Asynchronous Software Thread Integration for Efficient Software Implementations of Embedded Communication Protocol Controllers

Nagendra J. Kumar, Siddhartha Shivshankar and Alexander G. Dean
Center for Embedded Systems Research
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695-7256
Alex_Dean@ncsu.edu

Abstract
Existing software thread integration (STI) methods provide synchronous thread progress within integrated functions. For the remaining, non-integrated portions of the secondary (or host) threads to run and avoid starvation, the primary (or guest) thread must have adequate amounts of coarse-grain idle time (longer than two context-switches). Fine-grain idle time is too short to be used for other threads, as it does not allow the context switches needed to switch between the primary and secondary threads.

Software-implemented protocol controllers are crippled by this problem; the primary thread “bit-bangs” each bit of a message onto or off of the bus, leaving only sub-bit duration fragments of idle time. This fragmented time is typically too short to recover through context switching, so only the primary thread can execute during message transmission or reception. Heavy network utilization can starve the secondary threads, leading to poor or unacceptable system performance.

We have developed a new set of methods called asynchronous software thread integration (ASTI) which address this problem of starvation through the efficient use of coroutine calls and integration. ASTI enables the implementation of embedded communication protocols on low-cost, moderate speed (1 – 50 MHz) 8- and 16-bit microcontrollers. Coroutines are used to enable asynchronous thread progress. Integration is used to reduce the number of context switches, improving performance. Instead of using a coroutine call (cocall) for each context switch, certain portions of the primary thread are removed and integrated into the secondary thread in multiple locations. This code removal converts the primary thread’s idle-time from fine-grain to coarse-grain, allowing secondary threads to be invoked with far fewer cocalls, and hence much more efficiently.

We demonstrate ASTI by applying it to a standard automotive communication protocol controller (J1850) for hardware to software migration (HSM). We show significantly improved secondary thread performance when compared with a traditional interrupt-based software approach.

1. Introduction
Specialized network protocols abound for embedded systems: in land vehicles (CAN, LIN, J1850, TCN), aircraft (1553, DATAC, ARINC-429), process control and factory automation (Profibus, Foundation Fieldbus, SERCOS), building automation (BACNet, LON, DALI) and sensor networks. These applications are best served by
buses with speeds ranging from under 10 kbps to over 1 Mbps, rather than with high-speed networks (e.g. Ethernet, Myrinet, ATM). These embedded network applications can benefit from protocol optimizations for which hardware versions are either unavailable or otherwise unsuitable due to cost (e.g. a $500,000 mask charge for a new chip), time to market (e.g. 1 to 2 years per ASIC) or functionality. For example, SmartDust [28] requires a multi-hop wireless network communication protocol which minimizes power and energy consumption. Tuning the protocol in software is much easier than in hardware. Harsh environments may also preclude or complicate the use of existing ICs. An automotive application required a CAN interface to operate at temperatures of 185°C leading to a software implementation of the protocol on a high-temperature microcontroller [22]. Radiation is another contributor to harsh environments, leading to the need for radiation-hardened or shielded ICs. Obsolescence may also lead to a software implementation.

There are many examples of software implementations of communication protocols which reach down to the bit level. For example, there are infrared protocols for remote controls, software UARTs, modems and protocols such as I2C and RDS, a radio-based data broadcast service [1][2][3][4][5][6][7][8][9][10][8].

Implementations of non-trivial protocols have multiple real-time events within each bit time, as shown in Figure 1a. Typical activities include sensing the bus during arbitration, sampling multiple times per bit for reliability through voting and detecting bus transitions for clock recovery (resynchronization). These events fragment the available idle time. Such a software implementation requires a mechanism for sharing the processor among the primary thread (implementing the protocol) and the secondary thread (implementing the interface to the host device,
or other functionality, such as a smart node or I/O expander). This mechanism must transfer control quickly and at the correct time; inefficiencies in either aspect will incur a significant overhead.

The fundamental problem facing software implementations is the run-time overhead of both scheduling and performing context switches to share the processor. Architectural modifications can improve concurrency. Fast context switches improve the emulation of concurrency by a uniprocessor. Some processors provide multiple or shadowed register files, allowing faster context switches and cutting this overhead dramatically. (STI can leverage these fast context switches to improve performance by eliminating dynamic scheduling overhead.)

Multiprocessors offer true concurrency. Network processors [34][35][36][37][39][40] target high-performance communication applications and include hardware support for concurrency (through multiple processors and/or multithreading), as well as cryptography and network protocols (e.g. Ethernet or ATM).

