System-Level Issues for Software Thread Integration: Guest Triggering and Host Selection

Alexander G. Dean and John Paul Shen
Carnegie Mellon University, Pittsburgh, PA 15213
adean@ece.cmu.edu

Abstract

Software thread integration provides low-cost concurrency on general-purpose processors by automatically interleaving multiple threads of control into one. This simplifies hardware to software migration and can help embedded system designers meet design constraints. Previous work describes how to efficiently integrate threads.

In this paper we demonstrate how to link trigger events with guest thread execution and how to analyze an application to determine which threads to integrate. The analysis involves timing measurement and prediction to identify the amount of easily accessible temporally deterministic code within each function. This information is used to predict quantitatively the impact of design decisions on system efficiency and help guide integration. To illustrate this process we evaluate an application predicted for the year 2005, when $20 buys a 2000 MIPS embedded processor -- a software-based high-resolution MPEG video player.

1 Introduction

Hardware to software migration enables embedded system designers to move functions from dedicated hardware into software running on a general purpose microprocessor or microcontroller. This transition can help designers meet tight design constraints such as cost, size, weight and function availability by eliminating hardware components. It also can help by giving the implementation the flexibility of software, allowing creation or customization of soft peripherals which replace the hardware components.

The pressures of embedded system constraints have led designers to perform extensive ad hoc hardware to software migration (HSM), as revealed by the many application notes and articles describing how to implement soft peripherals on a microprocessor [7, 9, 11]. These solutions suffer from one if not two shortcomings, depending upon how the software threads performing the hardware functions are scheduled and invoked. Interrupt-triggered guests, while easy to program, incur context switching penalties which reduce efficiency and limit guest execution rates. In addition, the preemptive nature introduces variability into the timing of other threads, which may be real-time as well. Busy wait scheduling passes idle time by executing nop instructions, reducing system efficiency.

In [4, 5, 6] we introduced software thread integration (STI), a novel compiler technique which merges multiple program threads of control into one. This integration uses code transformations to create interleaved (and hence concurrent) code which runs efficiently on general purpose unprocessors. STI's concurrency can provide benefits beyond those of basic multithreading. System resources can be better utilized, as more instructions are available to be scheduled. Context switching can be eliminated, increasing system efficiency. Error detecting code can be integrated to offer concurrent error detection. Real-time guest software threads can be written to replace dedicated hardware and then integrated with existing application code, allowing migration of functions from hardware to software.

We have concentrated on the opportunities STI offers for hardware to software migration in embedded systems. By eliminating hardware components such as dedicated network protocol interfaces or display controllers, designers can reduce unit cost, size and weight as well as eliminate hard-to-obtain or single-source devices. Performing functions in software offers much more flexibility to system designers, allowing them to develop custom algorithms, react to changing standards and offer in-the-field upgrades. Perhaps most dramatic reason for implementing functions in software rather than hardware is the ever-falling cost of instruction processing on general purpose processors, with the current price being under 5 cents per MIPS.

1.1 Software Thread Integration

STI works by interleaving multiple threads at the assembly instruction level. Using a hierarchical representation simplifies program analysis and manipulation. Figure 1 presents a code fragment with both its control-flow (CFG, Figure 1a) and control-dependence (CDG, Figure 1b) graphs. A CDG contains code, predicate, group and call nodes. A code node contains a basic block of instructions and is a leaf node. A predicate (conditional) node is connected with its children with labeled control edges (True, False, Always) which describe the condition under which a child is executed. A group node can be a multi-pred (with multiple immediate control dependences), a loop, or an irreducible loop (which contains multiple entry
points). Edges in the CDG emanating from a node are ordered from left to right to represent the original control flow order. This enables reconstruction of the CFG by a left-to-right traversal of the CDG. Executing the CDG involves the same traversal.

This hierarchical nature of the CDG simplifies code motion. Moving a node with respect to its siblings but retaining the same parent is simply a matter of reordering the parent-child edges. Moving a node into or out of another node’s subgraph requires more involved transformations, which have been developed. Transformations for loops include fusion, peeling and splitting, while predicates can be transformed through region scheduling, renaming, node splitting and duplication [1, 8, 12, 13, 15, 16, 19].

