CARNEGIE MELLON UNIVERSITY

SOFTWARE THREAD INTEGRATION FOR
HARDWARE TO SOFTWARE MIGRATION

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
for the degree

DOCTOR OF PHILOSOPHY
in
ELECTRICAL AND COMPUTER ENGINEERING

by

Alexander Guimarães Dean

Pittsburgh, Pennsylvania
May 2000
Abstract

This dissertation introduces software thread integration (STI) and its use for migrating functions from hardware to software (HSM). STI interleaves multiple software threads at the assembly language level, offering efficient concurrency for general-purpose processors. This concurrency is invaluable for HSM, which requires concurrent execution of multiple real-time software threads. STI provides a set of compiler-implemented code transformations which produce efficient code. Putting the transformations into a compiler lets the system developer work at a higher level of abstraction and therefore with more efficiency.

Existing methods of HSM suffer from a complicated manual process and inefficient output code. Despite these limits, embedded system designers frequently perform HSM to meet design goals. STI provides a mechanism for efficiently sharing the CPU by interleaving multiple real-time software threads into an integrated thread, replacing context switching and busy waiting with useful instructions to raise efficiency. It uses code transformations which can be automated and implemented in a post-pass compiler. This automation lets system developers perform HSM much more efficiently, reducing development time and effort. Timing accuracy, run-time overhead and code size can be traded off to meet a specific embedded system’s goals.

This dissertation contributes automatic methods for guiding, performing, and evaluating STI and HSM. It begins by presenting the code transformations needed to integrate real-time threads. These transformations are implemented in a compiler to automate integration and speed migration. The code produced is more efficient than that of other methods and can be optimized for program size or speed. It shows how to measure thread suitability for integration based upon idle time distribution and segment timing determinacy. It then presents a method to select threads for integration based upon these metrics and profiling information, and then predict the resulting system performance and memory cost. It introduces methods for linking a trigger event with the execution of a service thread. These methods allow a system designer to trade off efficiency for response latency. Fi-
nally, this dissertation also presents other benefits of STI such as concurrent error detection and performance enhancement.

The STI and HSM concepts are illustrated with a variety of embedded applications ranging from low-end (2 MHz 8-bit) to high-end (2000 MHz 32-bit) processors. These applications include audio compression and decompression, video decompression, graphics rendering, image processing and network communication protocols. These experiments reveal a large amount of software suitable for STI because of idle time and temporal determinism. One sample application is developed, built as an industrial prototype and tested for an extended period at high temperature.

We believe that the concepts and techniques presented in this dissertation offer an exciting new way to implement real-time functions in software. Our work formalizes and automates the process of hardware to software migration. Engineers have performed ad hoc HSM manually for point design solutions at great cost. By introducing new, automatic methods we can make embedded system designers more productive and make their designs more efficient and less expensive. STI for HSM can revolutionize embedded system design.
Acknowledgments

I would like to thank various people and organizations who helped me with this work. My parents provided me with imagination and persistence which have proved valuable in the course of my work.

On the academic and industrial sides, Dr. John Shen served as my advisor and Dr. Phil Koopman was my mentor. Carnegie Mellon University has been a stimulating environment with many opportunities. United Technologies Research Center helped first with hands-on experience with embedded systems and second with most of the funding for this work, allowing me to focus on embedded systems and demonstrate my research.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1. HSM Benefits</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2. Current HSM Practice and Problems</td>
<td>3</td>
</tr>
<tr>
<td>1.2. Software Thread Integration</td>
<td>5</td>
</tr>
<tr>
<td>1.3. Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.4. Outline of Dissertation</td>
<td>7</td>
</tr>
<tr>
<td>2. Software Thread Integration</td>
<td>9</td>
</tr>
<tr>
<td>2.1. Overview of STI</td>
<td>10</td>
</tr>
<tr>
<td>2.2. STI Preparations</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1. Program Representation and the CDG</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2. Register Reallocation</td>
<td>14</td>
</tr>
<tr>
<td>2.2.3. Static Timing Analysis</td>
<td>15</td>
</tr>
<tr>
<td>2.3. Integration Procedure</td>
<td>16</td>
</tr>
<tr>
<td>2.3.1. Integration of Single Events</td>
<td>17</td>
</tr>
<tr>
<td>2.3.2. Integration of Looping Events</td>
<td>20</td>
</tr>
<tr>
<td>2.4. Experimental Results</td>
<td>29</td>
</tr>
<tr>
<td>2.4.1. Handheld Computer</td>
<td>30</td>
</tr>
<tr>
<td>2.4.2. Vehicle Cellphone with External Network</td>
<td>33</td>
</tr>
<tr>
<td>2.4.3. Cellphone with Internal Network</td>
<td>36</td>
</tr>
<tr>
<td>2.4.4. Summary of Experiments</td>
<td>38</td>
</tr>
<tr>
<td>2.5. Conclusions</td>
<td>39</td>
</tr>
<tr>
<td>3. Thrint: A Tool for Automatic STI</td>
<td>41</td>
</tr>
<tr>
<td>3.1. Real-Time Embedded System Development Tools</td>
<td>41</td>
</tr>
<tr>
<td>3.2. Tool Chain Overview</td>
<td>43</td>
</tr>
<tr>
<td>3.3. Operation of Thrint</td>
<td>44</td>
</tr>
<tr>
<td>3.4. Future Extensions</td>
<td>57</td>
</tr>
<tr>
<td>3.4.1. Implementation Optimizations and Their Consequences</td>
<td>57</td>
</tr>
<tr>
<td>3.4.2. Making STI Viable for Design</td>
<td>58</td>
</tr>
<tr>
<td>3.5. Conclusions</td>
<td>59</td>
</tr>
<tr>
<td>4. Guest Triggering and Host Selection</td>
<td>61</td>
</tr>
<tr>
<td>4.1. Guest Triggering Methods</td>
<td>61</td>
</tr>
<tr>
<td>4.1.1. Event Recognition</td>
<td>63</td>
</tr>
<tr>
<td>4.1.1.1. Polling (Time-Triggered)</td>
<td>63</td>
</tr>
<tr>
<td>4.1.1.2. Interrupt (Event-Triggered)</td>
<td>64</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.1.3. Guest Triggering and Host Selection</td>
<td>127</td>
</tr>
<tr>
<td>8.1.4. Constraints on Using STI for HSM</td>
<td>127</td>
</tr>
<tr>
<td>8.1.5. Experimental Prototype</td>
<td>128</td>
</tr>
<tr>
<td>8.1.6. STI for Concurrent Error Detection and Performance</td>
<td>128</td>
</tr>
<tr>
<td>8.2. Future Work</td>
<td>129</td>
</tr>
<tr>
<td>8.3. Publications</td>
<td>130</td>
</tr>
<tr>
<td><strong>References</strong></td>
<td><strong>131</strong></td>
</tr>
</tbody>
</table>
# List of Illustrations

<table>
<thead>
<tr>
<th>Fig.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Processing costs per MIPS for 8-bit microcontrollers</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Overview of hardware to software migration with STI</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Overview of software thread integration</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Integrated thread</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Control flow graph and labeled control dependence graph</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Initial (degenerate) integration</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Summary of single event integration</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td>Summary of loop integration with unrolling</td>
<td>22</td>
</tr>
<tr>
<td>2.8</td>
<td>Guest loop unrolling integration example</td>
<td>25</td>
</tr>
<tr>
<td>2.9</td>
<td>Host loop unrolling integration example</td>
<td>26</td>
</tr>
<tr>
<td>2.10</td>
<td>Guest guarding integration example</td>
<td>29</td>
</tr>
<tr>
<td>2.11</td>
<td>Handheld computer hardware components</td>
<td>30</td>
</tr>
<tr>
<td>2.12</td>
<td>LCD refresh timing and CPU activity</td>
<td>31</td>
</tr>
<tr>
<td>2.13</td>
<td>Handheld computer CPU capacity vs. line drawing activity</td>
<td>32</td>
</tr>
<tr>
<td>2.14</td>
<td>Vehicle cellphone hardware components</td>
<td>33</td>
</tr>
<tr>
<td>2.15</td>
<td>CAN message timing and CPU activity</td>
<td>34</td>
</tr>
<tr>
<td>2.16</td>
<td>Vehicle cellphone CPU capacity vs. message activity</td>
<td>35</td>
</tr>
<tr>
<td>2.17</td>
<td>Handheld cellphone hardware components</td>
<td>36</td>
</tr>
<tr>
<td>2.18</td>
<td>Handheld cellphone CPU time vs. message activity</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>STI tool chain overview</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>Thrint overview</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of automatically generated CDG program representation</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Sample integration directives file</td>
<td>47</td>
</tr>
<tr>
<td>3.5</td>
<td>Automatically derived deterministic segment information</td>
<td>48</td>
</tr>
<tr>
<td>3.6</td>
<td>Guest1 moved to execute at T=10</td>
<td>50</td>
</tr>
<tr>
<td>3.7</td>
<td>Guest2 moved to execute at T=35</td>
<td>51</td>
</tr>
<tr>
<td>3.8</td>
<td>Guest3 replicated to execute at T=45, jitter padded out</td>
<td>52</td>
</tr>
<tr>
<td>3.9</td>
<td>Target time 25, tolerance of ±6 cycles</td>
<td>54</td>
</tr>
<tr>
<td>3.10</td>
<td>Target time 25, tolerance of ±4 cycles</td>
<td>55</td>
</tr>
<tr>
<td>3.11</td>
<td>Target time 25, tolerance of ±2 cycles</td>
<td>56</td>
</tr>
<tr>
<td>4.1</td>
<td>Guest trigger issues</td>
<td>62</td>
</tr>
<tr>
<td>4.2</td>
<td>Selecting a guest trigger method</td>
<td>63</td>
</tr>
<tr>
<td>4.3</td>
<td>Polling timing</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>Guest service overview</td>
<td>66</td>
</tr>
<tr>
<td>4.5</td>
<td>Line drawing integrated with LCD refresh</td>
<td>67</td>
</tr>
<tr>
<td>4.6</td>
<td>Timeline for deferred service (one guest)</td>
<td>68</td>
</tr>
<tr>
<td>4.7</td>
<td>Timeline for deferred service (multiple guests)</td>
<td>69</td>
</tr>
<tr>
<td>Fig. 4.8</td>
<td>Guest timing analysis</td>
<td>70</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------</td>
<td>----</td>
</tr>
<tr>
<td>Fig. 4.9</td>
<td>HSCAN idle time bubble size during message transmission and reception</td>
<td>71</td>
</tr>
<tr>
<td>Fig. 4.10</td>
<td>Deterministic host segments in host code</td>
<td>72</td>
</tr>
<tr>
<td>Fig. 4.11</td>
<td>MPEG video player on the microcontroller of 2005: 2000 MIPS for $20</td>
<td>75</td>
</tr>
<tr>
<td>Fig. 4.12</td>
<td>Guest thread timing diagram</td>
<td>76</td>
</tr>
<tr>
<td>Fig. 4.13</td>
<td>mpeg_play segment jitter</td>
<td>79</td>
</tr>
<tr>
<td>Fig. 4.14</td>
<td>Host function and segment processing loads</td>
<td>80</td>
</tr>
<tr>
<td>Fig. 4.15</td>
<td>Integrated thread memory expansion</td>
<td>82</td>
</tr>
<tr>
<td>Fig. 4.16</td>
<td>Deep conditionals in ReconBMBlock</td>
<td>83</td>
</tr>
<tr>
<td>Fig. 4.17</td>
<td>Code memory vs. system throughput</td>
<td>84</td>
</tr>
<tr>
<td>Fig. 5.1</td>
<td>Architectures for function implementation options</td>
<td>87</td>
</tr>
<tr>
<td>Fig. 5.2</td>
<td>Operation timing definitions</td>
<td>88</td>
</tr>
<tr>
<td>Fig. 5.3</td>
<td>HSCAN idle time bubble size during message transmission and reception</td>
<td>90</td>
</tr>
<tr>
<td>Fig. 5.4</td>
<td>Operations and subthreads with traditional HSM</td>
<td>92</td>
</tr>
<tr>
<td>Fig. 5.5</td>
<td>Operations and threads with software thread integration</td>
<td>94</td>
</tr>
<tr>
<td>Fig. 5.6</td>
<td>Sweet spots for migration methods</td>
<td>95</td>
</tr>
<tr>
<td>Fig. 5.7</td>
<td>Example operations and idle time</td>
<td>97</td>
</tr>
<tr>
<td>Fig. 5.8</td>
<td>CPU capacity remaining during average loading</td>
<td>98</td>
</tr>
<tr>
<td>Fig. 6.1</td>
<td>HSCAN hardware diagram</td>
<td>105</td>
</tr>
<tr>
<td>Fig. 6.2</td>
<td>SPDB before testing: HSCAN (top) and Controller (bottom)</td>
<td>106</td>
</tr>
<tr>
<td>Fig. 6.3</td>
<td>HSCAN software overview</td>
<td>106</td>
</tr>
<tr>
<td>Fig. 6.4</td>
<td>HSCAN idle time bubble size during message transmission and reception</td>
<td>108</td>
</tr>
<tr>
<td>Fig. 6.5</td>
<td>Control flow graphs of controller interface</td>
<td>110</td>
</tr>
<tr>
<td>Fig. 6.6</td>
<td>Periodic CPU activity during unrolled CAN bus sampling loop</td>
<td>111</td>
</tr>
<tr>
<td>Fig. 6.7</td>
<td>HSCAN development time</td>
<td>113</td>
</tr>
<tr>
<td>Fig. 6.8</td>
<td>Smart Power Distribution Box after testing</td>
<td>113</td>
</tr>
<tr>
<td>Fig. 7.1</td>
<td>Integrated thread execution</td>
<td>118</td>
</tr>
<tr>
<td>Fig. 7.2</td>
<td>CDGs and thread duplication</td>
<td>119</td>
</tr>
<tr>
<td>Fig. 7.3</td>
<td>Loop jamming</td>
<td>119</td>
</tr>
<tr>
<td>Fig. 7.4</td>
<td>Original CFG and CDG fragments</td>
<td>120</td>
</tr>
<tr>
<td>Fig. 7.5</td>
<td>Integrated CFG and CDG fragments</td>
<td>120</td>
</tr>
<tr>
<td>Fig. 7.6</td>
<td>Optimized and integrated CFG and CDG fragments</td>
<td>121</td>
</tr>
</tbody>
</table>
List of Tables