There are two potential disadvantages for using a specialized network processor or multiprocessor. First, the processor cost is typically much higher than for a generic processor due to the need to amortize design costs over lower volumes [35][38]. (For comparison, we target low-end 8- and 16-bit microcontrollers (MCUs) with prices of $0.50 (US) to $5.00 (US)). Second, businesses favor the use of known products with proven performance. Time-to-market pressures may rule out using a new, untested processor with similarly untested development tools. These forces make a software implementation on a low-cost generic microprocessor a useful alternative.

Regardless of how quickly the processor can switch contexts, something must trigger the switches. In Figure 1, the ReceiveBit function must sample the bus three times for voting and also take a late sample for clock recovery (resynchronization to the incoming data stream). For a flexible solution, these events must be able to occur at arbitrary times defined at design time. Static scheduling satisfies this requirement but is tedious if not automated. Dynamic switching (interrupt-based) must be triggered by something. The obvious solution is a programmable timer, but the overhead of reloading the timer for each new context switch is significant. Using multiple timers is unattractive due to the scarcity of (e.g. 4 timers) and demand for timers on our target MCUs. Furthermore, any interrupt response latency (e.g. 5 – 20 cycles) is added to the overhead of this approach.

The tempting solution of raising the processor’s clock frequency to reduce the temporal impact of this overhead leads to a host of new problems: power consumption rises with frequency, electromagnetic interference and noise grow, design and prototyping become more complicated, and greater demands are placed on the power supply and distribution system [39]. Designers of embedded systems limit processor and bus clock frequencies and signal slew rates to minimize these problems [27].

In our previous work we have developed software thread integration (STI), which creates a single implicitly multithreaded function from multiple functions[22][23][24][25][26]. STI is useful because it enables the scheduling of multiple processes on a uniprocessor while eliminating most of the context switching overhead. This overhead becomes prohibitively large when implementing threads with fine-grain concurrency (on the order of cycles or tens of cycles). To cross the boundary and schedule one thread’s instructions within another we create integrated threads using compiler techniques such as code motion, control-flow transformations, register reallocation, static timing analysis and time-based instruction scheduling.
Existing STI work relies upon the primary thread having large amounts of \textit{coarse grain} idle time (defined as being longer than two context-switches) in which \textit{any} part of the secondary thread can run. Existing STI works by selecting suitable functions of the secondary thread which can be deferred and run \textit{synchronously} with suitable functions of the primary thread. The remainder of the secondary thread is run within the primary thread’s coarse grain idle time. The secondary thread must accumulate deferrable work (e.g. enqueue it) for later processing by the integrated threads.

Bit- and byte-banging software implementations of communication protocols do not fit this model. (Byte-banging involves using a generic shift register as a serializer/deserializer, reducing the processing load). The primary thread has little or no coarse grain idle time when transmitting or receiving a message. During periods of 100% utilization, the communication bus will force all nodes on the bus to run their primary threads. Integrated threads will execute if data is available, but eventually will run out of work, and the secondary thread will not proceed until the bus utilization falls below 100% (leading to coarse-grain idle time in the primary thread). It is not practical to integrate all of the secondary thread with the primary thread using this model for two reasons. First, the primary functions with idle time would need to be integrated with every function in the secondary thread. Second, the idle time within the primary functions is very short, so the secondary thread’s work would not fit without extensive modification of either primary or secondary code. Both of these issues raise the complexity of integration to an unmanageable level.

This paper presents our solution to this design challenge. We use context switches (coroutine calls, in particular) to allow asynchronous progress for the primary and secondary threads. We mitigate the cost of context switching (which is large relative to the limited coarse grain idle time) by reducing the number of these switches through software thread integration. Portions of the primary thread’s code are integrated and duplicated throughout the secondary thread, ensuring that a context switch to the secondary thread will still result in the proper primary code executing at the correct times. In addition, coroutine calls are inserted into the secondary thread to ensure that it yields control back to the primary thread before the idle time expires.

These methods eliminate most of the context switches needed to share the processor, enabling recovery of finer-grain idle time for use by the secondary thread. The two benefits of these recovered compute cycles are improved performance of the secondary thread and reduced minimum clock speed for the microprocessor. The former allows more work to be done, while the latter enables the use of a lower cost processor or a more power-efficient operating frequency. These benefits enable embedded system designers to use existing processors more efficiently with less software development effort. With these techniques, designers can implement their communication software methodically without resorting to ad hoc solutions which are brittle and difficult to change.

