Providing Time- and Space-Efficient Procedure Calls for Asynchronous Software Thread Integration

Vasanth Asokan and 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
Overview

• The Big Picture

• Overview of ASTI

• Dealing efficiently with procedure calls

• Experiments
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

• Threads must be able to make independent (asynchronous) progress

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

• Our approach
  - $M = 1$ processor, and that processor is generic (low-cost) with no special features for accelerating context switch bottlenecks
  - $N = 2$ threads. $N > 2$ is an extension of $N=2$
  - Context switches are scheduled statically, cutting run-time overhead
  - Partition register file to accelerate performing each context switch
Basic Idea of ASTI

• Examine application to find a function $f$ with significant internal idle time

• **Long** - If an idle time piece $n$ is coarse-grain ($T_{Idle}(f,n) \gg 2^*T_{ContextSwitch}$), then we can recover it efficiently with a context switch

• **Short** - If it is fine grain ($T_{Idle}(f,n) !\gg 2^*T_{ContextSwitch}$), then apply ASTI
  - Remove intervening code (which fragments the idle time) and integrate it into any other thread which we might run
    • (pad conditionals to same time modulo $T_{Idle}$)
  - Break those other threads into segments which are statically scheduled and guaranteed to return precisely when needed
  - Use coroutine calls to switch between primary and secondary threads
Break secondary thread into segments lasting approximately for the total idle time.

Integrate intervening primary code into each segment.

Insert coroutine call back to primary at end of each segment.
ASTI vs. STI

• 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 execute 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.
Assumptions - Small Embedded Systems

• Processors
  - Not practical to design a custom processor
  - Not practical to use fast processor (e.g. raise clock speed by 100x)
  - 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 significantly longer
  - Interrupts are delayed or handled with polling servers (CASES 2003)
Application Domain: Protocol Controllers

- Need concurrency – fully loaded bus
- Idle time is very fine grain (under one bit time)
- Each application domain customizes its protocols
  - Wireless sensor networks tweak the medium access control, etc. for minimal energy use
  - Automotive: optimize for guaranteed (hard real-time) delivery
- Chicken and egg problem
  - Protocol controller chip won’t appear until adequate market anticipated
  - Chip costs remain high until volumes amortize design costs
  - Delay until protocol controller appears as peripheral on cheap MCUs
- MCUs are good fit for many embedded protocols, if concurrency is cheap
  - 10 to 200 cycles of processing needed per bit
  - 1 kbps – 1 Mbps bus speed
  - Temporally predictable MCUs are cheap and flexible
    - 1 MHz for $0.25
    - 100 MHz for $5-$10 (but you pay in increased energy use and other issues)
Protocol Controller Options

- Analog & Digital I/O
- System MCU
- Discrete Protocol Controller
- Physical Layer Transceiver
- I/O Expander

- Analog & Digital I/O
- System MCU
- On-Board Protocol Controller
- Physical Layer Transceiver

- Analog & Digital I/O
- Optional System MCU

- Analog & Digital I/O
- Generic MCU with ASTI S/W Protocol Controller
- Physical Layer Transceiver

Communication Network

- I/O Expander
- Discrete protocol controller with MCU
- MCU with on-board protocol controller
- Generic MCU with ASTI SW protocol controller
Motivation - Constraint on Procedures

- Procedure calls in the secondary thread are a problem.

- Conflicts arise in the placement of cocalls in the body of the procedure.
  - Extends to placing intervening primary code.

- Ad hoc solution – Pad call sites to a worst case offset and schedule cocalls in the procedure according to that offset.

- Problem - Wasted cycles, lower processor efficiency.

![Diagram illustrating the concept with arrows and sample PROC calls.](image-url)
Solution: Select Mix of Inlining and Cloning

• Procedure inlining
  – Replace calls with body of procedure.
  – Primarily trading off code size for execution speed.
  – Another benefit – whole-program optimization
  – Drawbacks – code size, register and stack pressure, compile time

• Procedure cloning
  – Form clones of a procedure optimized for a certain context.
  – Retarget original calls to appropriate clones based on context of the call.
  – Similar and an alternative to inlining.
  – Drawback – code expansion (might be lower than inlining).
Overall Approach to Solution

• Begin with call graph of secondary thread.

• **Phase one - Inlining**: From the set $S$ of procedures in the secondary thread, select a subset $I$ of procedures to be inlined, such that a maximal number of call sites are removed while not exceeding some code expansion limit $L$. Inline the set of procedures in $I$.

• **Phase two - Cloning**: Transform remaining $S-I$ of procedures to remove timing variability. Use procedure cloning to reduce the amount of padding cycles.
Phase One – Decide What to Inline

• Primary concern
  – Remove as many \emph{static} call sites as possible given a code size constraint.

• Similar to the knapsack problem.
  – However, new call sites may be created with the inlining of one call site.
  – The cost of each call site is also not constant.

• Full exploration of search space is too slow.