Table 2.1: Handheld Computer Thread Statistics 33
Table 2.2: Vehicle Cellphone Thread Statistics 36
Table 2.3: Handheld Cellphone Thread Statistics 37
Table 2.4: Costs of Implementing Guest Functions in Software, without and with STI 39
Table 4.1: Guest Triggering Mechanisms and Examples 62
Table 4.2: Example Guest Service Dispatch Table 67
Table 4.3: Guest Thread Characteristics 76
Table 4.4: Host Function Profile Information 77
Table 4.5: Static Timing Analysis and Deterministic Segments for Host Functions 78
Table 4.6: Predicted Thread Integration Results 81
Table 6.1: Code Expansion (in bytes) from Integration with Controller Interface 111
Table 6.2: Code Expansion (in bytes) 112
Table 7.1: Duplication and Integration of Trans 123
Table 7.2: Non-Duplicate Thread Integration 123
Chapter 1

Introduction

Nearly all of the 3.8 billion microprocessors and microcontrollers sold in 1999 were used for embedded systems; only 112 million became CPUs for personal computers, servers and workstations [Can99][IDC00]. Embedded systems are sold in competitive markets, so designers are forced to minimize development and recurring costs, as well as limit system size, weight, power consumption and design time. Often these design pressures can be met by moving functionality from hardware to software (through hardware to software migration, or HSM), since this reduces component count, increases design flexibility and makes more of the system cost depend on the low and consistently falling cost of executing software. HSM is currently performed by writing guest functions to replace the hardware and then using interrupts, polling or busy waiting to ensure that all parts of the functions execute on time. This process is manual, tedious and error-prone, yet it is common industry practice because of the design pressures. The code produced is inefficient if it is unable to reclaim much idle time within the guest functions. This raises system processing requirements and hence costs.

This thesis introduces software thread integration (STI) and how to use it for HSM. STI is a compiler technology for automatically interleaving multiple real-time software threads of control into one integrated thread at the assembly language level. STI uses a hierarchical program representation which simplifies control flow analysis and transformation. Data flow analysis and register reallocation allow arbitrary code motion. Static timing analysis (STA) guides the transformations to place guest code where it will meet its timing requirements. The resulting integrated thread provides efficient low-cost concurrency for general purpose microprocessors and microcontrollers without the need for hardware support such as multiple register banks or windows [HKT93][WW93], multiprocessors [Tosh97][Tosh98] or multithreading [CSS+91][TE94][CSS97]. HSM uses this concurrency to great advantage. The techniques can be implemented in a compiler for largely au-
1.1. Motivation

Embedded systems are found in a wide range of applications and must balance various design constraints such as development and unit costs, size, weight, power, temperature, reliability, lifetime, ease of maintenance and component availability. These constraints influence how a designer partitions a system’s functionality into hardware and software.

1.1.1. HSM Benefits

There are three main reasons for choosing software over hardware: cost, component count and flexibility.

- Running software is cheap and keeps getting cheaper. The cost of executing software is falling about 50% every 18 months as an indirect result of Moore’s Law. This cost reduction is apparent even in 8-bit microcontrollers, as shown in Figure 1.1.

![Figure 1.1](image_url)

**Fig. 1.1.** Processing costs per MIPS for 8-bit microcontrollers

1.1. Improvements in integrated circuit fabrication processes and architecture/microarchitecture are responsible for the performance increase, while economies of scale reduce microprocessor costs. Although any hardware design can be improved by a faster process, and perhaps also with a better architecture, few designs have the...
1.1. Motivation

Production volumes to justify the development costs. Processors are general purpose hardware, so their volumes are large enough to justify the expense of the development costs, and so their price/performance ratios fall much more rapidly than dedicated hardware. This trend will push more functions into software over time.

- A lower hardware component count makes it easier to meet requirements. Assuming the existing processor is fast enough to support the added software load, there are various direct benefits to HSM. The unit cost is lower. With fewer components, the system is smaller and weighs less. Power dissipation falls. Assuming a correct design, the system reliability rises because there are fewer parts to fail.

- Design and maintenance flexibility increase. The embedded system can use custom or proprietary functions not available in hardware. This is a major benefit for systems operating in harsh environments (e.g. high temperature, high radiation), as the selection of components rated for such use is tiny, seriously limiting the designer’s options. Field upgrades become much easier, reducing service costs. In some embedded systems (e.g. elevators and aircraft engines) the product is sold at or below cost; the service contract yields the profit. Many embedded systems have lifespans measured in decades, making maintenance costs a critical issue. Finally, some algorithms or communication protocols require a licensing fee if implemented in hardware. A software approach may eliminate this issue.

1.1.2. Current HSM Practice and Problems

Any method for HSM must address three issues: the representation of hardware functionality in guest software, the mechanism for scheduling and activating the components of these guests, and the assurance of program characteristics such as correctness of semantics and timing as well as system efficiency. These issues and the industry-practice solutions are explained below.

- Hardware functionality is moved to software by writing guest software threads in a high-level language or assembly code. In the current practice, precise timing requirements typically force the programmer to use assembly language for greatest accuracy, which slows down the development process by discouraging methods which improve programmer productivity (e.g. modularity and abstraction) [Broo95].

- The guest threads must share the processor with other application threads, so there must be a mechanism for scheduling and activating them. Current practice uses in-
1.1. Motivation

Interrupts, polling and busy waiting. The guest’s functionality does not require all of the processor’s throughput; instead there is idle time scattered throughout. Large pieces of idle time can be reclaimed by switching contexts (to execute other work), but overhead is incurred with each switch. As the idle time becomes finer, this overhead increases until there isn’t enough time to switch. At this point busy-waiting (padding) must be used. Busy-waiting requires multiple iterations of the count cycles/pad/verify/adjust process, which is very tedious due to limited support in current tools. If this padding is implemented with nop instructions, the processor’s time is wasted and system efficiency falls. Replacing these nops with useful instructions requires code motion, which must maintain control and data correctness, which is described next. HSM with manual code motion has been demonstrated in industry but only in simple systems with a tremendous amount of development effort [Laco98]. Debugging or modifying such a system requires repeating much of this labor.

- Finally, HSM must ensure semantic and timing correctness, and preferably produce efficient code which uses little additional memory.
  - Enforcing semantic correctness requires observing control and data dependences. These are easy to maintain when using current HSM methods (nop padding), as the nops change neither control nor data flow. However, assembly code motion complicates the issue. The control-flow structure is not obvious in assembly code, forcing the programmer to manually derive the control flow graph (CFG) to simplify program comprehension and modifications. Data-flow information is obscured by the reuse of registers. The benefits of a high-level language, including structured code and named variables in registers, are unavailable because of the lack of timing precision when writing software at that level.
  - Timing correctness requires scheduling each guest operation within a host thread to execute during a target time range. The host thread must be analyzed to predict the execution schedule. Current development tools include a simulator with a cycle counter, allowing the developer to estimate execution time along one path. Evaluating multiple paths requires additional simulations, with the user controlling path selection. This is a tedious task which scales up with program complexity.
1.2. Software Thread Integration

- Ideally the resulting code should be efficient and small. The current practice involves adding nops to create time delays, polling an input or switching contexts. These methods increase the run-time of a thread and hence reduce efficiency. Nops also increase code size slightly, though delay loops make this expansion negligible.

In summary, the existing method for HSM is practical only for small, simple systems or those with very loose timing requirements. This results from the tremendous amount of manual activity needed to ensure timing and reclaim idle time. However, the benefits are substantial, leading to extensive ad hoc HSM [Fink97] [Geor] [Herb99] [Holl99] [Bres] [Topp] [Emba96] [Good] [Glen] [Nauf]. One microcontroller maker specifically designs its products for use in HSM and promotes “virtual peripherals” [Burs97].

1.2. Software Thread Integration

STI meets the needs of HSM in all three necessary areas. A hierarchical program representation simplifies code analysis and modification, a variety of CPU sharing methods enable most idle time to be reclaimed and semantics and optimization can be enforced automatically.

- STI uses a control dependence graph (CDG) to represent each procedure in a program. In this hierarchical graph, control dependences such as conditionals and loops are represented as non-leaf nodes, and assembly instructions are stored in leaf nodes. Conditional nesting is represented vertically while execution order is horizontal, so that an in-order left-to-right traversal matches the program’s execution. The CDG is a good form for holding a program for STI because this structure simplifies analysis and transformation through its hierarchy. Program constructs such as loops and conditionals as well as single basic blocks are moved efficiently in a coarse-grain fashion, yet the transformations also provide fine-grain scheduling of instructions as needed.

- STI improves the allocation of a processor’s time by merging multiple threads of control into one. This improves performance over traditional context-switching by reducing the number of switches needed, which cuts run-time. This is also better than busy-waiting because idle time is reclaimed to perform useful work. STI should be used when context-switching and busy-waiting are too slow or inefficient.
1.3. Contributions

- STI automatically ensures semantic and timing correctness with its transformations. The variety of integration methods and decisions enable the STI tool to automatically optimize for execution speed or code size.

- All control and data dependences must be observed to ensure semantic correctness. The CDG explicitly represents control dependences as graph structures; STI’s code transformations modify the graph yet maintain these dependences. These transformations enable STI to interleave code from different threads, which is the key to reclaiming idle time efficiently. STI only needs to handle false data dependences when integrating threads; no other data dependency issues arise because each individual thread remains in order. Assembly code contains many false data dependences because of register reuse, so STI automatically reallocates registers to remove this constraint and make code motion easier.

- All real-time dependences must be observed to ensure timing correctness. Each RT guest instruction must be moved to execute within its target time range. STI automates this process. First the host and guest threads are statically analyzed for timing behavior, with best and worst cases predicted. Then, during integration, timing directives (supplied by the programmer) guide integration. Timing jitter in the host thread is automatically reduced to meet guest requirements. The CDG’s structure makes the timing analysis and integration straightforward.

- STI produces code which is more efficient than context-switching or busy-waiting. The processor spends fewer cycles performing overhead work. The price is expanded code memory. STI may duplicate code, unroll and split loops and add guard instructions. It may also duplicate both host and guest threads. Memory may be cheap, but it is not free. The memory expansion can be reduced by trading off execution speed or timing accuracy. This flexibility lets STI tailor its transformations to a particular embedded system’s constraints.