STI uses these transformations to merge multiple threads into a single efficient thread, placing time critical nodes at appropriate locations to meet timing requirements. HSM involves replacing hardware circuits with real-time software which we call guest code. Within this guest code, shown in Figure 1a, are instructions which must execute at specific times with some error tolerance (in blocks with a heavy border). If the processor is fast enough, there will be idle time between some instructions. Figure 1c shows an example of timing requirements for the guest code of Figures 1a and b. As the processor speed increases, the amount of idle time rises as well, reducing system efficiency. As mentioned earlier, the existing methods of using this idle time can become quite inefficient for some designs.

Before actual integration, a suitable host function must be found. This function must run frequently enough and without too much internal jitter in order to meet the guest’s timing requirements. This paper shows how to select appropriate hosts.

Figure 1. Overview of Software Thread Integration

Once a potential host function (or a set of such functions) has been found, integration can begin. First, static timing analysis [14] of the host CDG predicts its execution time bounds (see Figure 1d). The guest code CDG, marked with timing requirements, is initially appended to the host code as the right-most child of the host procedure, effectively putting the guest code at the end of the host code. Each real-time guest node is then moved to the left until reaching its target execution time range. Non-real-time guest nodes are moved left, retaining their original order in the guest. If a real-time node’s time range target completely contains the gap between two children of the host procedure node, the guest can simply be moved to the new location. For example, guest nodes GTIT0 and
GT1F0 (mutually exclusive predicate children) must start no sooner than cycle 10 but no later than cycle 30 (designated as the time range [10, 30]), so the subgraph containing them (rooted by node GT1) can be moved to the gap between host nodes HT1 and HT2, which will occur during the time range [13, 18]. If the host’s jitter of 5 cycles is excessive for the guest, it can be padded to regularize code timing.

If there is no appropriate gap at this level of the host CDG hierarchy, greater timing resolution is needed and the guest code must be moved down into the subgraph of the node executing at the target time. Guest nodes GT3T2 and GT3F1 must execute at [120, 140] but there is no such gap at the root level of the CDG. Loop node HT2 executes during the entire target time range, so it is entered and its children are examined for a suitable gap. The target time [120, 140] falls between the execution of HT2T1 and HT2T2 on the second iteration (i == 1) of loop T2, so the guest subgraph rooted by GT3 is moved there. Because the guest code was moved into a loop, a guard predicate must be added to ensure the guest code only executes in iteration i == 1. During code motion timing analysis, the timing effects of both previously performed and potential code motions are considered.

1.2 Previous Work

Previous work on task fusion at such a fine grain level by a compiler is limited. [13] presented a technique called interleaving for compile-time task scheduling which identifies a task’s idle times and schedules other tasks at those times. It is a coarse grain approach and incurs context-switching penalties for each task switch. [19] extends this work to provide non-intrusive real-time task monitoring. However, as the fragment size decreases, the performance penalty exacted by context switching increases. We have found no work other than ours which eliminates this penalty by merging two threads at the assembly instruction level into a single integrated thread. Previous work in code motion with a PDG ([1, 8, 12, 13, 15, 16, 19]) focuses on reducing program run time.

[4] introduced STI for HSM, presenting background, basic concepts and perspective on its position in the design space of embedded real-time systems. It also considered the impact of modern microarchitectures and architectures on STI/HSM assumptions.

[5] presented the code transformations needed to perform such operations easily while generating efficient code, especially fused loops. As most of a program’s time is spent executing loops, any inefficiency in an integrated loop is multiplied by the iteration count.

[6] demonstrated the use of STI/HSM to implement a bit-banged communication network protocol bridge for a high-temperature (200°C/392°F) application. Although no high-temperature CAN controller integrated circuit existed, a high-temperature 8051-compatible microcontroller did. STI/HSM enabled efficient migration of a CAN controller’s functions from hardware into software so it fit within an 8-bit microcontroller’s meager 1.8 million instructions per second (maximum) instruction processing throughput. The system delivers reliable, simultaneous 10 kbps CAN and 9.6 kbps serial communications.

1.3 Overview

These previous works [4, 5, 6] present motivation for STI, describe how to integrate real-time threads and present performance results. However, they do not address how to make a triggering event cause a guest thread to begin execution. Furthermore, they do not present a method for analyzing a potential application to find suitable host functions. In this paper we present runtime support techniques to recognize a trigger event and invoke a guest thread, taking into account real-time constraints and unpredictable host function invocation. We then develop methods (built into our new tool Thrin) to analyze an application, finding good potential host functions through profiling and static timing analysis. This provides guidance to select threads to integrate. We use these methods to predict the performance of an integrated application without actually having to perform the integration. This paper introduces methods with the goal of extending software thread integration from merely a code motion mechanism to a viable technique for moving real-time functions from hardware to software.