• Solution – Use branch and bound algorithm.
  – Simplified version of Leupers & Marwedel, ICCAD ’99
  – Use static, not dynamic information
Phase Two – Decide How Much to Clone

Now create a small set of procedure clones to handle temporally-clustered calls, reducing the padding needed.

Definition:
- Phase $\phi$ at any point in the code: Number of cycles left before which a cocall must be made back to the primary.
- Alternatively,
  $$\phi = (T_{SegmentIdle} - \text{number of cycles elapsed since last cocall})$$

For each procedure,
- Plot the distribution of $\phi$ at each call site to the procedure.
- Cluster call sites and dedicate a clone of the procedure, reducing the amount of padding required.
- Re-target calls to appropriate clones.
Clustering of Call Sites

- Phase two uses the Minimal Spanning Tree clustering algorithm.
  - Agglomerative approach.
  - Iteratively merge sets whose cost of linkage is minimum.
- Distance metric for two elements $\Phi_1$ and $\Phi_2$
  - Translates to padding cycles required when $\Phi_1$ and $\Phi_2$ are two calls that need to be serviced by a single clone.
- **Within Groups Linkage** metric for gauging similarity between two sets
  - Translates to padding cycles required when two sets are merged to form a single set.

\[
d_{\text{straight}} = 140 \text{ cycles (green)}
\]
\[
d_{\text{wrap}} = 80 \text{ cycles (blue)}
\]

A cocall is inserted at this point

Cocall duration = 220 cycles

distance = $\text{MIN}(d_{\text{straight}}, d_{\text{wrap}})$

= 80 cycles
Control Knob – Clone Ratio

- Controlling the number of clones created
  - Tradeoff between
    - number of clones formed – code size
    - reduction in the static padding cycles – processor efficiency

- Define for each procedure,
  - Clone Ratio: \( CR = \frac{N_c}{N_{cl}} \)
    \( N_c \) - Number of clones created,
    \( N_{cl} \) - Number of incoming calls to the procedure

- \( CR \) can range from \( \frac{1}{N_{cl}} \) to a maximum of 1.
  - No Clones: at \( CR = \frac{1}{N_{cl}} \), min code size, max padding cycles
  - All Clones: At \( CR = 1 \), max code size, min padding cycles
Experiments with Benchmark Suite of 2\textsuperscript{ary} Threads

- Atmel AVR
  - 8 bit RISC ISA, 32 registers
  - 8 MHz (\(\leq 20\) MHz)
- Code developed in C, compiled with avr-gcc 3.2
- Evaluated performance statically
- Code structured as two threads
  - Primary thread - Protocol controller
  - Secondary thread - Event-processing loop
- Each secondary thread gets inlined and cloned
  - Host Interface
    - Soft J1850 to UART bridge with queue interface
  - Smart Node (Serial I/O Expander)
    - I/O ports, ADC, PWM controller, fan control
  - PID Controller
    - Proportional/Integral/Derivative controller
    - Feedback signal is filtered (FIR) and linearized before use
Secondary Thread Call Graphs

Host Interface
- cnt_interface_fsm
  - flush_rx_buffer
  - flush_tx_buffer
  - receive
  - receive_buf_init
  - report_status
  - send_data_to_int
  - receive_data_from_int
  - transmit_data

Smart Node
- main_control_loop
  - check_for_events
  - dequeue
  - handle_message
  - send_acknowledgement
  - handle_fan
  - set_fan_speed
  - shutdown_fan
  - handle_error
  - enqueue
  - modulo_increment
  - handle_event
  - send_acknowledgement
  - change_power_mode
  - handle_adc
  - handle_power
  - handle_ioprt
  - write_data_register
  - get_bit_value
  - handle_pwm
  - read_data_register

PID Controller
- main_control_loop
  - UpdatePID
  - LinearizeFixed
  - fir_circular
  - fpmul
  - mult
  - asr
  - write_bit_value
  - get_port_and_bit
  - configure_bit
  - enable_pwm
  - disable_pwm
Phase 1 Results

- Inlining *reduces* code size for these benchmarks!
  - procedure prolog and epilog **large**
  - procedure body **small**
  - calls not nested deeply
- Our heuristic didn’t factor in reduction due to elimination of prolog/epilog
- Chose inlining based on 105% expansion limit
Phase 2 Results – Sensitivity to $T_{\text{SegmentIdle}}$

- Clustering limits padding cycles required, clipping the penalty for long $T_{\text{SegmentIdle}}$

![Graph showing static padding cycles needed before and after Phase 2](image-url)
Phase 2 Results – Raising Clone Ratio

- Payoff is largest for large values of $T_{SegmentIdle}$
- Clone ratios of 0.2, 0.3 cut most static padding cycles
- What about the memory expansion cost?