1.3. Contributions

STI provides a framework for understanding and implementing HSM automatically and efficiently. It draws upon methods from compilers, real-time systems and operating sys-
tems. STI’s novelty lies in the use of code transformations for RT HSM purposes, triggering methods and host/guest thread selection.

• STI uses code transformations to provide low-cost concurrency for HSM, addressing concurrency, timing, code size and execution time issues. Nearly all previous work in code transformation has been for reducing a program’s average run time. There has also been work on transformations to reduce timing variability or code size.

• We introduce metrics for ranking and selecting host functions before integration. These consider timing behavior, code expansion, profiling information and efficiency to allow a cost-benefit analysis of STI options.

• We have developed a post-pass compiler which implements STI for research purposes. We have also demonstrated STI manually in an industrial prototype high-temperature network bridge.

1.4. Outline of Dissertation

Chapter 2 introduces the process of using STI for HSM and then explains STI in detail. It presents the code transformations used to move CDG nodes for both single and looping guest events. As programs spend most of their execution time in loops, efficient loop transformations are important. The results of applying STI for HSM to three applications are presented to demonstrate the benefits.

Chapter 3 describes the research tool Thrint, a back-end compiler which automatically performs static timing analysis (STA) and STI. Thrint fits into a tool chain after the compiler (GCC) but before the assembler (also GCC). The integrated code is assembled and linked by GCC and can be debugged with GDB or DDD. High-level integration decisions can be guided using profiling information derived from GProf. Thrint can help a developer understand program structure and timing behavior by generating visualization aids through XVCG and GnuPlot.

Chapter 4 addresses system-level issues when using STI for HSM. A guest thread may be enabled immediately or after some delay depending upon system efficiency goals. Timing analysis techniques find deterministic segments in hosts. The guest threads are integrated into these segments. The performance of potential host-guest thread pairs is predicted and combined with profiling information to enable thread selection based on a cost-benefit
1.4. Outline of Dissertation

analysis. These concepts are demonstrated with an experiment using an MPEG video decoder and the refreshing of a high-resolution color liquid crystal display.

Chapter 5 characterizes the performance of HSM methods at a low level. It develops methods to mathematically characterize the processor throughput used by HSM for both traditional methods and STI. It then evaluates three issues important to real-time embedded systems: memory system expansion, increased processor throughput requirements and program execution determinism. STI raises program memory size requirements which may increase system cost. The additional load from guests may require the use of a faster processor, raising system costs. Finally, RT requirements make prediction of the host software’s timing behavior critical, though both hardware and software can make this complicated.

Chapter 6 shows how STI was used manually on an industrial prototype to migrate an interface for an embedded network protocol from hardware to software. The application had to support the CAN protocol at a very high temperature, but no hardware CAN controllers were available. However, high-temperature 8-bit microcontrollers do exist, so support for the protocol is added through STI for HSM. The system is built and tested.

Chapter 7 presents other uses for STI. Concurrent error detection (CED) enables a system to detect errors quickly, which is important for making a system reliable. We integrate an error detection thread with an image processing thread to demonstrate the benefits. STI can also increase a program’s efficiency by raising the number of independent instructions available to a code scheduler. This improves the execution schedule by using the processor’s functional units more efficiently and hence reducing total run-time.
Chapter 2

Software Thread Integration

Software thread integration (STI) is well-suited for hardware to software migration (HSM) because of the concurrency it offers. This chapter describes the concepts of STI and how it is used for HSM. Before actual integration begins the threads must be converted to an efficient representation and then analyzed for timing and data characteristics. Then code transformations can be performed to integrate the threads. These preparations and code transformations are presented here along with experimental results.

For perspective, Figure 2.1 presents an overview of how STI is used for HSM. A hardware function is replaced with software written by a programmer. This code consists of one or more guest threads (represented by the blue bar) with real-time requirements. When the threads are executed on a CPU, gaps will appear in the schedule of guest instructions, as illustrated by the white gaps in the blue bar. These gaps are pieces of idle time which can
be reclaimed to perform useful host work. This chapter describes how STI recovers fine-grain idle time efficiently and automatically.

2.1. Overview of STI

STI works by interleaving multiple threads at the assembly instruction level. In order to transform code effectively it depends upon efficient software representation, register reallocation, and static timing analysis as enabling technologies.

STI uses a hierarchical representation to simplify program analysis and manipulation. Figure 2.2 presents a code fragment with both its control-flow (CFG, Figure 2.2a) and control-dependence (CDG, Figure 2.2b) graphs. A CDG contains code, predicate, group and call nodes. Edges in the CDG emanating from a node are ordered from left to right to represent the original control flow order. Executing the CDG involves a left-to-right pre-order traversal of the CDG. This hierarchical nature of the CDG simplifies code motion and encapsulates timing and control information.

Fig. 2.2. Overview of software thread integration
2.1. Overview of STI

Code motion must observe data dependences to produce semantically correct code. Assembly code presents many false data dependences because of register reuse, but dataflow analysis removes these by identifying the thread’s variables and then allocating new registers for them.

Timing requirements for guest code (in the form of target time and error tolerance for critical instructions) are provided by the programmer and then later used to direct code motion automatically. As shown in Figure 2.2a and b, there are instructions within the guest code (in nodes G3, G4, G10 and G12) which must execute at specific times with some error tolerance (in blocks shaded yellow). Figure 2.2c shows an example of timing requirements for the guest code.

Tree-based static timing analysis provides best and worst case timing information for each host node’s start and duration. The analysis is performed from the bottom of the CDG up. This information provides guidance for future code motion. For example, in Figure 2.2d host conditional node H2 can finish executing as early as cycle 13 or as late as cycle 18 (the time range [13,18]).

The guest code CDG, marked with timing requirements, is initially appended to the host code as the right-most child of the host procedure. Each real-time guest node is then moved to the left until reaching its target execution time range. Non-real-time guest nodes are also 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 that gap. For example, guest nodes G3 and G4 (mutually exclusive predicate children) must start within 10 cycles of cycle 20, denoted [20±10]. This translates to a time range of no sooner than cycle 10 and no later than cycle 30, denoted by [10,30]. The subgraph containing them (rooted by node G2) can be moved to the gap between host nodes H2 and H4, which will occur during the time range [13,18]. If the host’s jitter of 5 cycles is excessive for the guest, the conditional node H2 can be padded to regularize code timing.

If there is no appropriate gap at the current level of the host CDG hierarchy, greater timing accuracy is needed and the guest code must be moved down into the subgraph of the node executing at the target time. Guest nodes G10 and G12 must execute within [140±10]. The execution duration of the guest code before G10 and G12 is analyzed and subtracted to determine that G6 must begin execution within [120±7], but there is no such gap at the root level of the CDG. Loop node H4 executes during the entire target time range, so it is en-
2.2. STI Preparations

STI requires several preparatory steps before code motion can be performed. First, CDGs must be created for the host and guest threads. Second, program variables must be identified and extracted to simplify code transformations. Third, the CDGs must be analyzed statically to predict timing behavior.
2.2. STI Preparations

2.2.1. Program Representation and the CDG

Software thread integration uses the CDG (control-dependence graph) representation of a program, which stores program control dependence information concisely and hierarchically, and hence enables efficient code analysis and motion.

Figure 2.4 presents control flow and dependence graphs of a code fragment to illustrate the graph structure. The graph $CDG(N, E)$ (based on the program dependence graph (PDG) [FOW87], extended in [Newb97]), contains control nodes such as Code, Predicate, Group and Call nodes. A predicate node is connected to its children with labeled control edges (e.g. True, False, Any) which describe under which conditions each child is executed. A Group node can be further classified as Multipred (a group of nodes with multiple predicates, which represent immediate control dependences), Loop, and IrrLoop (an irreducible loop has multiple entry points). A Proc node is the root of a CDG and represents a procedure.

The nodes are labeled according to the scheme presented in [Newb97]. This labeling method simplifies node position comparisons, making it easy to determine dominance, control dependence, reachability and order. The labels are efficient and can represent both structured and unstructured code. A program’s control structure has three dimensions: type of condition, order and conditional nesting depth. Each label consists of a string of...
2.2. STI Preparations

Each cell contains a condition such as true or false (or an index for multiway branches) to indicate when the node is executed. Each cell also has a sequence number to describe its execution order. A node’s label represents the conditional nesting depth through the concatenation of labels corresponding to the path from the graph root to the node. For example, node G4’s label is T2F0, indicating that the node is located in section 2 of true child nodes at the top level of the graph (T2), and then in section 0 of the false children within the T2 section (F0).

Edges emanating from a single node are ordered from left to right to capture the original lexical order of the code. This enables reconstruction of the CFG by a left-to-right preorder traversal of the CDG. In addition, executing a program visits the CDG in this left-to-right preorder traversal, entering subgraphs rooted by valid predicates while repeating loop subgraphs.

The PDG (which adds data dependency information to the CDG) has been used to implement code optimizations which reduce the average execution time of programs on uniprocessors and multiprocessors [FOW87] [GS87] [AJLS92]. This research effort has resulted in a group of code motion, deletion and creation transformations which we modify for thread integration.

2.2.2. Register Reallocation

Any significant motion of assembly code is limited by false data dependences caused by multiple variables reusing the same register over time. To remove this constraint STI uses register reallocation. Using the registers most efficiently requires first deriving data-flow information for both host and guest threads to identify variables, replacing each instruction’s register operands with variable references, then performing the code transformations, and finally allocating registers for the variables (e.g. through coloring [CAC+81], [Chai82] or tiling [CK91]).

An alternative approach, which works for threads with low register pressure, is simply to partition the register file and provide a fixed set of registers for each thread. Spill and fill code is placed at either end of the integrated thread. This results in less efficient register use but simplifies implementation. More complex threads with more register pressure will benefit from the full register allocation of the first method.
To support debugging all reallocation information must be given to the programmer. Source level debugging requires that debugging data be maintained by the reallocator for later use by the debugger.

### 2.2.3. Static Timing Analysis

The host and guest CDGs are annotated with execution time information to guide the placement of real-time guest code. Each host node contains a prediction of its best and worst case beginning execution time $T_{\text{beg}}(\text{Host})$ measured in cycles. These times are derived hierarchically. Both host and guest nodes are marked with execution duration predictions ($T_{\text{dur}}(\text{Node})$). Host nodes are marked with predicted ending times ($T_{\text{end}}(\text{Node})$) as well. Each loop is annotated to describe its beginning and ending times as well as iteration duration in the form $[T_{\text{beg}}(\text{Host}), T_{\text{end}}(\text{Host}) : T_{\text{iter}}(\text{Host})]$. Guest target times are adjusted to compensate for preceding guest code.

As discussed in Chapter 5, characteristics of the system’s software and hardware can complicate program execution time prediction. Tree-based analysis determines the timing behavior of the host and guest code. This analysis requires that the threads have bounded loop iterations and no recursive or dynamic function calls. Much embedded software meets these criteria, including the examples used here. Variations in the use of system hardware resources can affect execution time. Caches, pipelines, superscalar instruction dispatch behavior, virtual memory and variable latency instructions will increase the variability of system performance with different data sets. Embedded real-time systems often lack these features because of cost constraints. In our applications we assume these features are not used (e.g. no virtual memory), or are configured to operate predictably (caches are locked or fast on-chip memory is used). Given current trends in microcontroller design, this is not unreasonable [ARM98] [Cata99].

We assume all loop iteration counts are bounded, and excessive differences in execution time for conditionally executed code are eliminated through code motion into or out of predicates as well as padding with nops. The CDG of the guest procedure is constructed and processed to separate each node into a series of nodes with at most one real-time event in each. Nodes in the guest procedure containing operations with real-time constraints are marked with a begin time $T_{\text{beg}}$ and the maximum jitter tolerance $T_{\text{tol}}$. The begin time can be derived directly from system requirements, or through some form of a time or task graph [Rama90] [Neih91] [GS94].
2.3. Integration Procedure

After the graph construction, data-flow and timing analysis of the previous section, software thread integration can begin. This section describes first the general approach to STI, and then provides details on code transformation. We have developed efficient transformations for two types of guest events, those within and those not within loops.