This paper is organized as follows. Section 2 presents our design space target. Section 3 introduces our new ASTI methods. Section 4 demonstrates their use on J1850, a standard automotive embedded network protocol. Section 5 analyzes the resulting system’s performance, timing accuracy and code size. Section 6 draws conclusions and points toward future work.
2. Design Space Target

2.1. Hardware Protocol Controllers

Hardware protocol controllers are commonly available in three forms. Figure 2a shows the simple “I/O expander” (or “Serially Linked I/O”) device, which connects discrete analog and digital inputs and outputs to a network[13][14][15][16]. This device can act upon messages commanding certain output values, and can reply to requests for the state of the inputs. It can also automatically generate a message when specific input conditions occur. Some may even provide local control loop closure. Figure 2b shows a discrete protocol controller, which acts as a bridge between the communication bus and the system’s microcontroller, using either a serial (e.g. UART) or parallel link. Figure 2c shows a microcontroller with an on-board protocol controller; this is the most powerful option.

![Diagram of hardware protocol controllers]

Figure 2. Three types of hardware protocol controllers, and their replacement with a generic MCU.

Figure 2d shows the architecture of the system based on ASTI. The ASTI techniques are most suitable for replacing the I/O expander and stand-alone protocol controller. This is because integration complexity rises with application code size and the secondary threads have reduced real-time performance. However, a simple application could use ASTI to replace a microcontroller containing an integrated protocol controller with a generic (less expensive) microcontroller.

2.2. Microcontrollers as Protocol Controllers

The processing needed per bit for common embedded network protocols is typically quite small, ranging from 10 to 200 instruction cycles [17][18][19][20][21][22]. Given the means to achieve adequate concurrency, a simple
and inexpensive microcontroller running at a clock frequency of 100 to 1000 times the network bit rate could implement these protocols and perform other processing as well. 8 and 16 bit microcontrollers with clock rates of 1 MHz to 50 MHz are quite inexpensive, with prices ranging from $0.50 (US) to $5.00 (US). These are low cost in comparison with discrete network interface ICs or network processors, leading to a significant potential cost savings.

Although the timing variability of modern high-performance CPUs and memory hierarchies greatly reduces the temporal determinism which STI and ASTI require, this is a non-issue. STI and ASTI target applications which neither need nor can afford these CPUs and memory systems. For perspective, in 2001 75% of the 8 billion microprocessors sold were four- and eight-bit units [31]. These microcontrollers run applications which are not computationally intensive, and do not need more parallelism or faster clock rates. They lack sophisticated microarchitectures and memory systems, and often cannot afford them. Instead these applications are constrained by other issues such as functionality, cost, power dissipation, design time and use of commercial off-the-shelf products. Hardware to software migration with STI and ASTI allows the designer to address these issues efficiently.

We target applications with only one hard real-time thread (the primary thread, used for the communication protocol). During message transmission or reception, any other (secondary) threads will only run in the available idle time, slowing them by a factor of $T_{idle \, per \, bit}/T_{bit}$. (Note that this is an improvement over the non-integrated case, where threads may be completely blocked). We leave schedulability analysis for these threads for future work.

2.3. System Architectures

![Diagram of Protocol Executive](image)

Figure 3. Three layers of functions implement the primary (protocol controller) thread

The baseline software implementation presented here uses a single thread to perform the communication work, using functions at three levels as shown in Figure 3. The top level executive function implements a finite state machine to monitor an idle bus, send a message or receive one. The middle level consists of send and receive message functions, which deal with message fields (e.g. sending an identifier, format header, data and CRC). The lowest level deals with sending and receiving individual bits, and also needs to sample the bus multiple times per bit for voting, arbitration and resynchronization. Other bit-level activities for other protocols might include bit-(de)stuffing.
Figure 4. ASTI code on a generic microcontroller provides J1850 protocol functionality to system microcontroller.

We use the architecture of Figure 2d (including the optional system MCU) to replace that of Figure 2b. Figure 4 presents the internal software overview. The protocol thread runs on a generic microcontroller to replace the protocol controller IC. This thread is integrated with another “host interface” thread which communicates with the system MCU through a UART, SPI bus, or parallel I/O. Our implementation uses this approach and UART communications.