![Figure 2. Microcontroller of 2005: 2000 MIPS for $20](image)

We use a hypothetical future application to illustrate these techniques. If present industry trends continue, in 2005 a $20 embedded microcontroller will offer a 2000 MIPS/2000 MHz 32 bit CPU with 4 MBytes of Flash and 1 MByte of SRAM onboard, with a 100 MHz off-chip bus, as shown in Figure 2. This prediction may in fact be overly pessimistic [2, 3].

We use this future microcontroller as a platform for a digital video (MPEG) playback device. STI enables soft-
ware to replace dedicated LCD display controller hardware, reducing system size, cost and weight while adding design flexibility. STI can also improve the performance of existing software which uses the off-chip bus, which is a bottleneck at 1/20th the speed of the CPU core.

2 Guest Triggering Methods

Our previous work in STI describes how to automatically merge multiple threads of control into one efficient thread. However, most integration results in significant code expansion because of code transformations such as duplication, padding, unrolling and guarding. As a guest is integrated into multiple hosts, system efficiency (useful work performed) rises, but more program memory is required. Careful selection of host functions will provide adequate performance while minimizing memory requirements. Finding this set of hosts requires first selecting a guest triggering method and then choosing the best hosts for that method.

Two design decisions must be made when choosing how to trigger the guest, as illustrated in Table 1. First, the event can be recognized through polling or an interrupt. Second, the recognizer can trigger a thread containing the guest thread immediately or defer such service.

Table 1: Guest Triggering Mechanisms and Examples

<table>
<thead>
<tr>
<th>Event Recognition</th>
<th>Event Service</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Immediate</td>
</tr>
<tr>
<td>Polling</td>
<td>High priority send message, or little guest idle time</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Receive message (fast bus or complex system)</td>
</tr>
</tbody>
</table>

The choice of the event recognition method (polled or interrupt-driven) depends upon the timing requirements and behavior of the host and guest functions, as well as allowable guest dispatch latencies. Polling requires fewer hardware resources, but an interrupt may be needed because of latency, efficiency, determinism or host complexity. In this section we present reasons and methods for choosing and implementing a guest triggering method.

The choice of event service method (immediate or deferred) depends upon the available slack time (deadline minus service time) as well as guest idle time. If there is enough slack time available, the system can avoid using an inefficient thread to service the guest immediately, instead allowing an efficient version to be executed some time later. This increases system efficiency, which is the fraction of time in which the CPU performs useful work rather than padding instructions (which provide time delays). Note that for a hard real-time system, the CPU throughput must be sufficient to meet all real-time requirements, even if all guest events are serviced with dedicated busy-wait threads.

With deferred service, the central integration issue is finding enough host functions to ensure guests are executed often and soon enough. With immediate service, the focus is on finding host work which can be performed in the guest’s idle time. As the trigger will occur asynchronously, the integration tool cannot determine what host work will be available, so it must provide conditional execution of host code. Integrating a guest with code from multiple hosts improves efficiency by increasing the chance that the guest will be able to perform some useful host work. In this paper we examine deferred guest execution only.

2.1 Event Recognition

We begin by examining the timing requirements and characteristics of the guest function. If the guest trigger event is periodic, we can directly integrate a recognizer as a polled implementation into host functions as needed to meet frequency and latency requirements.

Figure 3 illustrates the decisions involved in choosing a guest triggering method. If the guest trigger event is not periodic, we attempt to convert it into an efficient periodically polled system. If the periodic solution is infeasible or excessively inefficient, or if there are no appropriate host functions available, an interrupt is used to trigger the guest.

2.1.1 Polling (Time-Triggered)

The timing for a generic guest event appears in Figure 4. A guest trigger event occurs at most every \( T_E \), requiring the guest’s work to be completed before the deadline \( T_D \).
A guest trigger as mentioned above.

Guest events with short interarrival periods or short deadlines require frequent polling. Unless the guest function can be integrated with a host which runs frequently and predictably, the guest will need to be integrated with many hosts, possibly in several locations in each host. This increases code size and run-time overhead, reducing the attractiveness of polling. In some systems, host function call behavior is unpredictable at compile-time because of system complexity, making hosts for polling the guest trigger difficult to find.