STI works by first merging the guest thread with the host thread’s CDG (i.e. degenerate integration) by placing it to the right of the host thread (equivalent to appending at the end of the host thread), then by moving guest thread nodes towards the left and into the host CDG to meet real-time requirements. This process is guided by timing annotations in the CDG. In cases requiring extreme timing accuracy, guest code can be scheduled with single cycle accuracy by moving individual host instructions or adding nops for padding. As the vast majority of program execution is spent in loops, most guest real-time events will need to be executed inside a host loop. STI can transform loops to enable accurate guest node placement for timely execution while allowing trade-offs between execution efficiency and program memory requirements. These transformations also enable efficient integration of guest loops with host loops.

Figure 2.5 illustrates how the two graphs of Figure 2.2 are merged into a single CDG by adding the children (and their subgraphs) of the guest CDG after (to the right of) all other children of the host Proc node. The merged CDG now represents a procedure in which the host operations are first executed followed by the execution of the guest operations.
2.3. Integration Procedure

For this discussion of code motion, Node refers to a generic node, Host to a node from the host thread, Guest to a node from the guest thread, Code to a code node, Proc to a procedure node, Pred to a predicate node, and Loop to a loop node.

An event is called a looping event if it is located within a loop in the guest procedure, otherwise it is a single event. An event can be described by its timing characteristics. It must be executed at its event time $T_{ev}$ with a tolerance of $T_{tol}$. A looping event of loop node Loop, described by $[T_{beg}(Loop), T_{end}(Loop) : T_{iter}(Loop), T_{comp}(Loop)]$ begins execution at time $T_{beg}(Loop)$, and repeats execution every $T_{iter}(Loop)$ until the last iteration ends at $T_{end}(Loop)$. Each iteration requires $T_{comp}(Loop)$ of computation time; the remainder is idle time.

In the final (but fundamental) phase of thread integration, each guest node is processed, beginning with the left-most guest child of the Proc node. If the node and its subgraph contain no real-time events, they are moved left as far as possible in the CDG by reordering the edges from the Proc node. If the node and its subgraph contain a real-time event, the event must be moved to the location or locations which will be executing at the event’s desired time. Events which require accurate timing may need to be inserted in a subgraph to improve scheduling options. This subgraph insertion depends upon its control structure; the recursive algorithm of Figure 2.6 (described next) must be used to move the guest nodes. The CDG presented in Figure 2.5 is used to illustrate each case.

2.3.1. Integration of Single Events

Figure 2.6 shows the code transformations implemented by the recursive algorithm for integration of single events. The first step for integrating a guest event is finding which host CDG node Host will be executing when guest node Guest’s event must occur (at $T_{ev}(Guest)$). To begin the integration process, the root node Proc is chosen as the parent host node Par. Each child of Par is inspected to identify the node Host which minimizes the error time $T_{err}(Host)$.

$$T_{err}(Host, Guest) \equiv |T_{beg}(Host) - T_{ev}(Guest)|$$
2.3. Integration Procedure

a. If the error time is acceptable, i.e. \(|T_{err}(Host, Guest)| \leq T_{tol}\), then node \(Guest\) is moved to execute immediately before \(Host\). For example, guest node \(G2\) and its subgraph must execute within 10 cycles of \(T=20\). There are two gaps in the host which always execute in this range. The guest can be placed immediately before host \(H2\), in which case it will begin executing at \(T=112\) with no jitter. It can also be placed immediately before \(H4\) and begin within the range [13,18]. Both options meet the guest’s timing requirements.

b. If the error time is not acceptable, i.e. \(|T_{err}(Host, Guest)| > T_{tol}\) then node \(Host\) must be entered to find a position in the subgraph with execution time closer to \(T_{ev}\). This traversal depends upon the type of the node \(Host\). This is the case for the guest subgraph rooted by \(G6\), which must begin within 7 cycles of \(T=120\). There is no gap at the top level of the CDG which coincides with this target time. Instead, loop node \(H4\) executes during this time. Moving \(G6\) to before \(H4\) would lead to an error time of 120-13 = 107 cycles, which clearly exceeds the tolerance of 7 cycles.

c. If \(Host\) is a code node \(Code\), it is split into two nodes \(Code_1\) and \(Code_2\) at some point between \(T_{ev-T_{tol}}\) and \(T_{ev+T_{tol}}\). The algorithm is invoked again, with the same parent node \(Par\). In that iteration, \(Guest\) will be moved to between \(Code_1\) and \(Code_2\). For example, if guest \(G2\) had a target time of 5 and a tolerance of 3, host \(H1\) would be split at some point between 5-3 = 2 and 5+3 = 8 cycles to create two code nodes. \(G2\) and its children would then be moved to this new gap.

d. If \(Host\) is a predicate (conditional) node \(Pred\), the algorithm is invoked repeatedly with \(Pred\) as the parent \(Par\) once for each possible value \(V\) of labels on control edges emanating from \(Pred\). If guest \(G2\) had a target time of 14 and a tolerance of 0, it would need to be executed during host \(H2\). Since this is a predicate node, \(G2\) and its children would need to be replicated into each of \(H2\)’s conditions (true and false). The false case is simple; host node \(H3\) would be split at cycle 14 and \(G2\)’s subgraph is copied and inserted into the gap. However, the true case is more complex, as no code exists for this condition. A padding code node of nops would be created and added as a true child of \(H2\). \(G2\) and its subgraph would be copied and added as a true child of \(H2\) after the padding node. The original \(G2\) subgraph is deleted after these transformations. The padding node is sized to ensure the guest code executes on time.
2.3. Integration Procedure

d. If \( \text{Host} \) is a loop node \( \text{Loop} \) iterating from \([1..N]\), it can be split (option 1) into two loops \( \text{Loop}_B \) and \( \text{Loop}_E \) iterating from \([1..n]\) and \([n+1..N]\).

\[
n = \text{round} \left( \frac{T_{ev} \setminus T_{beg} (\text{Loop})}{T_{iter} (\text{Loop})} \right)
\]

![Diagram showing integration procedure](image)

**Fig. 2.6.** Summary of single event integration
2.3. Integration Procedure

If $T_{err}(\text{Loop}_E) > T_{tol}$, one iteration is peeled from the beginning of $\text{Loop}_E$ to form a central node $\text{Host}_C$ which contains the loop body. The algorithm is invoked again with the same parent node Par. In that iteration, Guest will be inserted into $\text{Host}_C$.

As stated before, guest $G6$ and its subgraph must begin at $[120\pm7]$. This occurs during host loop $H4$, which starts between cycles 13 and 18 and performs 10 iterations lasting 50 cycles each. $G6$’s target time occurs during the third iteration (ceiling((120-13)/50) = 2), so the loop can be split into two. The loop subgraph is duplicated, and the two versions are renamed to $H4B$ and $H4E$. $H4B$ performs iterations 0 and 1, while $H4E$ performs iterations 2 through 10. The gap between the two loops occurs at $T_{beg}(H4E) + 2\times50 = [113,118]$. This fits within the target time of $[120\pm7] = [113,127]$, so guest $G6$ and its subgraph can be placed between $H4B$ and $H4E$ with an acceptable timing error.

An alternative (option 2) is to insert the guest node guarded by a predicate node. The predicate node activates the guest when a specific loop iteration is reached.

As calculated above, the $G6$ subgraph begin early in the third iteration ($i = 2$) of host loop $H4$. We examine the timing behavior of the loop body during that iteration. The target time range of $[120\pm7]$ reaches from before $H5$ (with a beginning time of $[13,18] + i\times50 = [113,118]$) to before $H6$ (which begins at $[21,26] + i\times50 = [121,126]$). Either option will create code with acceptable timing, but we select the second case because the timing error is lower. We add a code node $GC1$ to compare the loop iteration counter with an iteration trigger value of 2 at run-time. This comparison result is tested by the guard predicate in $GP1$. If it is true, the guest code below is executed.

Both of these techniques increase code size. Loop splitting does not directly incur a run-time performance penalty, while guarding the guest does. Since this overhead grows more significant as the loop body size decreases, shorter loops should use splitting while longer loops should use guarded execution.

After performing any code insertion, the resulting program must be rescheduled from the insertion point onward and the beginning execution time predictions updated accordingly.

2.3.2. Integration of Looping Events

This section describes how looping guest events are integrated. It presents methods for integrating a guest loop with non-looping host code as well as looping host code. The methods for integrating loops are presented in detail with examples.
Integrating looping events requires extending the algorithm presented above to allow transformation of the looping event in \( \text{Loop}_G \) to match the structure of the host procedure. Depending on host structure, looping guest events are handled in one of two ways: loop peeling or fusion. Loop peeling removes a guest loop iteration and spreads it across non-looping host code. Loop fusion merges the host and guest loops to create one loop. The issue in fusion is creating a loop which meets real-time requirements while executing efficiently.

Looping event integration requires determining how many iterations of the looping event need to be executed within the node \( \text{Host} \), which executes from \( T_{\text{beg}}(\text{Host}) \) to \( T_{\text{end}}(\text{Host}) \). If \( \text{Host} \) is not a loop, iterations from \( \text{Loop}_G \) are peeled off, one at a time, and inserted into \( \text{Host} \) using single event integration. If \( \text{Host} \) is a loop \( \text{Loop}_H \), then it must be combined with \( \text{Loop}_G \) taking into account the different iteration counts and times for each loop. The goal of this loop integration is to fill the idle time in the guest loop with useful work from the host loop. If the idle time is shorter than the host loop body, the guest must be unrolled to increase the idle time. If the idle time is longer than the host loop body, it may be possible to unroll the additional host loop iterations to reclaim more idle time. These two methods of using idle time are the key to efficient loop integration and are presented in detail in this section.

The number of iterations for both loops (\( I(\text{Loop}_H), I(\text{Loop}_G) \)) need not be known at integration time. There may be more than one real-time event \( \text{Guest}_i \) in the guest loop body \( \text{Loop}_G \). To integrate two loops, we convert them to top-test form, copy and integrate their bodies and adjust the loop control tests. The integrated loop performs as many iterations as possible, while the original guest and then host loops (placed after the integrated loop) perform any remaining work. Figure 2.7 presents the code transformations graphically, with sections a and b described below.

The loop control tests for the host must also be integrated. A node \( \text{Code}_{\text{int. test}} \) is created to find the logical product of the two tests. This product is then used by the loop control test predicate node \( \text{Pred}_{\text{int. test}} \) to enable iterations. This form of loop integration with unrolling supports variable iteration counts if the unrolled loop tests a counter which can determine whether at least \( n \) (the degree of unrolling) more iterations remain. Otherwise each unrolled body must be guarded with a predicate, and timing analysis is further complicated.
If there is execution time jitter in the integrated loop body it must be padded to make the guest timing error acceptable for all loop iterations. This means the jitter per iteration of
the integrated loop must be no larger than the smallest guest timing tolerance divided by the number of integrated loop iterations, as shown in the following equation.

\[ T_{jitter}(T_{\text{iter}}(\text{Loop} \text{init})) \leq \frac{T_{\text{tot}}(\text{Guest})}{I(\text{Loop} \text{init})} \forall \text{Guest} \]

Integrating two loops involves fusing the two bodies together and adjusting the execution time per iteration to be an integer multiple of the looping guest event’s period. If the guest’s idle time is longer than a host iteration, the two loops can simply be fused without any unrolling, using nop padding to fill in remaining idle time. This approach becomes very inefficient as the iteration period mismatch grows. Loop unrolling enables the CPU to perform useful work in place of the busy waiting.

Either the host or guest loop may need to be unrolled, depending upon the iteration time relationship. If the guest must be executed more frequently, it is copied and inserted into the host. Otherwise the guest may have sufficient idle time to allow execution of more than one host iteration, allowing greater efficiency.