PID Controller Static Padding Cycles vs. Clone Ratio

- $T_{SegmentIdle} = 50$
- $T_{SegmentIdle} = 100$
- $T_{SegmentIdle} = 150$
- $T_{SegmentIdle} = 200$
- $T_{SegmentIdle} = 250$
- $T_{SegmentIdle} = 300$
- $T_{SegmentIdle} = 350$
- $T_{SegmentIdle} = 400$
Phase 2 Results – Exploring Costs and Benefits

- Approach works: first clones save the most padding cycles most densely
- CR = 0.3 eliminates 70% to 90% of static padding cycles
- PID, SmartNode require more clones than Host Interface
Secondary Thread Speed-Up from Phase 2

- How much does Phase 2 (clustering and cloning) speed up the secondary thread?
- Based on dynamic performance of secondary thread
- Host Interface and Smart Node have many calls in loops
- PID Controller has only two calls in loops, is instead dominated by math
Conclusions

• Need efficient support for procedures in secondary threads
  – ASTI techniques lose efficiency as $T_{\text{SegmentIdle}}$ grows
  – Easier to develop, debug and maintain modular code.
• Approach provides an algorithmic basis for trading off code size and processor efficiency.
• Application of inlining and cloning with real-time twist
• Greater program run-time efficiency
  – Can avoid raising clock speeds, staying in cheap-and-easy-hardware land
    • not past the memory wall (avoid unpredictable memory access times)
    • minimal pipelining (avoid long branch penalties)
    • low operating frequencies (avoid high power consumption)
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
Backup Slides
Protocol Software Structure

Most idle time is located in these functions
Obtaining Timing Information in Phase Two

- Timing information ($\Phi$) of call incoming calls, required completely, before a procedure can be processed.
- Can proceed through call graph in topological order.
- However, timing information is not easy to obtain.
  - Nested calls to topologically lower procedures.
  - Interleaving of calls in the same level.
- For e.g. while processing $P_1$, call site $b$ may be present in between $a$ and $c$. But call site $b$ has not been processed yet – therefore $\Phi$ of $c$ cannot be determined.
- Solution – Pad all procedures to last a multiple of a cocall duration. This way, the duration of all procedures is known ahead of time.
- Problem – Contrary to purposes.
Why Not Use Dynamic Call Graph Information?

• Gathering profile information is difficult with these applications
  – Limited resources
    • 4 kB SRAM
    • No file system
  – Applications tightly coupled to I/O, complicating simulation
    • Stimulus and log files don’t scale well, becoming cumbersome
• This is the first step into solving the problem
  – Performance penalty of static vs. dynamic information is expected to be small (e.g. 24%) compared with impact of ASTI
  – Future work
Thrint Overview

**Parse Asm**

- Form CFG/CDG
- Read Integration Directives
- Static Timing Analysis
- Node Labelling
- Idle Time Analysis
- Temporal Determinism Analysis
- Data-Flow Analysis
- Integration

**STI**

- Pad Timing Jitter in secondary thread
- Plan integration
- Pad excess timing jitter
- For each guest:
  - Do host loop transformations
  - Pad excess timing jitter
  - Clone and insert guest node(s)
- 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 in secondary thread
- 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

---

Pad Timing Jitter in message level function.

Plan integration in secondary thread.

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.
Microcontroller configurations and High Volume Prices

<table>
<thead>
<tr>
<th>Chip</th>
<th>Flash Memory Size in kilobytes</th>
<th>Clock Speed in MHz</th>
<th>Cost per unit at 50k Units (in dollars)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATMega8L</td>
<td>8</td>
<td>0-8</td>
<td>2.35</td>
</tr>
<tr>
<td>ATMega161</td>
<td>16</td>
<td>0-8</td>
<td>5.10</td>
</tr>
<tr>
<td>ATMega323</td>
<td>32</td>
<td>0-8</td>
<td>7</td>
</tr>
</tbody>
</table>
Timing Conflict Example

Body of Secondary Thread

10 cycles

- Cocalls

30 cycles

- Cocall placement within Sample’s body

\( T_{SegmentIdle} \) is 50 cycles

- Conflict
Problems Obtaining Timing Information

\[ P_1(); \rightarrow \text{Call site } a \]
\[ . \]
\[ . \]
\[ P_2(); \rightarrow \text{Call site } b \]
\[ . \]
\[ . \]
\[ P_1(); \rightarrow \text{Call site } c \]
### Breakdown of Automatic and Manual Tasks

<table>
<thead>
<tr>
<th>Automatic</th>
<th>Manual</th>
<th>Description of Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>Profile code expansion and static call site reduction.</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Form the call graph of secondary thread.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Topologically sort the call graph.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Phase 1</strong></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Apply Min_IV algorithm for various code expansion limits.</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Choose an inlining plan.</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Inline procedures chosen in plan using gcc.</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Profile code expansion and static call site reduction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Phase 2</strong></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Pad jitter in each procedure’s CDG.</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Pad blocking I/O loops to duration of a cocall.</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Calculating durations of each procedure in reverse topological Order.</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Obtain timing information on $\Delta c$ distribution for each procedure.</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Apply minimal spanning tree clustering algorithm for each procedure and get cloning plans.</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Choose desired cloning plan.</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Profile padding cycles inserted and code expansion due to Clones.</td>
</tr>
</tbody>
</table>
References