2.1.2 Interrupt (Event-Triggered)

Recognizing a guest trigger with an interrupt provides an attractive solution to the problems of polling as it eliminates both the complexity of deciding when to poll and the code and time overhead of polling. However, it does incur delays from at least two context switches and it requires a free interrupt.

Either an external or an internal interrupt can be used to recognize an event, depending upon its nature. If the guest is event-triggered, an external interrupt is used, while if it is time-triggered, an internal interrupt can be generated by a timer. An event trigger can be converted into a time trigger as mentioned above.

An external interrupt may be connected to a hardware signal indicating an event has happened. This is the standard use of hardware interrupts. For example, the interrupt may be triggered by a high to low transition on a communication bus, indicating the start of a message frame and causing the execution of a function which receives the message.

An internal interrupt may be triggered by a timer to ensure periodic guest execution in a system without suitable periodic host functions. For example, the LCD row refresh function in [5] uses this method to ensure a pair of display rows is refreshed every 60 µs regardless of other activities on a hand-held computer, as they are impossible to predict at compile time.

2.2 Event Service

a) Immediate Service

b) Deferred Service

Figure 4. Polling Timing

Figure 5. Guest Service Overview
After a valid guest trigger event has been detected by the recognizer, a thread must be activated within a limited time to provide guest service. There may be several threads containing the guest; a timely and preferably efficient one must be selected. The slack time (how late guest service can begin and still meet the deadline) determines how elaborate a method of guest thread selection and dispatching can be used.

If there is little slack time, the guest must be serviced immediately. Figure 5a shows that after trigger recognition, the service dispatcher examines a table of the available integrated functions, looking for one ready to execute (i.e., with input data ready) with the highest efficiency. If one is found, it is executed. Otherwise a dedicated guest function executes; its internal timing is set through busy waiting so it is inefficient. [5] uses this technique to share line rendering work between application code and an integrated LCD refresh function. This requires manual source code modification to restructure work into a client-server structure. The change allows the original host and any integrated versions to request work from a common data pool. This will be examined in detail in future work.

If there is significant slack time, guest service can be deferred in the hope that an integrated function will service the guest efficiently at some time before the deadline. Figure 5b shows how the recognizer sets a flag indicating the guest is ready to start, but also sets a watch-dog timer (WDT) to expire nearly at the end of the slack time. The recognizer then returns, allowing the previously running software to continue. If in the course of execution an integrated version of the guest is reached, it disables the WDT and efficiently services the guest. However, if no such version begins execution before the WDT expires, its ISR executes a dedicated guest function (which is implemented with busy waiting).

In both the immediate and deferred service cases, a dedicated guest thread ensures the guest is serviced soon enough to meet its deadline. The dedicated thread helps deal with applications in which call schedules for integrated functions are not known a priori, and it cannot be guaranteed such a function will be called soon enough to meet any deadlines.

Deferred service uses a state flag to control guest execution, as shown in Figure 6. If more than one guest must be executed to complete the guest work, a counter variable is added to keep track of remaining guest work, such as loop iterations. The guest threads update this counter as they execute. Figure 7 illustrates this situation.

3 Code Timing Analysis and Host Selection

The choice of host functions for STI determines the resulting system’s efficiency, latency, throughput and memory requirements. Poor performance in any of these categories can make an STI implementation of a real-time embedded system infeasible. In this section we show how to analyze the application code and guest code. We use this data to find suitable host functions (which help mitigate STI’s drawbacks) and then implement the run-time support needed to create a functioning system.

We begin by analyzing the computing and timing requirements of the guest code. We examine the system’s application code behavior over time, identifying segments of code which can be made sufficiently deterministic at a low cost. Finally we select host functions and predict resulting system performance.

Embedded systems are typically reactive and feature a cyclic structure. A control loop executes frequently enough to meet system timing requirements as governed by a schedule. A static schedule simplifies the task of STI in a real-time system, as it can be used to build a table of information about function call behavior over time. This table can then be used to select host functions which will ensure a guest’s real-time requirements are met.

3.1 Guest Analysis

The goals of guest analysis are first determining if a guest thread is worth integrating and then finding its timing characteristics, deriving them from timing constraints
and static timing analysis. By finding when guest events and processing occur, we also identify idle time which may be reclaimed for host processing.

A trigger event initiates each guest. This may be a state change of a hardware signal or an internal system variable, or a timer overflow, or some similar event. The guest must respond to this event before \( T_{\text{deadline}} \) has passed.