A loop’s iteration time must be broken down into two components to identify the amount of time \( T_{\text{iter body}} \) used for the loop body computations and the time \( T_{\text{iter test}} \) used to test whether to execute another iteration.

\[ T_{\text{iter}}(\text{Loop}) = T_{\text{iter body}}(\text{Loop}) + T_{\text{iter test}}(\text{Loop}) \]

\[ T_{\text{iter test}}(\text{Loop}) = T_{\text{comp}}(\text{Code test}) + T_{\text{comp}}(\text{Pred test}) \]

The test time accounts for the time needed for instructions which perform comparisons \( (T_{\text{comp}}(\text{Code test})) \) and the time for the conditional branch in the predicate \( (T_{\text{comp}}(\text{Pred test})) \).

a. If the idle time per guest loop iteration is shorter than a host loop iteration \( (T_{\text{iter}}(\text{Loop}_G) - T_{\text{comp}}(\text{Loop}_G) < T_{\text{iter}}(\text{Loop}_H)) \) the guest loop body must be executed more than once per host loop iteration. The guest loop is unrolled to increase the idle time so it is large enough to hold an entire host loop iteration. The new loop contains \( n \) guest loop bodies.
2.3. Integration Procedure

This takes into account the additional computation time added by each unrolled body of Loop\(_G\). The unrolled guest loop is fused with the host loop body. The resulting loop body is padded with nop instructions lasting \(T_{pad}\) to bring the execution time per loop iteration to \(n \cdot T_{iter}(\text{Loop}_G)\), taking into account the host loop body time, the guest loop computations, and the loop iteration control test instructions.

\[
n = \left\lfloor \frac{T_{iter} \cdot \text{body}(\text{Loop}_H) + T_{iter} \cdot \text{test}(\text{Loop}_i)}{T_{iter}(\text{Loop}_G) \angle T_{comp}(\text{Loop}_G)} \right\rfloor
\]

For example, Figure 2.8 shows a host loop with an iteration body time of 28 to 30 cycles and a guest loop which performs four cycles of computation every 10 cycles (starting at cycles 4, 14, 24, etc.). The guest loop must be unrolled to accommodate the worst-case execution time of a host loop iteration (body and control, 34 cycles). First, the host loop body is padded (\(H3FPad\)) to regularize its timing, extending its iteration body duration to its worst case of 30 cycles. The guest loop is then unrolled ceiling\((34/(10-6)) = 6\) times and is integrated to create the CDG shown. The two loop control tests are merged into a node \(\text{Code}_\text{test}\) which creates a data value for testing by the node \(\text{Pred}_\text{test}\). Two cycles of padding (in \(HGPad\)) are needed to compensate for the loop body duration mismatch. The closer the host loop duration is to a multiple of the guest loop idle time, the less padding will be needed.

b. If the idle time per guest iteration is longer than a host loop iteration \((T_{iter}(\text{Loop}_G) - T_{comp}(\text{Loop}_G) > T_{iter}(\text{Loop}_H))\), then more than one host loop iteration can occur within this idle time. This can be achieved in two ways. The host loop can be unrolled, or execution of the guest can be guarded with a predicate. The first method emphasizes execution efficiency at the expense of more code memory, while the second does the opposite and becomes more attractive for larger host loop bodies.

b1. In the first method, the host loop is unrolled to contain \(n\) host loop bodies and is fused with the guest loop body. Guest nodes are transformed according to their timing re-
2.3. Integration Procedure

quirements and the local host CDG structure. The loop control test and predicate are not included in the loop unroll time calculations, as they are not duplicated when unrolling.

\[
n = \left\lceil \frac{T_{iter}(\text{Loop}_G) \angle T_{comp}(\text{Loop}_G) \angle T_{iter \cdot test}(\text{Loop}_{int})}{T_{iter}(\text{Loop}_H)} \right\rceil
\]

**Timing Characteristics**

- \( T_{ev(G_2)} = 4 + 10i \)
- \( T_{col(G_2)} = 0 \)
- \( T_{err(G_1)} = 10 \)
- \( T_{comp(G_1)} = 4 \)
- \( T_{iter\ body(H_1)} = [28,30] \)

**Unrolling Factor**

\[
n = \left\lceil \frac{T_{iterbody}(\text{Loop}_H) + T_{iter\ test}(\text{Loop}_{int})}{T_{iter}(\text{Loop}_G) \angle T_{comp}(\text{Loop}_G)} \right\rceil = \left\lceil \frac{30 + 4}{10 \angle 4} \right\rceil = \left\lceil \frac{34}{6} \right\rceil = 6
\]

**Padding**

\[
T_{pad} = n \cdot T_{iter}(\text{Loop}_G) \angle T_{iterbody}(\text{Loop}_H) \angle \left( n \cdot \sum_{i} T_{comp}(\text{Guest}_i) \right) \angle T_{iter\ test}(\text{Loop}_{int}) \]

\[
T_{pad} = (6 \cdot 10 \angle 30 \angle (6 \cdot 4) \angle 4) = 2
\]

The host loop body is padded with nops lasting \( T_{pad} \). This padding takes into account the computation time of the host loop iterations, the guest nodes and the loop control instructions.
Figure 2.9 shows an example in which host loop $H_1$ (which lasts 28 to 30 cycles per iteration body) is unrolled twice to fill most of the 76 cycles of idle time in the guest loop $G_1$. The unrolling factor $n$ is floor$(71/30) = 2$. The host predicate nodes $H_3$ and $H_3'$ are padded to fully eliminate timing jitter. The loop tests are combined to form the node Code$_{test}$ and the integrated loop body is then padded (with $HGPad$) to bring the total iteration time up to $T_{iter}(Loop_G)$, which is 91 cycles.

b2. The second method uses a predicate node in conjunction with a counter variable to ensure each guest node is executed once every $n$ host loop iterations. Instructions to initialize the guard counter are inserted before the host loop initialization code. A guard
2.3. Integration Procedure

Predicate $Pred_{Guard_j}$ is added to the host loop body for each guest $Guest_i$ or contiguous group of guests in the guest loop body. A guard predicate may control execution of one or more guest nodes if there is no idle time between them. As a result, there may be fewer guard predicates $Pred_{Guard_j}$ than guest nodes $Guest_i$. A code $Code_{Guard_j}$ node is added at the beginning of the host integrated loop to set a flag to enable guest execution based on integrated loop iteration number. This node may need to be repeated if register pressure keeps the flag from being kept in a register until the last guard predicate. A code node $Code_{Guard\_inc}$ to increment the guard counter may be added to the host loop body if there is no such loop counter already. This node may detect target iterations by comparing for an explicit value, performing a modulo operation, or maintaining its own counter variable which is reset upon each guest execution.

The time per iteration for the new host loop is calculated as the sum of the original host loop body, all the guard code and predicate nodes, any code for incrementing the guard variable, the code for testing the loop condition, and the loop closing branch in the predicate.

$$
T_{iter}(Loop_{int}) = T_{iter}(Loop_H) 
+ \sum_{j} (T_{comp}(Code_{Guard_j}) + T_{comp}(Pred_{Guard_j}))
+ T_{comp}(Code_{Guard\_inc}) + T_{comp}(Code_{test}) + T_{comp}(Pred_{test})
$$

The period of the guard counter is $n$ host loop iterations; the counter ranges from 0 to $n-1$.

$$
n = \left\lceil \frac{T_{iter}(Loop_G) \angle T_{comp}(Loop_G)}{T_{iter}(Loop_{int})} \right\rceil
$$

The delay $I_{del}(Guest_j)$ is measured in host loop iterations from the beginning of the new host loop’s first iteration to the guest event $Guest_i$, taking into account the execution of any previous guests $Guest_j$ in this iteration of this guest loop body.
These delays determine the position of the guard predicates (and hence guest nodes) within the new host body, as well as the values for which the guard predicates allow the guest to be executed. The predicate for Guest\(_i\) is set to match the proper host loop iteration counter value and is placed in the correct location of the loop body by calling \textit{Integrate}().

Non-real-time guests are moved as far left (early) as possible. Each guard predicate controls execution of one or more consecutive guest instructions. There may be multiple real-time guest nodes guarded by one predicate if no idle time separates them in the original guest thread.

The host loop body is padded with nops lasting \(T_{pad}\). This padding takes into account the computation time of the host and guest nodes and the time to reset the guard counter variable, if one is used. The padding is added below the last guest guard predicate.

\[
T_{pad} = T_{iter}(Guest) \angle (n \cdot T_{iter}(Loop_{int})) \angle \sum_{\forall i} T_{comp}(Guest_i) \angle T_{comp}(Code_{Greset})
\]

Figure 2.10 shows the example of Figure 2.9 implemented with guest guarding rather than unrolling. The reduction in code size explosion is immediately obvious.

Integration begins by finding the iteration time of the integrated loop body without the guest executing, which is the sum of the original host loop body duration, the guard code and predicate nodes, the guard variable increment (if needed) and the loop control code and test predicate nodes. The idle time per guest iteration is divided by this iteration time to determine that two iterations fit. The iteration delay \(I_{del}\) is next calculated to find that the first guest event (with a target time of 10±0) iterates in the first iteration (floor(10/36) = 0) of the integrated loop body, so its guard test should trigger on 2i+0. The guard predi-
cate is inserted four cycles into the loop body to ensure the real-time guest node \( G2 \) at \( t = 10 \pm 0 \) into the iteration. Finally, \( T_{pad} \) is evaluated and found to be two, and a two cycle padding node \( \text{CodeHGPad} \) is inserted following the guest \( G2 \). Remaining iterations of either loop are handled by discrete loops after the integrated loop.

### Timing Characteristics

<table>
<thead>
<tr>
<th>Timing Characteristics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{ev}(G2) )</td>
<td>10+91t</td>
</tr>
<tr>
<td>( T_{dcl}(G2) )</td>
<td>0</td>
</tr>
<tr>
<td>( T_{iter}(G1) )</td>
<td>91</td>
</tr>
<tr>
<td>( T_{iter\ body}(H1) )</td>
<td>[28,30]</td>
</tr>
</tbody>
</table>

### Integrated Loop Iteration Time

\[
T_{iter\ (Loop\ int)} = T_{iter\ (Loop\ H)} + \sum_{ij} (T_{comp}(\text{Code}_{Guardj}) + T_{comp}(\text{Pred}_{Guardj})) + T_{comp}(\text{Code}_{Guardinc}) + T_{comp}(\text{Code}_{test}) + T_{comp}(\text{Pred}_{test})
\]

\[
T_{iter\ (Loop\ int)} = 30 + 2 + 0 + 3 + 1 = 36
\]

### Unrolling Factor

\[
n = \left\lfloor \frac{T_{iter\ (Loop\ G)} \times T_{comp\ (Loop\ G)}}{T_{iter\ (Loop\ int)}} \right\rfloor
\]

\[
n = \left\lfloor \frac{91 \times 15}{36} \right\rfloor = 2
\]

### Padding

\[
T_{pad} = T_{iter\ (Guest)} \times (n \times T_{iter\ (Loop\ int)}) - \sum_{i} T_{comp}(\text{Guest}_{i}) \times T_{comp}(\text{Code}_{Guest_{i}})
\]

\[
T_{pad} = 91 \times (2 \times 36) - 15 \times 0 = 4
\]

**Fig. 2.10.** Guest guarding integration example

The loop control tests for the integrated loop are formed as the logical product of the tests of both host and guest loops. Unrolling a loop requires changing its counter variable target value to match \( n-I \) iterations earlier. Any remaining iterations are performed by the original loops, located after the integrated loop. The guest loop is padded and placed immediately after the integrated loop to allow satisfaction of real-time requirements.

### 2.4. Experimental Results

This subsection presents three examples to illustrate HSM via STI and the potential benefits. It demonstrates the hardware reduction and analyzes the performance of the resulting system. Three hypothetical applications have been processed to implement real-time func-
2.4. Experimental Results

2.4.1. Handheld Computer

The first application is a small portable computing device such as a hand-held PC (HPC) or portable video game with a high resolution graphic liquid crystal display (LCD). Current HPCs use CPUs with performance of nearly 100 MIPS [Phil98], and future devices will grow faster. Thread integration uses part of this growing CPU capacity to refresh the LCD, eliminating the need for a dedicated LCD controller and its local frame buffer memory. A line drawing routine is integrated with the LCD row refresh function, improving system efficiency. Software thread integration has promise for these markets as it eliminates hardware, cutting system size, cost, weight and time-to-market.

Figure 2.11 shows the original system hardware architecture. The CPU communicates with an LCD controller (LCDC) [Hita2], which generates control and data signals for the LCD based upon data stored in the frame buffer.

![Handheld computer hardware components](image)

[Fig. 2.11. Handheld computer hardware components]

A high resolution monochrome LCD (640 by 480 pixels, 1 bit per pixel) displays information and must be refreshed 70 times each second to avoid flickering. Column pixel data is loaded serially into a shift register and then latched every 59.5 \(\mu s\), driving an entire row simultaneously.

The data and control signals are generated by dedicated LCD controller which requires its own memory or else arbitrated access to the CPU’s memory. Using this dedicated hardware solution increases chip count, size, weight and power, which are typically at a premium in this type of device. Some microcontroller makers address this problem by integrating the LCD controller with the microcontroller. The main disadvantages of such
hardware integration are that it limits the designer’s options in selecting a microcontroller and increases device cost.

