240 frame buffer
- **115.2 kbaud serial port**
  - one character per 87 us
Processor Utilization Comparison

MIPS Used

- No Integration
- Int. - Horizontal Line
- Int. - Vertical Line
- Int. - Diagonal Line
- Int. - X-Major Line

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

Normalized Performance (1/time)

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

Discrete
Integrated
**ASTI: Extending STI for Independence**

- **An Issue with STI**
  - What if we can’t accumulate secondary thread work to execute later? Many applications can’t be restructured this way.

- **Synchronicity**
  - STI provides *synchronous* (lock-step) progress of integrated threads. *Entire* primary and entire secondary function execute.
  - ASTI provides *asynchronous* (independent) progress. Entire primary function and a *segment* of the secondary function executes at a time.

- **Nature of integration**
  - STI: *Entire primary* function is integrated into a copy of the secondary
  - ASTI: *Portion of primary* is integrated into every location in the secondary.
Control-Flow View of ASTI

Break secondary thread into segments lasting approximately for the total idle time

Integrate intervening primary code into each segment

Insert coroutine calls at start of idle time and end of each segment
Protocol Controller Options

- Analog & Digital I/O
  - System MCU
    - Discrete Protocol Controller
    - Physical Layer Transceiver
    - I/O Expander
  - MCU with on-board protocol controller
- Optional System MCU
  - Generic MCU with ASTI SW Protocol Controller

Communication Network:
- I/O Expander
- Discrete protocol controller with MCU
- Generic MCU with ASTI SW protocol controller
ASTI Applied to Comm. Protocol Controller

Executive
ReceiveMessage
ReceiveBit

Subroutine calls
Check for errors, save bit, update CRC

Prepate message buffer

Read bit from bus 3 times and vote

Sample bus for resynchronization

Idle time
Return

Secondary Thread
• Implement automotive protocol controller (J1850) in software (C and a little assembler)
• Compare how much progress the secondary thread makes for two implementations
  • ASTI approach uses coroutine calls
  • Interrupt-based approach assumes J1850 code implemented in state-machine ISR, triggered by timer
• Context switches swap all 32 registers - room for improvement
  • Cocall takes 153 cycles
  • Interrupt takes 151 cycles
• Primary thread: 129 lines of C, 1646 bytes compiled
• Secondary thread: 180 lines of C, 857 bytes compiled
Results: Performance

- Transmit: ASTI is 83% faster than ISR
- Receive: ISR can’t provide any secondary thread progress
Conclusions and Current Work

- **Conclusions**
  - STI and ASTI cut cost of context switching for generic microcontrollers
    - STI: most efficient, but only provides *synchronous* thread progress
    - ASTI: less efficient, but provides *asynchronous* thread progress
  - Can be applied to real-time systems, scheduling and compilation for uni- and multi-processors, and hardware/software cosynthesis

- **Current Work**
  - Investigating register file partitioning to speed context switch
  - Replace coroutine call with dispatcher to support multiple secondary threads
  - Deriving throughput estimates for real-time guarantees for secondary thread
  - Applying STI to VLIW, superscalar CPUs for throughput

www.cesr.ncsu.edu/agdean
Questions?

- Have you applied this to SPEC?
  - No, that’s not representative of embedded software-implemented communication protocols
- Don’t caches break the timing predictability you need?
  - The processors we use run at under 50 MHz, so we don’t have a memory wall
- Why not use a multithreaded processor?
  - They’re too expensive, too rare, and businesses prefer familiar processors
- Why not just design an ASIC to do it?
  - Too expensive to get the first one
- Why not program an FPGA to do it?
  - Too expensive to get the rest of them
ASTI Motivation

• How do we efficiently allocate $N$ concurrent (potentially real-time) tasks onto $M<N$ processors?
  • Compilation and scheduling for concurrent/parallel/distributed systems
  • Real-time systems
  • Hardware/software cosynthesis

• Bottlenecks
  • *Scheduling* each context switch
  • *Performing* each context switch

• We focus on $M=1$ processor, and that processor is generic (low-cost) with no special features for accelerating context switch bottlenecks

• Note: threads must be able to make independent (asynchronous) progress
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)
Thrint Overview

- Parse Asm
- Form CFG/CDG
- Read Integration Directives
- Static Timing Analysis
- Node Labelling
- Idle Time Analysis
- Temporal Determinism Analysis
- Data-Flow Analysis
- Register Virtualization
- Integration
- Register Reallocation
- Static Timing Analysis
- Timing Verification
- Code Regeneration

**STI**
- Pad Timing Jitter in secondary thread
- Plan integration
- Pad excess timing jitter
- For each guest
  - For each host
    - Do host loop transformations
    - Pad excess timing jitter
    - Clone and insert guest node(s)
- For each guest
  - For each host
    - If Fused loop, add fused loop control test code
- Delete original guests

**ASTI**
- Pad Timing Jitter in message level function
- Plan integration
- Pad jitter in predicate nodes and blocking I/O loops
- Integrate cocalls within the secondary thread.
- Integrate intervening guest code at appropriate locations.
- Delete original guests
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 (example)
• Perform basic data-flow analysis to identify loop-control variables and possibly iteration counts
• Compare duration of primary functions with maximum allowed latency for ISRs and other short-laxity tasks
  • Create polling servers to handle these as needed
• Compare duration of secondary functions with amount of idle time time in primary functions, considering minimum period for primary function
  • Break long secondary functions into segments which fit into primary functions’ idle time minus polling servers minus two context switch times
  • Also end segments when reaching a loop with an unknown iteration count
• Define target times for regions in primary code which are time-critical
Steps in STI: Integration

• Note: conditionals have been padded away previously
• Single primary events
  • Move primary code to execute at proper times within secondary code
    – Replicate primary code into conditionals
    – Split and peel loops and insert primary code
    – Guard primary code within loop to trigger on given iteration
• Looping primary events
  • Peel off primary function loop iterations which don’t overlap with secondary loops
    – Integrate as single primary events
  • Fuse loop iterations which do overlap
    – Fuse loop control tests
      • Unroll loop to match idle time in primary loop to work in secondary 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!
Results: Code Size

- **ASTI has code expansion of 95% vs. original code**