![Figure 8. Guest Timing Analysis](image)

Not all potential guests should be integrated; see [4] for details. A thread with little internal idle time (as identified through static timing analysis) does not waste much processor throughput. A guest which executes infrequently may also load the processor very lightly, reducing the need for STI. Finally, a guest may have long sections of idle time which minimize context-switching penalties.

The guest code is prepared for analysis by padding so the execution time is consistent. For example, in figure 8 the first two predicates are padded with 7 and 2 naps. This CDG is then examined for gaps in processing, which are idle time. The total guest duration is 995 cycles, with 701 of those cycles being idle time. The bulk of the idle time is in the loop, with 34 free cycles in each of 20 loop iterations.

This time is short enough that context switching would consume much if not all of the free time, yet long enough to make a busy wait implementation wasteful, so STI may be useful for recovering these cycles.

### 3.2 Host Analysis

We next perform timing analysis for potential host functions in the application, extracting information on deterministic execution segments within the functions. This segment timing information is then used to help select hosts for integration. Thrint examines potential host functions to identify segments of code which are sufficiently deterministic. These segments are good locations for integration because their low timing jitter eliminates or reduces the need for padding instructions, which reduce system efficiency.

Scheduling or profiling information is used to determine how often each function is called. The most frequently called functions are analyzed to find deterministic segments and characterize them with minimum and maximum execution times, as well as padding overhead in cycles and memory size. Deterministic segments are identified by visiting the procedure node’s children from left to right. Nodes are added to a segment until the timing variability becomes excessive. The current segment is terminated before the node which led to the excessive jitter.

The next segment begins with this node. This method enables the identification and use of deterministic code which may be hidden between unpredictable code segments (e.g. a subroutine call of unknown duration), and hence broadens the range of suitable host functions.

Figure 9 illustrates the deterministic segments within a procedure. To create each segment, Thrint begins with the left-most child node of the Proc node (except for call or multipred nodes), defining it to be the segment start node. Its execution time bounds are predicted with recursive static timing analysis as a range from best to worst case. These times are added to the cumulative time \([\Sigma_{\text{min}}, \Sigma_{\text{max}}]\). Thrint evaluates the cost of nop padding to regularizing the segment’s timing, determining cycle cost and program memory cost. Thrint finds the incremental costs for adding each subgraph as well as the cumulative costs.

![Figure 9. Deterministic Host Segments in Host Code](image)

Static timing analysis identifies unpredictable subgraphs containing recursive calls, calls to unpredictable subroutines, and unbounded loops. A segment ends upon reaching a subroutine call, the end of the procedure or a node which makes the segment’s total jitter exceed the limit determined by the guest. This may be a predicate with cases of very different lengths, a loop with such a predicate, or a loop with an unknown number of iterations. The following segment may begin with the immediate right sibling of the proc child node containing the unbounded node. Alternatively, a segment may be contained completely within a loop body.

Note that a sufficiently deterministic segment may be surrounded by unpredictable (or overly variable) code. This segment can still be used as a host given two constraints. First, the guest must fit completely within the segment (in terms of duration). Second, a WDT must be used to ensure a back-up dedicated guest service thread begins on time. If the guest thread is too long to fit the segment it may be partitioned. This will be discussed in future work.
3.3 System Performance Predictions

After characterizing the application code and guest code, we are ready to select host functions. The goal of integration for deferred service is to cut the system’s processing requirements by choosing the best host functions for integration. An alternate goal might be to integrate until average guest service latency falls to a desired level.

The host functions which have been integrated with the guest thread off-load work from the dedicated, inefficient busy-wait guest, increasing system efficiency. This dedicated guest is triggered by a watchdog timer if the integrated functions have not performed enough guest work to meet the deadline. In a hard real-time system, the CPU’s throughput must still support the worst case conditions, with all guests serviced by dedicated busy-wait routines. By integrating the guest with more hosts, the overall system efficiency can be increased. The drawback of integrating more functions is an increase in code size, with each integration at least adding as much code as is in the original guest (and potentially the host as well, if optimizing for speed).

The first step in host selection is to build a list of potential host functions which contain deterministic segments as long as the guest, sorted by deterministic segment cycles per second. The segment length is weighted by execution frequency, derived from an execution schedule or through profiling.