It is possible to generate the LCD control signals in software. A periodic interrupt every 59.5 $\mu$s calls a function which shifts a row of data out and clocks the shift registers. The primary bottleneck of this scheme is the low maximum clock speed for common LCD driver shift registers, which ranges from 4 to 12 MHz. As seen in Figure 2.12, this bottleneck forces a 100 MIPS CPU to spend nearly half of its time during LCD refresh waiting for the shift register, in the form of one and two nop busy waits. Every 59.5 $\mu$s, the CPU spends 1280 of its 2984 LCD refresh cycles as busy waits. As a result, the 100 MIPS CPU has only 50 MIPS remaining for applications, as 29 MIPS are used for the display refresh and 21 MIPS are used for busy waits. Thread integration enables the CPU to use those wasted cycles to perform useful work. In this example a fast line drawing routine [Bres65] is integrated with the LCD refresh thread to take advantage of the free time.

Figure 2.13 shows remaining CPU capacity as a function of line-drawing activity; refreshing the LCD in software requires 50% of the processor’s time. The horizontal axis measures the amount of line drawing work presented to the system. It is measured in line length (in pixels) per LCD row refresh period (59.5 $\mu$s). The vertical axis measures the CPU capacity remaining after refreshing the display and drawing the lines.
2.4. Experimental Results

The plot shows capacity for three design options. The first uses a dedicated hardware LCD controller to refresh the display, the second uses an integrated software solution, and the third uses a traditional, discrete software solution with busy-waiting. The hardware solution obviously places no load on the CPU for refreshing the LCD, so it has the highest CPU time remaining. Moving the refresh to software immediately imposes a 50% load on the CPU, as shown by the two software option plots. As the line drawing work rises from 0 pixels per refresh row, however, the integrated version reclaims the idle time for line drawing until it is all used (at 40 pixels). At this point there is no more idle time to reclaim and the CPU capacity starts to fall. By comparison, the discrete busy-waiting software refresh solution never reclaims any idle cycles, so CPU capacity begins to fall immediately as line drawing work increases from 0 pixels per refresh row.

The integrated software solution uses the idle time in the refresh function to plot up to 40 pixels per display refresh row, freeing up to 21% of the CPU’s capacity for other functions. Table 2.1 presents information on the discrete and integrated threads.

Thread integration allows elimination of the LCD controller and its frame buffer. The cost for an integrated software solution is 2412 bytes of program memory (1580 more than the discrete software version), 38,400 bytes of data memory for the new frame buffer and 29 MIPS of processor throughput. Thread integration uses nearly all of the idle cycles in
2.4. Experimental Results

LCDRow to perform useful work, and mitigates about half of the performance impact of implementing LCD refresh in software.

### Table 2.1: Handheld Computer Thread Statistics

<table>
<thead>
<tr>
<th>Thread</th>
<th>Cycle Count</th>
<th>Idle Cycle Count</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrawLine (40 pixels long)</td>
<td>1390</td>
<td>0</td>
<td>616</td>
</tr>
<tr>
<td>LCDRow</td>
<td>2984</td>
<td>1280</td>
<td>336</td>
</tr>
<tr>
<td>Discrete DrawLine (40 pixels) and LCDRow</td>
<td>4374</td>
<td>1280</td>
<td>952</td>
</tr>
<tr>
<td>Integrated DrawLine (40 pixels) and LCDRow</td>
<td>3063</td>
<td>0</td>
<td>2412</td>
</tr>
</tbody>
</table>

2.4.2. Vehicle Cellphone with External Network

The two digital cellphone examples use GSM 06.10 lossy speech compression [Dege94] integrated with a communication protocol. These applications have tight cost, size, weight and power constraints yet benefit from protocol inclusion. Thread integration is used to eliminate network interface hardware by performing such functions efficiently in software. Both examples integrate a message transmission function into a GSM function which is called once per 20 ms frame; this introduces a message transmission delay of up to 20 ms, which is acceptable for many applications. Message reception is asynchronous and is not integrated; instead a discrete interrupt service routine is used.

![Vehicle cellphone hardware components](image)

**Fig. 2.14.** Vehicle cellphone hardware components

The first example features a cellphone embedded into a vehicle. With its CAN interface, the phone can signal and react to events in the vehicle, such as muting the stereo during phone calls and automatically calling emergency service dispatchers upon airbag deployment. As the automotive application domain is very price-sensitive, the CPU speed is cho-
2.4. Experimental Results

SEN to be 33 MHz for a 72% load from speech compression. Figure 2.14 shows a block diagram of the phone’s digital architecture for the two network implementations.

CAN 2.0A [Bosc91] is a robust multimaster bus designed for real-time communication with short messages (up to eight bytes). Transmitters perform bitwise arbitration on unique 11 bit message identifiers to gain access to the bus. During message transmission, the sending node monitors the bus for errors. At the end of the message frame (up to 131 bits), all nodes on the bus assert an acknowledgment flag. Figure 2.15 shows the timing of operations within each bit cell for this application when using a 33 MHz CPU and 500 kbps CAN bus. The CAN code requires only 29 cycles of work per 66 cycle bit time, so the CPU utilization of a discrete, busy-wait version is only 44% during message transmission. This CAN message transmission code has been optimized to reduce cycle count; it is assumed that the bit pattern is ready for transmission, with bit stuffing and cyclic redundancy check codes precomputed. The CAN implementation of HSCAN in Chapter 6 is more general and performs these operations at transmission or reception time, resulting in more instructions needed per bit time.

Integrating the CAN function with a GSM function (Reflection_Coefficients) allows the CPU to reclaim some of these idle cycles. Figure 2.16 presents the CPU capacity remaining after message transmission and GSM compression.

The horizontal axis presents the transmit message load presented to the CPU. The vertical axis shows the impact on CPU time remaining after CAN message transmission and the GSM function (Reflection_Coefficients) perform their work. A hardware CAN implementation places nearly no load on the CPU at these message rates. However, the software implementations do because of the protocol processing and idle time. The integrated software version improves upon the discrete version by reclaiming some idle time. Its performance depends upon the message load, as with the LCD refresh in Figure 2.13. Below 50
messages transmitted per second, the integrated version is able to reclaim idle time during message processing and hence increase efficiency. However, only 19% of the available idle time is reclaimed due to the limited integration performed, so the slope is not horizontal (as it would be if integration were 100% efficient). More extensive integration would increase the efficiency. The LCD refresh idle time was reclaimed with nearly perfect efficiency, so the corresponding line segment (yellow) in Figure 2.13 is nearly horizontal.

Beyond 50 messages per second a dedicated software CAN transmit function must be called, as the host function is only called 50 times per second. This dedicated function is the one used for the discrete software solution, so its reduces the CPU capacity accordingly. Additional idle time can be reclaimed by integrating the CAN transmit thread into additional hosts. This will add additional knees and push the last knee in the plot farther to the right, improving the benefits from integration. If this integration recovers idle time more efficiently than this current example, the slope before the last knee will become more horizontal.

Thread integration replaces 19% of the idle cycles with useful work. As summarized in Table 2.2, an additional 2062 bytes of program memory are needed for the integrated version as compared with the discrete. Thread integration enables system designers to eliminate a dedicated CAN protocol controller chip, reducing system size, weight and cost.

![Fig. 2.16. Vehicle cellphone CPU capacity vs. message activity](image-url)
2.4. Experimental Results

Table 2.2: Vehicle Cellphone Thread Statistics

<table>
<thead>
<tr>
<th>Thread</th>
<th>Cycle Count</th>
<th>Idle Cycle Count</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reflection_Coefficients</td>
<td>6344</td>
<td>0</td>
<td>1264</td>
</tr>
<tr>
<td>CAN</td>
<td>8674</td>
<td>4847</td>
<td>498</td>
</tr>
<tr>
<td>Discrete Reflection_Coefficients and CAN</td>
<td>14990</td>
<td>4847</td>
<td>1762</td>
</tr>
<tr>
<td>Integrated Reflection_Coefficients and CAN</td>
<td>14436</td>
<td>0</td>
<td>3824</td>
</tr>
</tbody>
</table>

2.4.3. Cellphone with Internal Network

The second cellphone example, a handheld device, communicates with its smart battery using the $I^2C$ protocol, a 100 kbps multimaster bus popular for communication within small embedded devices [Phil95a]. The message transmission function ($I^2C$) is integrated with an autocorrelation function (Fast_Autocorrelation). $I^2C$ implements a subset of the protocol, being limited to sending one byte messages (called Quick Commands in the SMI-Bus extension to $I^2C$) in a system with only one master and regular speed peripherals. The CPU runs at 66 MHz; voice compression requires 30% of the CPU’s cycles. The remaining capacity might be used for advanced features such as voice recognition, soft modem/fax, image compression/decompression, and encryption.

![Handheld cellphone hardware components](image)

Figure 2.17 shows the two hardware architectures which support the hardware and software implementations of $I^2C$. The hardware $I^2C$ version contains a dedicated bus controller, while the software version reduces system hardware.
Table 2.3 summarizes characteristics of the two software implementations while Figure 2.18 plots CPU time required for $I^2C$ message transmission based on message rate. The horizontal axis shows the message load per second, while the vertical axis presents CPU time remaining after performing host and guest work. The slope of the integrated implementation shows that integration efficiency is less than perfect, but better than the previous example (Figure 2.16). The host function Fast_Autocorrelation is called only 50 times per second, so beyond this frequency the dedicated (and inefficient) busy-wait $I^2C$ function is used, and the CPU time remaining falls accordingly.

<table>
<thead>
<tr>
<th>Thread</th>
<th>Cycle Count</th>
<th>Idle Cycle Count</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast_Autocorrelation</td>
<td>25768</td>
<td>0</td>
<td>268</td>
</tr>
<tr>
<td>$I^2C$</td>
<td>6612</td>
<td>6404</td>
<td>280</td>
</tr>
<tr>
<td>Discrete Fast_Autocorrelation and $I^2C$</td>
<td>32380</td>
<td>6404</td>
<td>548</td>
</tr>
<tr>
<td>Integrated Fast_Autocorrelation and $I^2C$</td>
<td>27943</td>
<td>0</td>
<td>1416</td>
</tr>
</tbody>
</table>

Fig. 2.18. Handheld cellphone CPU time vs. message activity
The discrete software message transmission function presents a small load for the high-performance CPU (chosen to support the advanced features mentioned previously), but it is reduced further through thread integration. The integrated version supports rates of up to 50 messages per second and is limited by the call frequency of its host function Fast_Autocorrelation. At higher message rates the surplus messages are transmitted by the less efficient discrete I2C function, which is beyond the knee in the plot.

Integration improves the run time efficiency of the discrete functions but expands the code memory requirement for the two functions from 548 to 1416 bytes. This integration fills the idle time with code from Fast_Autocorrelation efficiently enough to mask 67% of the I2C message transmission time.

In this example, thread integration allows system designers to eliminate a dedicated I2C controller or increase the efficiency of a software implementation at the price of slightly more program memory.

2.4.4. Summary of Experiments

Table 2.4 summarizes the cost of implementing each application’s real-time function in software. The discrete, or non-integrated, software implementation (baseline) in each case uses busy waiting to meet guest function timing requirements, while thread integration increases the software implementation efficiency, reducing the run time at the expense of increased code memory.