3. Using ASTI and Coroutines for Bit-Banged Communications

ASTI enhances traditional methods for implementing communication protocols in software by reducing the number of context switches needed (illustrated in Figure 1c). The idle time within the primary thread is fragmented by real-time instructions; each transition between the primary and secondary threads requires a context switch, consuming valuable processor time. ASTI integrates these groups of primary instructions into multiple locations in the secondary thread, reducing the number of context switches needed.

3.1. Software Thread Integration Background

Software Thread Integration is a compiler technique that merges two functions into one implicitly multithreaded function [22][23][24][25][26]. When used for real-time software, it enables the placement of time-critical instructions from one thread so they execute at a given time relative to the beginning of the integrated function, regardless of the control or data flow characteristics of either thread. STI begins with timing regularization; execution paths of uneven duration are padded to last the same amount of time. Next, either the register file is partitioned or def-use webs are extracted. Code from one thread can now be moved to execute at given time in the other, using code transformations such as motion, replication, loop peeling / splitting / guarding / unrolling / fusion. The transformations are driven by timing directives which indicate when specific real-time instructions must execute. We have implemented a thread-integrating compiler Thrint which implements many of these analyses and transformations for the AVR and Alpha ISAs.

STI aids the efficient implementation of real-time applications in software. Currently, the approaches used for hardware-software migration cannot recover idle time chunks that are not large enough to do context switch twice. STI makes it possible to recover even these idle time slots.
3.2. Asynchronous Software Thread Integration

ASTI code transformations remove primary code (intervening primary code) which fragments the idle time, creating a longer time in which the secondary thread can run. The secondary thread is modified so it always executes a cocall back to the primary thread just before the end of this newly enlarged idle time. The secondary thread is also modified by integrating in the intervening primary code so it executes at the correct times. To do these two modifications, the secondary thread is divided into segments lasting the available idle time minus two cocall times. A copy of the intervening primary code and a cocall are integrated into each segment.

Padding of the primary thread functions with nops regularizes timing. The primary thread’s bit-level functions are modified so that each takes a constant amount of time. Each message-level function is padded twice, first to eliminate jitter and then so that calls to bit-level functions occurs at a fixed frequency, eliminating the inter-bit processing time variations.

The secondary thread is padded to remove jitter. Blocking I/O loops are identified for later padding and cocall insertion. Current methods preclude subroutine calls in the secondary thread; all must be in-lined. However, we are developing automated methods to support them using padding, cloning and inlining. These methods allow us to trade off code size for execution speed.

We have also developed other supporting methods which are not presented here due to space constraints, but appear elsewhere [30]. These methods include the following. Blocking I/O loops are common in embedded software (e.g. busy waiting on incoming UART data); such loops in the secondary thread are padded and converted to non-blocking loops. Coroutine calls and intervening primary thread code are guarded with conditionals to enable the use of a single rather than multiple copies of a host interface functions as presented here, reducing code explosion. Mechanisms for switching between multiple copies of integrated host interface functions (e.g. when switching to transmitting a message after receiving one) are introduced. Methods for receiver clock resynchronization based upon detecting bus transitions during message reception are introduced to allow greater tolerance when integrating.

3.2.1. Step 1 – Primary Thread Padding and Analysis of Idle and Segment Times

The first step in planning integration is to find how much idle time is available for use by the secondary thread. This defines the duration of each segment between coroutine calls in the secondary thread. Idle time is allocated to the bit-level rather than message-level functions to minimize fragmentation of idle time and to reduce the number of cases to be considered in integration. One result of this choice is the bit-level functions will complete (execute a return) as late as possible, constrained by the worst-case amount of processing needed in the message-level function before the bit-level function can be called again.
We use Thrint, our back-end compiler, to eliminate all timing jitter in the message and bit-level functions by padding all execution paths with nops to last the same amount of time. We then create a timeline of the processor activity (e.g. Figure 5a) for a given bit-rate and clock frequency for both the send_bit and receive_bit functions, based upon static timing analysis performed by Thrint. An instruction which samples or controls the bus imposes explicit timing requirements based on the protocol. These instructions serve as timing anchors; other instructions have the implicit timing requirement of executing in order. If there are conditional control-flow paths in the function, the worst-case (slowest) processor activity is used to determine the processor activity. As in our previous work, we use integration directives in a control file (Figure 5b) to specify the desired start time (as a delay from procedure entry) and tolerable error for each basic block holding a real-time instruction. Thrint reads these directives and uses them to guide integration.