The second step is to predict the results of integrating the guest with each host. The host segments used and guest code must be padded with nops to regularize the execution timing enough to meet guest requirements. We assume that each host function is copied for integration. This represents the worst case, as it is possible to duplicate only the portion of the host code which will be integrated with the guest, with a guard predicate controlling whether the original host or integrated code is executed.

The memory cost from actual integration is predicted by examining how much guest code replication is needed to place each guest at the appropriate time in the host. Placing a guest in a host predicate requires replication; nested predicates require multiple steps of replication. Fusing loops requires unrolling the host or guest loop, with the memory cost growing with the unrolling factor as defined in [5], (essentially the ratio of longer to shorter loop body).

The final step consists of selecting enough host functions so that integration will bring the predicted system throughput requirement down to the desired level. This process may be stopped early because of various reasons. All candidate host segments may have been used for integration, any more integration would expand the output code size beyond a user-specified limit, or the incremental gain in efficiency for the next integration may be below a user-specified threshold.

4 Experimental Results

To investigate the suitability of STI in a real application we examine a high-resolution MPEG [10] video player. We use the Berkeley MPEG-1 video decoder mpeg_play [17] to decode a bitstream into a CCIR 601 image, with a frame resolution of 720x486 pixels at 30 frames per second. We evaluate how to integrate code to refresh a color liquid crystal display of the same resolution across a 100 MHz bus.

The mpeg_play code is profiled on a DECstation 3000, with a 133 MHz Alpha 21064. We scale the clock rate to target what we expect to be a common high-performance embedded CPU in 2005, offering 2000 MIPS of performance for under $20. This performance prediction may be too pessimistic, given high-end embedded processor product announcements from manufacturers [2, 3]. Designers of the latter processor (to be introduced in late 2000) plan to offer 500 to 1000 MIPS of throughput. Both of these devices will offer high throughput with scalar rather than superscalar instruction processing. This instruction issue method, combined with substantial on-chip memory (or locked cache), will make system timing analysis tractable.

The CPU’s external bus runs at 100 MHz and is a bottleneck, reducing system efficiency. This bus speed is limited by interface, noise, power and timing issues, which are often critical in embedded systems. The bus drives a color LCD with a 100 MHz interface, with timing extrapolated from an existing high-resolution color display with a 67.5 MHz bus [18]. On-chip memory consisting of 1 MByte of SRAM enables the CPU core to run at full speed when not accessing external devices. The baseline video player system uses a dedicated busy-wait refresh function and requires a total of 1870 MIPS of processing: 1380 MIPS for MPEG decoding and 490 MIPS for display refresh.

We compile mpeg_play with gcc 2.7 with full optimization (-O3) and then process it with Pedigree [15], which performs control- and data-flow analysis, and Thrint, our new tool which performs static timing analysis, integration planning and data visualization. Thrint uses profiling information derived from gprof to help plan integration.

4.1 Guest Analysis

LCDRefreshDRow is a function which generates the control and data signals needed to refresh a 720x486 resolution color LCD at a 70 Hz frame rate. As the LCD is double scanned, this function is called at a rate of 17.01 kHz (every 58.8 microseconds) to load 1440 pixels of
data into the display’s column driver input shift register. A horizontal synchronization signal HSync then loads the shift register data into registers which supply data to the column drivers.

This sequence implies that timing jitter when loading the shift registers will have no impact on display quality, as long as set-up and hold times are met. On the other hand, timing jitter in producing the HSync signal will affect the amount of time a row is displayed, affecting the row’s brightness and contrast. These constraints must be observed when deciding how to integrate the function.

The 20-to-1 speed mismatch between the CPU core and LCD bus results in significant fine grain idle time within `LCD_Refresh_DRow`. Table 2 and Figure 10 show how only five useful instructions need to be executed over a period of 20 instruction cycles, repeated 1440 times per call. The resulting idle time is too short to be easily used for other purposes and is wasted, effectively quadrupling the cost of implementing LCD refresh in software from 122 MIPS to 490 MIPS. Overall, the guest thread’s worst case execution time (WCET) is 14.4 µs, 10.8 µs of which is idle time.

In this example we use deferred execution of guest work. The event recognizer, triggered by a 17.01 kHz timer interrupt, sets a flag to enable guest thread execution and also sets a watchdog timer to expire at the end of the slack time (44.4 µs).