The “Run Time Change” columns show how much time is needed to perform the guest’s instructions compared with a busy-wait guest implementation. An integrated implementation overlaps host and guest execution and so reduces the time difference below 100%. For example, adding software LCD refresh support to the handheld computer using a discrete busy-wait version requires 2984 cycles per display row. Integrating the LCD refresh function with a graphics line drawing routine reduces the refresh cost to 1673 cycles per display row; now 45% of the refresh work is performed during line drawing. The “Code Memory Change” columns show how much code memory is needed to add the guest function. Thread integration increases code size (e.g., by splitting or unrolling loops), resulting in significant code expansion. Note that typically only one or two functions will need to be integrated, so the 2x to 6x memory increase is only incurred for the code of those functions, and hence is minor. For example, the handheld cellphone’s 1148 byte increase is small in comparison with the 60 kilobytes needed for the voice codec functions.
Table 2.4: Costs of Implementing Guest Functions in Software, without and with STI

<table>
<thead>
<tr>
<th>System</th>
<th>Discrete (without STI)</th>
<th>Integrated (with STI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run Time Change</td>
<td>Code Memory Change</td>
</tr>
<tr>
<td>Handheld Computer</td>
<td>+100% of Guest</td>
<td>+336 bytes</td>
</tr>
<tr>
<td>Vehicle Cellphone</td>
<td>+100%</td>
<td>+498</td>
</tr>
<tr>
<td>Handheld Cellphone</td>
<td>+100%</td>
<td>+280</td>
</tr>
</tbody>
</table>

2.5. Conclusions

The methods presented in this chapter enable the automatic integration of real-time guest events from a guest thread with a host thread, optimizing for execution speed or program size. When combined with the thread selection and triggering support presented in Chapter 4, these methods enable the straightforward migration of real-time functions from special-purpose peripheral hardware to software running on a general-purpose microprocessor. This chapter illustrates the feasibility and benefits of STI by replacing peripheral integrated circuits in three different embedded applications, reducing system size, cost and weight.
2.5. Conclusions
Chapter 3

*Thrint*: A Tool for Automatic STI

This chapter describes our tool (*Thrint*) for automatic software thread integration. First we present background on why there are so few tools available to embedded system designers with support for real-time issues in general and hardware to software migration in particular. Second we describe *Thrint*, our software tool which performs software transformations driven by static timing analysis to implement hardware to software migration using software thread integration. We demonstrate its use and describe how to extend its usefulness.

3.1. Real-Time Embedded System Development Tools

Software development efficiency has improved dramatically in recent decades, yet the technologies responsible for the gains have had much less effect in the real-time embedded systems community for various reasons.

“Sharp” tools simplify software development [Broo95] by allowing programmers to develop code at a high level of abstraction. This management of complexity enables larger systems to be created and maintained more efficiently. High-level languages free the developer from having to consider technicalities such as register use and memory allocation and instead focus on larger issues. However, many of these technical details are critical in real-time embedded systems (RTESs), where resources may be limited and performance critical.

As one RTES pundit has stated, “Today’s state-of-the-art of real-time development is appalling.” [Gans99] This reflects various characteristics of the research community and industry:

- It is difficult to bring software from research lab to market because of its intrinsic complexity. Compilers are especially complex because of their need to maintain
3.1. Real-Time Embedded System Development Tools

program correctness. Hardware, by comparison, is designed with a well-structured methodology and a toolset to manage complexity as designs scale up.

• The embedded system market is extremely fragmented, with dozens of processor architectures and hundreds of variants available. This reflects the abundance of high-volume applications which are cost-sensitive enough to mandate a tight fit between microcontroller (MCU) features and application requirements. This heterogeneity magnifies the tool development and maintenance effort needed since so many architectures must be supported.

• “Color of money” management issues can limit the budget for software and hardware development tools if they are classified as capital items and hence must be paid for with a separate and potentially smaller budget.

• The current state of RTES development does not reveal system timing behavior until late in the process, after most of the software (and perhaps hardware as well) are made. At this point three pitfalls join to form a larger, more deadly trap. First, the “engineers are free” syndrome hinders the purchase of tools which might simplify development [Koop96]. Second, the difficulty of seeing the bugs beyond the current one weakens the argument for buying more tools. Finally, the dangerous flexibility and fragility which come hand-in-hand with assembly code programming seduce the developer, leading him onto the treacherous ground of hacking assembly code.

• There is significant academic research interest in real-time issues such as schedulability and static timing analysis. However, most of this work has been confined to applications for high-end embedded systems -- those which require the processing power of the 32- or 64-bit CPUs running the ubiquitous workstation. This tends to ignore issues relevant to applications running on 8- and 16-bit CPUs, which make up 95% of the CPU market by revenue (their market share by volume is essentially 100%) [Levy99]. Interest in the smaller processors has gradually grown in the research community, but much remains to be done to establish an underlying theoretical foundation for RTESs.

In the embedded system tool industry there is currently minor and fragmented support for performing ad hoc HSM. At a higher level there are schedulability tools to determine if a given set of tasks can meet its deadlines. At a lower level there are simulators which allow prediction of program timing along a single path, though more sophisticated tools are being investigated by industry [Engb99]. Between these two levels there is very little avail-
3.2. Tool Chain Overview

The research from the academic community on static timing analysis is gradually entering commercial tools, but there is a call for much more [Gans99]. In part this lack of tools comes from the vaguely-defined ad hoc methods for HSM, which our research attempts to clarify.

In this chapter we describe Thrint, our software tool which performs software transformations driven by static timing analysis to implement hardware to software migration using software thread integration. The STI experiments which are described in the following chapters are performed with a combination of manual and automatic operations, as the emphasis of our research has been first to develop STI methods and second to implement them in a tool capable of automated operation. The examples presented in this chapter were all performed automatically by Thrint.

3.2. Tool Chain Overview

Thrint fits into a code compilation and visualization tool chain as shown in Figure 3.1. The user’s source code (foo.c) is compiled with GCC to assembly code (foo.s) for preprocessing by PCOM, which structures the program into a CDG (control-dependence graph) and performs some loop analysis to create a structured assembly code file (foo.pdg.txt). The program is also compiled with profiling enabled (-pg) and executed to create an execution profile (foo.gmon.out) for analysis by GProf. The output of this tool (foo.gprof) is examined later to determine which threads to integrate and to perform system performance predictions. The structured assembly file from PCOM is then analyzed and transformed by Thrint to create an integrated output assembly file (foo.int.s). Thrint can produce additional information for program visualization. It can create graph description files (foo.vcg) for program visualization with the graphing program XVCG as well as scatter plots describing timing jitter and delay in deterministic segments (foo.j.gpc and foo.s.gpc) using the plotting program GnuPlot. The integrated output assembly code (foo.int.s) is compiled and linked by GCC to create an integrated executable file (foo) which is debugged with GDB or DDD.
3.3. Operation of Thrint

Figure 3.2 presents an overview of the processing stages in Thrint. Thrint reads in the CDG-structured Alpha assembly code created by PCOM and creates an internal CDG representation of the program. This CDG can be visualized by creating a vcg file (-G command line option) for display with the XVCG program.
3.3. Operation of Thrint

Figure 3.3 shows an example of the output. Individual instructions can be included in the vcg output to aid program understanding.

The integration directives file (foo.id) is then read in. It contains commands which direct specific integration actions by specifying when each guest block of code must execute, with a given tolerance. Alternatively, a specific host node may be stated. All guest nodes must be declared in this file, as all other nodes are assumed to be host nodes. Figure 3.4 shows an example .id file which specifies that guest nodes Guest1, Guest2 and Guest3 are to be executed at clock cycle T = 10, 35 and 45 respectively, with no tolerance for timing errors. Although the guest nodes and target times are specified manually at the present time for expediency, we plan to simplify timing specification in the future by adding support for timing directives in the C source code.

Fig. 3.2. Thrint overview
Fig. 3.3. Example of automatically generated CDG program representation
Backwards data-flow (liveness) analysis is performed next on register data. Register use is analyzed to find def-use webs and hence variables. Each variable can be assigned its own virtual register to prepare for coloring after code motion. This variable information can be added to the CDG VCG graph output. Two separate sets of variables are created, one for host variables and one for those of the guest. Host and guest interference graphs are constructed for future use in register coloring. Currently the register set is partitioned to provide six registers for the guest thread. In the future register coloring will be added to support more efficient register use.

Static timing analysis (STA) is performed next to identify the start time and duration for each node in the CDG. We assume scalar instruction execution, no pipeline bubbles, no cache, and single cycle instruction execution. These assumptions are valid for most embedded systems. Significant work exists which extends STA theory to support these features, but that is not the goal of our research. Both best and worst case times are recorded, and indeterminate durations are marked. First, the duration of each node is computed by traversing the CDG in a post-order left-to-right traversal. Second, the start time of each node is computed in an pre-order left-to-right traversal, predicting the execution schedule of the program. The timing information is included in the CDG VCG output to simplify program timing understanding.

The scheduled CDG is next examined for deterministic segments. Segments are grown by concatenating nodes until the total jitter exceeds a specified threshold. These segments are then stored in a list for each procedure. GnuPlot data and driver files can be created to visualize segment and jitter data in a scatter-plot format. For example, Figure 3.5 shows the deterministic segments in the functions of the MPEG decoder examined in Chapter 5. These graphs reveal deterministic segments hidden between highly variable code within the application.
3.3. Operation of Thrint

The horizontal axis shows the host deterministic segment length as it grows, while the vertical axis shows the amount of timing jitter at that point in the segment. As a segment grows, its jitter tends to increase because of conditionals with different path lengths and loops with unknown iteration counts. The vertical axis therefore presents two views of one piece of information: how much timing jitter a segment has if not padded, and also how many cycles of processing will be wasted if the segment is padded to completely eliminate jitter. As a result, jitter is used as a cost function for whether timing or efficiency is more important.

In this plot, which is taken from Chapter 4, we see the jitter for various segments found within an MPEG decoder program. These segments come from 7 functions which account for 90.8% of the program’s execution time. At the bottom of the graph, with almost no jitter, are segments from four functions. Two are quite substantial, lasting about 1000 cycles each. Not shown on this plot is another function (OrderedDitherImage) which has no jitter in a 9.73 million cycle segment. An interesting segment comes from j_rev_dct, which performs a reverse discrete cosine transform. A loop with a fixed iteration count has an iteration duration lasting from 177 to 265 cycles. It contains two large blocks of deterministic code separated by two predicate nodes of variable duration. With each loop iteration the segment grows by 265 cycles and the jitter rises by 88. This appears as a stair-step pattern in Figure 3.5.
When weighted with profiling information, these deterministic segments account for about a third of the program’s total execution time. These concepts and results are presented in depth in Chapter 4.

The variables are reallocated into the partitioned register set at this stage to allow unrestricted code motion. The statically derived timing information may need to be modified slightly to reflect reallocation.

Integration can finally begin after these preparations. First, each deterministic segment in the procedure is examined to identify if it is long enough and deterministic enough to be a host for the guest thread. If a segment meets the requirements, it is annotated with commands describing the code transformations and padding needed to integrate the threads. The cumulative delay of the guest thread nodes is factored in when determining the necessary integration activities. The activities are then analyzed to determine the integration cost in terms of execution cycles and program memory. The “best” thread is then chosen, based on user-specified weighting criteria.

Second, integration is performed using the chosen thread. Guest nodes are replicated from the original (which is deleted after integration) as needed to create copies in the locations specified by the previously determined transformation commands. The following diagrams show various stages in integrating a host and guest according to the directives in Figure 3.4.

- In Figure 3.6, host code node ReconIMBlock..ng is executing during Guest1’s target time; T = 10±0 falls completely within the range 1..13. As a result, the host node is split at T = 10 and the guest is moved into the gap between the two pieces.
- Figure 3.7 shows how Guest2 is moved into the host predicate node ReconIMBlock..ng because the target time of 35±0 falls within the host’s 24 to 36 or 38 execution. This requires copying Guest2 into both the taken (marked with green arcs) and not taken (red arcs) cases.
- Finally, in Figure 3.8 Guest3 has been moved to execute at 45±0. Host code node $37 is split at T = 45 and Guest3 is moved into the new gap.
3.3. Operation of Thrift

Fig. 3.6. Guest1 moved to execute at T=10
Fig. 3.7. Guest2 moved to execute at T=35
Fig. 3.8. Guest3 replicated to execute at T=45, jitter padded out
3.3. Operation of Thrint

The timing jitter of the guest nodes is brought into compliance with their tolerance limits by padding away earlier jitter. Notice how in Figure 3.8 the jitter of 2 cycles created by the variable duration of predicate node ReconIMBlock..ng is reduced to 0 cycles by the automatic addition of a new padding code node (colored gray in the diagram) called ReconIMBlock..ng_pad containing 2 nops.

In the following diagrams we see integration for various cases derived from Figure 3.3 (with code node ReconIMBlock..ng modified by adding six register spill instructions). The sequence shows how decreasing the tolerance for timing error forces the guest to move deeper into host code structures.

- Figure 3.9 shows integration with a target of $T = 25\pm6$, which contains the gap immediately before the predicate node ReconIMBlock..ng. In this case the guest node can be moved directly without any duplication.

- In Figure 3.10 we see the case for a target of $T = 25\pm4$; this target time is completely contained within the predicate node ReconIMBlock..ng (which begins execution at $T = 19$ and lasts from 12 to 14 cycles), so the guest code must be moved into this node and replicated for both the taken (green arcs) and not taken (red arcs) cases.