**TCS** is the time for a context switch, implemented with a coroutine call. The segment time **TSegment** lasts from the beginning of the first idle time bubble large enough for a context switch to the end of the last such bubble. **TSegmentIdle** is the amount of idle time within the segment which can be used for a different thread, and is the sum of the individual idle time bubbles minus the overhead of performing two coroutine calls. In Step 3, the secondary thread will be sliced into segments of duration **TSegmentIdle** and each segment will be perforated with primary thread code and a coroutine call.

### 3.2.2. Step 2 - Transform Bit-Level Functions

As shown in Figure 5, the bit level functions are modified by first removing the code during **TSegment** and saving it for later replication into the secondary thread. Second, a coroutine call is inserted at the beginning of the idle time call the secondary thread.

### 3.2.3. Step 3 - Integrate Bit-Level Primary Code into Secondary Thread

The secondary thread is next modified to execute the intervening code at the correct times and also yield control back to the primary thread at **TSegment**. A separate version of the secondary thread is needed for each primary bit-level function (e.g. send_bit and receive_bit). For each case the procedure of transformation is the same, but differences will result from varying segment time durations and amounts of primary thread work. The secondary
thread contains an infinite loop which awaits commands from the system MCU (via the serial or parallel port) and acts upon them (e.g. enqueuing a message for transmission on the network). The body of this loop integrated with the primary thread.

Timing variations are padded away with nops or nop loops. Blocking I/O loops are padded to last exactly TSegmentIdle per iteration. Coroutine calls are inserted every TSegmentIdle. The primary thread code extracted from the bit-level function is now integrated into each segment of the secondary thread using standard software thread integration techniques and the algorithms in Figure 6. The result of this process is illustrated in Figure 1c.

```c
Copy_and_Integrate_Subgraph(target_offset, master_code)
/* Method called on a node to integrate in the master_code subgraph at target_offset cycles from the beginning of this node */
start_t = end_t = 0
cur_node = this node
while !Done && cur_node
    end_t += worst case duration of cur_node
    if target_offset > end_t // go to next node
        start_t += worst case duration of cur_node
        cur_node = cur_node->successor
    else
        if target_offset == start_t
            insert copy of master_code before cur_node
        else if target_offset == end_t
            insert copy of master_code after cur_node
        else // need to transform cur_node and integrate into it
            switch type of cur_node
            CODE: split cur_node at target_offset - start_t
                  insert copy of master_code after cur_node
            PREDICATE: call this method recursively on 1st true and 1st false child
            LOOP: split loop at target_offset – start_t
                  peel middle iteration of loop
                  call this method recursively on first node of peeled iteration
    Done = true;

Integrate_Asynchronous_Threads( )
/* Method called to integrate multiple copies of primary code and cocall into secondary thread */
interv_primary = extract intervening code from primary function
cocall_21 = load cocall code (from secondary to primary) from file
secondary_start = find first child node of infinite loop in secondary function
offset_time = start time of secondary_start
while (offset_time < end of infinite loop's first iteration)
    secondary_start->Copy_and_Integrate_Subgraph(offset_time, interv_primary)
    secondary_start->Copy_and_Integrate_Subgraph(offset_time, cocall_21)
    offset_time += TSegmentIdle
```

Figure 6. Copy_and_Integrate_Subgraph method integrates a subgraph based on a time offset from a given node. Integrate_Asynchronous_Threads integrates intervening primary code and cocalls into the secondary thread’s infinite loop body every TSegmentIdle.

3.2.4. Step 4 - Pad Message-Level Functions

The message-level functions must be modified to ensure that the calls to the bit-level functions occur at multiples of T_{bit}, taking into account the new duration of the bit-level functions. This is straightforward and uses padding to equalize the times to the worst case between the calls. The start of the receive message function must be modified with padding to ensure the first receive_bit call begins sampling the message at the correct time.

4. Experiments

We have written C code to implement a J1850 controller with transmit, receive and in-frame response support. It communicates with a host microprocessor through the UART. We have also written C code for a host interface thread which communicates through the UART. We have integrated send_bit and receive_bit primary thread
functions with the host interface function. We have tested these functions through simulation to ensure their correctness and timing accuracy.