### 4.2 Host Analysis

In order to demonstrate the presence of suitable deterministic segments of a useful size within typical application code, we analyze functions from `mpeg_play`, compiled with maximum optimization by gcc. We profile the program as it decodes a CCIR 601 format MPEG file (720 x 486 pixel image, 30 frames per second) and then analyze seven of the top eight functions, covering 90.8% of the program’s execution time. `ParseMacroBlock` is not analyzed due to its complexity. The analysis reveals a large amount of code which can easily be made deterministic and used for thread integration.

#### Table 3: Host Function Profile Information

<table>
<thead>
<tr>
<th>Function</th>
<th>Cycles per call</th>
<th>Calls per second</th>
<th>Cycles per second</th>
<th>% CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrderedDitherImage</td>
<td>12.8M</td>
<td>30</td>
<td>384M</td>
<td>27.8%</td>
</tr>
<tr>
<td>_rev_dct</td>
<td>2874</td>
<td>106k</td>
<td>306M</td>
<td>22.2%</td>
</tr>
<tr>
<td>ReconPMBlock</td>
<td>1367</td>
<td>131k</td>
<td>180M</td>
<td>13.1%</td>
</tr>
<tr>
<td>ReconBIMBlock</td>
<td>2372</td>
<td>62.9k</td>
<td>149M</td>
<td>10.8%</td>
</tr>
<tr>
<td>ParseReconBlock</td>
<td>1269</td>
<td>115k</td>
<td>147M</td>
<td>10.7%</td>
</tr>
<tr>
<td>ParseMacroBlock</td>
<td>2035</td>
<td>39.5k</td>
<td>80.5M</td>
<td>5.86%</td>
</tr>
<tr>
<td>ReconIMBlock</td>
<td>1995</td>
<td>24.7k</td>
<td>49.2M</td>
<td>3.55%</td>
</tr>
<tr>
<td>ReconBIMBlock</td>
<td>1911</td>
<td>18.4k</td>
<td>35.1M</td>
<td>2.55%</td>
</tr>
</tbody>
</table>

The functions profiled are listed in Table 3 with instruction cycle time scaled for the 2000 MIPS CPU. These counts include the effects of cache misses. Except for `ParseReconBlock` and `ParseMacroBlock`, these functions make no subroutine calls, making integration easier by allowing deterministic segments to be longer.

Most of the functions are rather short (a few thousand cycles worst-case) but are called very often. `OrderedDitherImage` is an exception as it lasts nearly ten million cycles but is called once per frame. A function’s duration limits its longest deterministic segment, which in turn limits how large a guest can be integrated.

#### Table 4: Static Timing Analysis and Deterministic Segments for Host Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Predicted Execution Time</th>
<th>Deterministic Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best Case</td>
<td>Worst Case</td>
</tr>
<tr>
<td>ParseReconBlock</td>
<td>114</td>
<td>no bound</td>
</tr>
<tr>
<td></td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>ReconBM-Block</td>
<td>115</td>
<td>2250</td>
</tr>
<tr>
<td>ReconBIM-Block</td>
<td>964</td>
<td>1673</td>
</tr>
</tbody>
</table>
Table 4: Static Timing Analysis and Deterministic Segments for Host Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>Length</th>
<th>Cost</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReconIM-Block</td>
<td>1068</td>
<td>1071</td>
<td>1071</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>ReconPM-Block</td>
<td>123</td>
<td>2261</td>
<td>127</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>j_rev_dct</td>
<td>1674</td>
<td>4258</td>
<td>14</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OrderedDitherImage</td>
<td>9.73M</td>
<td>9.73M</td>
<td>9.73M</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4 presents the static timing analysis predictions and deterministic segments found within the functions. Each segment is described by both length (duration in cycles) and cost (how many cycles of nops must be added to remove all timing jitter). Surprisingly, OrderedDitherImage, which loads the CPU the most, is fully deterministic (for its 9.73 million cycles). This will simplify integration. The other functions offer much shorter deterministic segments.

OrderedDitherImage is not plotted, given its extremely long and fully deterministic path. These segments were identified by specifying a jitter of under 1000 cycles. ReconBMBlock and ReconMBlock both offer long, highly deterministic segments with very low or no padding costs. j_rev_dct has a loop which contains both fully deterministic code and less deterministic code, resulting in the distinctive stair-step pattern of maximum jitter.

Figure 12. Host Function and Segment Processing Loads

We find the total CPU throughput spent executing deterministic code by scaling the segment size by call frequency. Figure 12 shows how many cycles each host function spends per second (dynamic cycle count) executing deterministic code segments, as compared with the minimum and maximum total cycle counts. By adding the deterministic dynamic cycle count in each host, we find 535 MIPS of easily accessible deterministic code which can be used to fill idle time in guest functions. This level suggests that integration will produce efficient code which can mask a significant amount of idle time.