4.1. J1850 Protocol

J1850 is a standard automotive communication network [32]. There are two versions: 41.6 Kb/s Pulse Width Modulation (PWM) and 10.4 Kb/s Variable Pulse Width (VPW). The latter is more prevalent and is used here. VPW communicates via time dependent symbols. Each symbol has a single transition, can be either at the ACTIVE or PASSIVE level and has one of the two lengths, 64us or 128us ($t_{NOM}$ at 10.4 Kb/s baud rate), depending on the encoding of the previous bit.

The J1850 protocol uses carrier-sense, multiple-access with lossless collision resolution (CSMA/CR) arbitration. When a transmitting node sends a passive symbol but sees an active symbol, it determines that a higher priority message is being transmitted by another node and hence it has lost arbitration for the bus so it must stop transmitting and continue to function as a receiver. The node(s) that won arbitration continue to transmit, checking each bit of the current message and dropping out when necessary until just one node is left. This checking is done for every bit.

![Figure 7. J1850 frame format](image)

Messages transmitted on the J1850 bus are structured according to the format shown in Figure 7. Each message has a maximum length of 12 VPW (Variable Pulse Width) bytes excluding SOF, EOD, NB and EOF with each byte transmitted most significant byte (MSB) first. A message’s header and data fields can total up to eleven bytes. The CRC byte is used by the receiver(s) of each message to determine if any errors have occurred during communication. During transmission, the CRC is calculated and the CRC byte is appended to the message transmitted onto the J1850 bus. Each receiver performs CRC detection on each message it receives from the bus and verifies proper message reception.

4.2. Experimental Method

4.2.1. Tools

The protocol’s C source code is compiled to assembly code with AVR-GCC v3.0 [1] at –O1 optimization (at higher levels of optimization GCC creates unstructured code, which our tools do not yet support). Our research compiler Thrint performs static timing analysis of assembly code, padding, and most of the integration. As shown in Table 1, most of the code transformations needed for integration are performed automatically by Thrint. Code is simulated using AVR Studio v3.52 [12].
Table 1: Integration Task Breakdown

<table>
<thead>
<tr>
<th>Automatic</th>
<th>Manual</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>parse assembly code and form CDGs</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>perform static timing analysis on all functions</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>pad timing jitter in primary thread functions</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>analyze and extract intervening primary code from bit-level functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>analyze secondary thread and pad timing jitter</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>all code but blocking I/O loops</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>blocking I/O loops</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>integrate cocalls and intervening primary bit-level code (except blocking I/O loops)</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>all code but blocking I/O loops</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>blocking I/O loops</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>generate final integrated assembly file</td>
</tr>
</tbody>
</table>

4.2.2. Protocol Implementation

Table 2: J1850 Source Code Sizes

<table>
<thead>
<tr>
<th>Thread</th>
<th>Function</th>
<th>C Source Size</th>
<th>Compiled (.text) size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>send_msg</td>
<td>33 lines</td>
<td>458 bytes</td>
</tr>
<tr>
<td></td>
<td>rec_msg</td>
<td>40 lines</td>
<td>96 bytes</td>
</tr>
<tr>
<td></td>
<td>send_bit</td>
<td>43 lines</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>rec_bit</td>
<td>13 lines</td>
<td>76</td>
</tr>
<tr>
<td>Secondary</td>
<td>host_int</td>
<td>180 lines</td>
<td>857</td>
</tr>
</tbody>
</table>

Bit-, message- and control-level functions were written in C for J1850 and are described in Table 2. This code implements protocol functionality but has not been scheduled for proper timing (e.g. with padding or timers). We target an 8 MHz 8-bit AVR microcontroller. The bit-level functions of the send and receive routines of J1850 access the bus multiple times while sending or receiving a bit. While transmitting a bit, the send bit function accesses the bus twice, once to transmit the bit and then to check for arbitration. The receive bit function samples the bus every 64µs. In order to sample the value on the bus correctly, the receive bit function samples the bus thrice and then votes using the samples received. This multiple sampling fragments the idle time. The timelines for the send and receive bit functions of J1850 are shown in Figure 8.
The coroutine calls are responsible for saving the context of one thread before switching to the other during the idle time slots. For simplicity, registers for an inactive thread are stored on that thread’s stack. During a coroutine call, all 32 data registers and the status register of the calling thread are pushed on its stack and those of the other thread are popped from the corresponding stack, so the coroutine call lasts 152 cycles. For the send routine, since the first idle time slot is less than 152 cycles, it cannot be utilized. For the receive routine, the first idle time slot is 158 cycles and the last idle slot is for 239 cycles. Hence by removing the intervening primary code, the entire idle time duration within the receive bit function can be used. Performance could be improved by partitioning some or all of the register file, reducing context switching time by four cycles per register dedicated to a thread. This would also allow finer-grain idle time to be recovered.