4.3 Predicted Performance Results

Table 5 presents the amount of guest work which will be executed by integrated threads. Predicting this value requires finding how many guest iterations fit into the host’s deterministic segments. These iteration counts are then weighted by the segment and host execution frequencies. For example, in j_rev dct two deterministic segments (91 and 154 cycles long, the latter in an eight-iteration loop) are used to refresh 64 pixels. The integrated thread is called 106,000 times per second, so 6.78 million pixels are refreshed per second, which is 27.7% of the 24.49 million required.

However, the row refresh frequency requirement limits the amount of guest work which OrderedDitherImage can perform. Only 10.8 µs of the host’s deterministic instructions can be used every row period (58.8 µs); the rest are wasted. OrderedDitherImage spans over 82 of these row...
periods, so the remaining 44.4 µs of deterministic code per row period are unavailable.

Table 5: Predicted Thread Integration Results

<table>
<thead>
<tr>
<th>Host</th>
<th>Pixels Refreshed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>per Call</td>
</tr>
<tr>
<td>OrderedDitherImage</td>
<td>119,520</td>
</tr>
<tr>
<td>j_rev_dct</td>
<td>64</td>
</tr>
<tr>
<td>ReconPMBlock</td>
<td>6</td>
</tr>
<tr>
<td>ReconMBlock</td>
<td>53</td>
</tr>
<tr>
<td>ReconBitMBlock</td>
<td>47</td>
</tr>
<tr>
<td>ReconBMBlock</td>
<td>6</td>
</tr>
<tr>
<td>ParseReconBlock</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 13 presents the predicted code expansion for integrating each host function with the guest. Each host function is copied prior to integration so that at run-time both the original host and the integrated host/guest thread are available to run. It is possible to perform each prediction twice, optimizing first for minimum code size and second for run time efficiency. In this example we optimize for run time and hence pay a penalty for memory use.

Figure 14. Deep Conditionals in ReconBMBlock

The overall system throughput required is predicted at each step. In order to simplify the task of predicting system performance, we assume that over the long term the host functions are called periodically and the phase differences between calls to different functions are uniformly distributed in time. A system without function calls matching these characteristics will still work with STI because the system is designed to handle the worst-case conditions. However, the uniformity assumptions will be violated, so performance predictions will be less accurate.

By integrating the guest with these seven hosts, an average of 64.3% of the idle time is reclaimed for useful work. The remaining pixels consume 253 MIPS for display refresh.

Figure 15. Code Memory vs. System Throughput

Figure 15 shows cumulatively how required CPU throughput falls and total code memory rises as the guest thread is integrated into additional hosts. The hosts are selected in order of the most pixels refreshed per second. Little performance is gained by integrating the guest into
more hosts beyond the sharp knee after ReconIMBlock. Despite the significant amount of guest code replication, the total code expansion of 40.6 kB is moderate compared with the size of the entire program (270 kB). This could be reduced, at the expense of performance, by not duplicating each host function completely, instead inserting additional guard instructions.

5 Conclusion

In this paper we have introduced system-level methods and concepts to support software thread integration for real-time systems. Guest triggering connects an internal or external event to the execution of a guest thread either immediately or later before its deadline. Application code analysis finds potential host functions which contain deterministic code segments suitable for integration, building the segments by adding code until jitter becomes excessive. It then guides the selection of hosts to create an integrated application which operates efficiently.

We demonstrate these new methods using a software-based high-resolution MPEG player with integrated software display refresh. Within this code we find a significant amount of nearly deterministic code, allowing the recovery of 236 MIPS of guest idle time, demonstrating the usefulness of this technique for complex applications.

The techniques introduced in this paper help move software thread integration from a compiler technique for code motion toward a viable system design technology capable of supporting hardware to software migration by analyzing an application and recommending an effective integration course. In the future we plan to demonstrate this technology on embedded application hardware prototypes.

Acknowledgments

This work was funded by a grant from United Technologies Research Center, East Hartford, CT, 06108, and in part by ONR (N00014-96-1-0347).

Bibliography


[17] Lawrence A. Rowe, “Continuous Media Applications”, Multiport Workshop held in conjunction with ACM Multimedia ’94, San Francisco, CA, Nov. 1994