In order to account for clock differences between the sender and receiver, resynchronization is required during reception. Resynchronization is performed dynamically by either extending or shortening the padding depending upon input signal transition times.

5. Analysis of Results

5.1. Performance Improvement

ASTI improves secondary thread performance compared to traditional approaches because it is better able to use primary thread idle time. We compare performance to an interrupt-based approach in which a context switch takes 151 cycles, which is nearly the same as the 153 cycles of the context switch of a coroutine call. (These times could be improved by partitioning the register file or using a processor with multiple register banks). The integrated code does not need to perform as many context switches and allows the use of shorter idle time slots, enabling the secondary thread to make progress even during periods of 100% bus utilization.

We compare the number of CPU cycles available to the secondary thread as a function of message size in Figure 9 and Figure 10. We present three cases, as message duration depends on the actual data sent due to variable pulse-width encoding. In the first case, all bits last 64 us (worst case, shortest message), the second is an average, and the third case has all bits lasting 128 us.

The progress through the secondary thread is much faster using the new STI methods compared to the interrupt-based approach. The integrated code enables the processor to spend approximately twice as much time on the secondary thread while transmitting. The case of reception is even more striking. Because the largest idle time
fragment in receive bit lasts only 239 cycles, there is not enough time to switch to and return from a secondary thread, so no secondary thread progress can be made using an interrupt-based approach. The integrated code is able to dedicate time to the secondary thread, guaranteeing progress even during 100% message reception.

![Secondary Thread Progress During Transmission](chart1.png)

Figure 9. Secondary thread progress during message transmission

![Secondary Thread Progress During Reception](chart2.png)

Figure 10. Secondary thread progress during message reception

5.2. Verification Techniques

A: What bits are put on the bus and when?  
B, C: When is the bus sampled?

![Overview of verification approach](diagram.png)

Figure 11. Overview of verification approach
To ensure that the code transformations are correct, the integrated code is simulated with AVR Studio [12] and log files (which record output port signals) are generated. The microcontroller is run at 8 MHz. These log files are then compared, transition by transition, with known good (reference or “gold”) log files created from simulations using the original (non-integrated) code. The timing of this code has been configured to match the J1850 specification exactly and represents what a dedicated hardware J1850 controller would send or receive. Verification is performed as shown in Figure 11. Comparison A determines if the times and values of the signals transmitted by the STI implementation match the reference. In comparisons B and C, the receive_bit code generates a hardware debugging output signal to indicate when the bus is sampled. These signals are compared to determine the timing error of the bit reception activity.

Table 3: Message Sizes Used in Simulation

<table>
<thead>
<tr>
<th>Data Field Size</th>
<th>0 bytes</th>
<th>2 bytes</th>
<th>4 bytes</th>
<th>6 bytes</th>
<th>7 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte header</td>
<td>2 bytes</td>
<td>4 bytes</td>
<td>6 bytes</td>
<td>8 bytes</td>
<td>9 bytes</td>
</tr>
<tr>
<td>3 byte header</td>
<td>4 bytes</td>
<td>6 bytes</td>
<td>8 bytes</td>
<td>10 bytes</td>
<td>11 bytes</td>
</tr>
</tbody>
</table>

The transmission and reception of 10 random J1850 messages (described in Table 3) were simulated. In each case the messages were correctly received and decoded and the timing jitter was within acceptable limits, as defined in the J1850 specification.

5.3. Timing Accuracy

![Figure 12. Timing error of ASTI transmitter compared with reference (comparison A)](image-url)
First we examine the timing error between the reference transmitter and the integrated send code. Figure 12 shows the distribution of timing error when transmitting messages. There is very little timing jitter and the integrated and discrete transmit functions match closely.

![Reception of Messages Transmitted by Hardware Node](image)

**Figure 13. Timing error of ASTI node receiving message from hardware node (comparison B)**

The timing variations between the reference and integrated versions of the receive routines when receiving a message from the discrete send code (equivalent to a hardware J1850 node) are shown in Figure 13. The maximum timing error is 128 cycles (16 microseconds at 8 MHz, or ¼ of a bit time), which is small enough to meet the J1850 specification.
Figure 14. Timing error of ASTI node receiving message from ASTI node (comparison C)

Figure 14 shows the timing error when an ASTI node receives a message transmitted another ASTI node. The maximum error is 128 cycles for the 8-byte message. This is a 16 microsecond difference between the discrete and integrated versions of the receive routine, and meets the J1850 specification.

These timing errors result from decisions made to simplify integration procedures and their implementations in Thrnt. In particular, certain sections of application code (whether primary or secondary) are treated as atomic and not divisible by a cocall. Multi-cycle instructions are atomic from an instruction-scheduling perspective; the AVR architecture average dynamic instruction duration is 1.3 cycles. The cocall implementation uses a register pair for a computed jump. If this register pair is live at the desired cocall location in the application code, the cocall is delayed until after the use. Similarly, AVR-GCC generates code which disables interrupts briefly when modifying the stack pointer. Intervening primary code and cocalls are placed before or after these sections. Primary code which defines or uses condition-code flags cannot be integrated into sections of the secondary code where these flags are live, as the condition-code flags cannot be reallocated efficiently. Finally, any jitter increases when it occurs with a loop.

Most of these atomicity issues can be solved by inserting padding and then the cocalls before the atomic or live-range sections, resulting in reduced or eliminated jitter. Future work will implement these improvements. In conclusion, the timing of the ASTI-implemented J1850 protocol controller matches the references well enough to meet the specification.
5.4. **Code Expansion**

![Graph showing code expansion after integration using padding loops]

Figure 15. Code expansion after integration using padding loops

Memory is a limited resource in low-end embedded microcontrollers, so we wish to minimize the amount used. ASTI leads to code expansion, mainly due to nop padding and code replication. As seen in Figure 15, the integrated code size for send or receive increases by less than 50% (compared with the original pre-integrated code) and within this, padding accounts for most of the code expansion. The total code size for the system using integration also rises due to the additional copy of the secondary thread. Before integration, 2503 bytes are required for the code (ignoring padding to schedule real-time instructions). After integration 4878 bytes are used. This code expansion, although significant, is acceptable, given the performance increase given for the secondary thread and the resulting ability to drop the system’s clock rate by at least a factor of two.

6. **Conclusions and Future Work**

Software implementations of protocol controllers introduce periods of processor inactivity or idle time within the threads implementing the controller. Due to multiple real-time constraints imposed on the threads, the idle time is fragmented. The idle time can be utilized by enabling another thread to execute during this period. Current techniques for context switching take too much time to efficiently perform this multitasking. This paper introduces a scheme called asynchronous software thread integration for efficiently utilizing the idle time by minimizing the number of context switches required per bit. Coroutine calls are used to perform the context switching. The key idea introduced is the removal of intervening primary code that not only exposes more idle time for secondary thread work but also requires that only two context switches be performed between transmission or reception of bits. The intervening primary code is inserted at the appropriate instants within the host interface thread. Thus the scheme can be used even to extract fine grain idle time and use it for host interface thread work.
A standard automotive network communication protocol (J1850) was implemented using ASTI on an 8 MHz, 8 bit microcontroller. The resulting code, when simulated, executes the secondary thread much faster (at least 100% faster) than an interrupt-based approach.

Modifications and additional work could make the approach even more efficient. We are currently working to remove the current restriction against the host thread containing subroutine calls by using procedure cloning and inlining. Using register file partitioning instead of saving registers on the stack would reduce the number of cycles required to perform a coroutine call, improving performance and enabling the recovery of finer-grain idle time. Total code size could be reduced by using a single host interface thread for integrating with the real-time threads. This requires inserting the intervening primary code and coroutine calls of all the real time threads within a single host interface thread and placing guard code in front of them to ensure the correct code executes. Equalizing inter-bit-call times through code motion rather than padding would increase idle time and improve performance.

7. References

[2] Scott George, “HC05 Software-Driven Asynchronous Serial Communication Techniques Using the MC68HC705J1A,” Motorola Semiconductor Application Note AN1240
[3] Mark Glenewinkel, “Interfacing the MC68HC705J1A to 9356/9366 EEPROMs,” Motorola Semiconductor Application Note AN1241
[7] Naji Naufel, “Interfacing the 68HC05C5 SIO/SP to an I2C Peripheral,” Motorola Semiconductor Application Note AN1066
[9] Peter Topping, “An RDS Decoder using the MC68HC05E0,” Motorola Semiconductor Application Note AN460
[27] Bill Beacham, Hamilton Standard (United Technologies Corporation), personal communication, March 1995