- Finally, Figure 3.11 shows the case for a target of $T = 25\pm2$. Here all of the target time in ReconIMBlock..ng’s taken case is contained within the second-level predicate $S34$ (starting at $T = 21$ and lasting 9 cycles), so the guest node must be replicated again, so a total of three versions of the guest node are needed. The price for the increased timing accuracy is increased code memory.

An output assembler program (foo.int.s) is then generated from the integrated CDG and is compiled with GCC and linked to form an executable file.
3.3. Operation of Thrint

Fig. 3.9. Target time 25, tolerance of ±6 cycles
3.3. Operation of Thrint

Fig. 3.10. Target time 25, tolerance of ±4 cycles
3.3. Operation of Thrift

Fig. 3.11. Target time 25, tolerance of ±2 cycles
3.4. Future Extensions

*Thrint* will be extended in two dimensions. First, implementation optimizations which were made in order to focus effort on demonstrating HSM with STI will be fleshed out with more complete or integrated solutions. Second, additional features will be implemented to improve usefulness.

3.4.1. Implementation Optimizations and Their Consequences

Various design decisions were made to concentrate on adding value with the tool, rather than duplicating existing work. Although more sophisticated solutions may exist, the most simple and minimally sufficient choice was preferred. To better explore the embedded system design space, these issues should be addressed.

- The Compaq (formerly DEC) Alpha AXP21064 is not a typical CPU for an embedded system. It was chosen because it was supported by an existing PDG-based research tool (Pedigree/PCOM [NNS94]), simplifying the initial exploration of thread integration. *Thrint* should be retargeted to a more representative embedded processor.

- PCOM performs the parsing of the input assembly code, analyzes loop bounds and creates a CDG-structured assembler file for *Thrint* to process. This functionality belongs in *Thrint*.

- Degenerate integration must be performed by the user before automatic integration can begin. This consists of concatenating the host function code with the guest code. Automating this step requires adding the guest’s automatic variables (modifying the stack and updating references), the guest’s arguments (modifying code in the new integrated function as well as at call sites) and also dealing with a possible guest return value.

- A manually generated file (*foo.id*) contains integration directives which specify guest CDG nodes and their target times with allowable timing tolerance. These directives should be replaced by source-level timing directives to insulate the programmer from assembly code.

- The register file is partitioned statically between threads. For greater integration flexibility a full register allocator is needed, such as coloring or some derivative [CAC+81] [Cha82] or a hierarchical approach such as tiling [CK91]. This will become necessary as programs with greater register pressure are integrated.
3.4. Future Extensions

- Multipredicate nodes [Newb97], which represent unstructured code (from *gotos* and *breaks*) are currently not handled by the data-flow analysis module. Unstructured code is likely to occur in both manually and automatically generated code and therefore must be handled [Engb98]. Although multipredicates can be eliminated through replication [FOW87], this increases code size, which can be at a premium in embedded systems. Instead, direct support for multipredicates should be added to *Thrint*; this will also require some changes in CDG traversal. The net effect of including these structures will be to extend deterministic segments slightly and improve integration without the code expansion caused by replication.

- Irreducible loop nodes [Newb97], which represent loops with multiple entry points [ASU86], are currently not handled by the data-flow analysis. Support will be added in the future.

- Source code location directives (*loc*) are discarded by PCOM and hence are unavailable for debugging at the source level. When *Thrint*’s parser is added it will maintain these directives and assign one for each assembly instruction. This is necessary for tractable debugging; imagine debugging assembly code with both reordered instructions and reallocated registers. Other debug support issues arise as well, and will be discussed in the next section.

3.4.2. Making STI Viable for Design

*Thrint* is currently a research tool designed to investigate concepts for STI and HSM, so it lacks features and “bullet-proofing” critical for an industrial tool. It must be extended in at least four ways to make it a viable option for performing HSM with STI in the industrial environment for production code. First it must be practical to debug at the HLL source level as well as on the assembly language level. Second, *Thrint* must be retargeted to a microprocessor architecture which is more appropriate for the embedded market. Third, it must be backed up with timing verification mechanisms to let a designer run an STI-based embedded system and confirm correct real-time behavior as well as semantics. Fourth, *Thrint* must provide or interact with an integrated development environment (IDE) with powerful and accurate visualization, simulation and debugging capabilities.

First, high-level debugging of integrated threads requires at least two changes. As mentioned in the previous section, source code location directives must be included in the output assembly code generated by *Thrint*. This will be incorporated when the parser is
3.5. Conclusions

added. Register reallocation will require the maintenance of symbol table information contained in the debug assembler directives.

Second, to make STI available to more than the research community it is of course necessary to retarget it to support a processor architecture which is popular among embedded system designers. One interesting question is whether to begin with a low-end (8-bit), mid-range (16-bit) or high-end (32-bit) architecture. Choosing the low-end would help target the most cost-constrained applications, which would benefit the most from HSM in terms of unit cost. However, mid-range and high-end architectures offer enough throughput to work on more elaborate and sophisticated problems. Of course, the availability of 8-bit processors which process 100 million instructions per second [Scen99] has blurred the boundary between low-end and mid-range devices.

Third, it is critical to verify the correct timing behavior of the integrated software. Simple systems can be verified manually. The high-temperature CAN bridge of Chapter 6 (HSCAN) used communication with other CAN nodes driven by Monte Carlo analysis (randomly generated test cases) to find bugs, with the bugs tracked down with an oscilloscope and debugger. As system complexity rises this type of approach will become impractical. Automatic verification requires some form of timing verification tool which can observe target system behavior directly and concurrently (either by monitoring hardware or listening for timestamps generated by instrumentation code within the target) or else during post-processing (using a log file of timestamps). Ideally all verification should be automated. This is an interesting field of research in its own right.

Finally, STI should be incorporated into an integrated development environment (IDE) to give the developer as much help as possible. A graphical interface would be preferable. IDEs typically offer a source code editor, compiler, assembler, linker, loader (including downloader) and debug capabilities both locally (with a cycle-accurate simulator) and remotely (on the target hardware). Linking these components together simplifies the developer’s job. The visualization methods offered by Thrint help the developer understand code structure and behavior better, and should be included in the IDE as well.

3.5. Conclusions

Software’s flexibility is a blessing and a curse. “Sharp” tools enable a programmer to manage a design’s complexity and benefit from this flexibility while avoiding its traps. This makes it easier to develop and maintain software. Embedded and real-time con-
3.5. Conclusions

Constraints complicate software development, but tools have been slow to react for various reasons. Hardware to software migration is an area of real-time embedded systems which is valuable enough to perform manually despite its complexity and tedium.

We have created a tool, Thrint, which automates most of the work in STI, making it much easier to design real-time embedded systems with HSM. Thrint operates on assembly code for timing accuracy, using a hierarchical program representation to simplify code manipulation. It uses static timing analysis to guide code transformations which interleave host and guest code into a single thread. It also provides various forms of graphical output to help the developer understand program behavior.
Chapter 4

Guest Triggering and Host Selection

The integration methods presented in Chapter 2 show how to integrate two or more threads, leading to the questions of when to execute the integrated threads and which threads to execute. This chapter shows how to answer these questions, presenting run-time support techniques to recognize a trigger event and to invoke a guest thread, taking into account timing constraints and unpredictable host function invocation. We develop methods (built into the tool Thrint) to analyze an application, finding good potential host functions through profiling and static timing analysis. This provides guidance to select threads to integrate. These methods help predict the performance of an integrated application before performing the integration.

Integration often 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.

4.1. Guest Triggering Methods

This section describes the two dimensions which must be considered when triggering guests, showing first how to detect a guest trigger event and then how to trigger a guest service thread. A trigger event can be detected through polling or an interrupt, leading to different detection latencies. The guest service thread can be triggered immediately or after some delay, allowing flexibility in choosing which thread to execute. In the first case, the integrated threads can be viewed as efficient interrupt service routines which execute
4.1. Guest Triggering Methods

upon triggering. In the second case, the guest service threads can be viewed as being embedded throughout the application, ready to run if enabled when encountered during the program’s execution.

Two design decisions must be made when choosing how to trigger the guest, as illustrated in Figure 4.1 and Table 4.1. First, the event must be recognized, either with polling or an interrupt. Second, the recognizer must trigger a thread to service the event either immediately or after some delay.

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. We present reasons and methods for choosing and implementing a guest trigger 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 defined as the fraction of time in which the CPU performs useful work rather than padding instructions (which provide time delays). Note

<table>
<thead>
<tr>
<th>Event Recognition</th>
<th>Event Happens</th>
<th>Which?</th>
<th>How?</th>
<th>Set of guest service threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polling</td>
<td>Recognizer</td>
<td>Service Dispatcher</td>
<td>Event Service</td>
<td>Immediate</td>
</tr>
<tr>
<td>High priority send message, or little guest idle time</td>
<td>Low priority send message, or much guest idle time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt</td>
<td>Receive message (fast bus or complex system)</td>
<td>Display refresh in complex system</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.1. Guest Triggering Methods

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.

4.1.1. 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 4.2 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.

4.1.1.1. Polling (Time-Triggered)
4.1. Guest Triggering Methods

The timing for a generic guest event appears in Figure 4.3. Guest trigger events are separated by at least $T_{E_{\text{Min}}}$, requiring the guest’s work to be completed before the deadline $T_D$ has passed. If $T_D > T_{E_{\text{Min}}}$, we set $T_D = T_{E_{\text{Min}}}$ to ensure guest service finishes before another guest event occurs. The guest’s work, including internal idle time, takes $T_G$ to execute on the processor, so it must begin within $T_D - T_G$ of the trigger event. A polling implementation must sample the signal at least every $T_D - T_G$.

To convert an aperiodic guest trigger into a periodic one, the system must poll the trigger frequently enough to invoke the guest service thread in time to meet its deadline. If the guest trigger event has a minimum interarrival period $T_{IA}$ (equal to the inverse of the maximum event frequency), we use this as the minimum event period $T_{E_{\text{Min}}}$ and set the polling period to $T_D - T_G$ as 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 long hosts. 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.

4.1.1.2. 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 described below.

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 [DS98b] uses this method to ensure a pair of display rows is refreshed every 60 µs regardless of other activities on a hand-held computer, which are impossible to predict at compile time.

4.1.2. Event Service

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. Figure 4.4 provides an overview of guest service methods. 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.
4.1. Guest Triggering Methods

a) Immediate Service

Trigger

Are any integrated functions ready to execute?

Yes

Choose most efficient one and execute it

Return

No

Execute dedicated busy-wait guest service routine

Return

b) Deferred Service

Event Recognizer

Trigger

Guest State ← Start

Set WDT to expire before end of slack time

Return

Guest State = Start?

Yes

Execute dedicated guest code

Guest State ← Off

Return

No

Disable WDT

Guest State ← Running

Execute integrated code

Guest State ← Off

Return

Integrated Host Thread

Watchdog Timer ISR

Disable WDT

Guest State = Start?

No

Guest State ← Off

Return

Yes

Reset WDT for new slack time

Execute dedicated guest code

Guest State ← Off

Return

Fig. 4.4. Guest service overview
4.1. Guest Triggering Methods

4.1.2.1. Immediate Service

If there is little slack time, the guest must be serviced immediately. Figure 4.4a shows that after trigger recognition, the service dispatcher must find an integrated function ready to execute. It examines a table such as Table 4.2 of the available integrated functions, looking for one with some input data ready and with the highest efficiency. If any are found, the best is selected and executed. Otherwise the less desirable dedicated guest function executes; its internal timing is set through busy waiting so it is inefficient.

<table>
<thead>
<tr>
<th>Guest</th>
<th>Buffer</th>
<th>Data Ready</th>
<th>Integrated Thread</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCDRow</td>
<td>LineQueue</td>
<td>21</td>
<td>DrawLine_LCDRow</td>
<td>100%</td>
</tr>
<tr>
<td>FillPolygonQueue</td>
<td>13</td>
<td></td>
<td>FillPolygon_LCDRow</td>
<td>81%</td>
</tr>
<tr>
<td>n/a</td>
<td>n/a</td>
<td></td>
<td>DedicatedLCDRow</td>
<td>43%</td>
</tr>
</tbody>
</table>

Table 4.2: Example Guest Service Dispatch Table

Figure 4.5 presents a method for distributing line drawing work among multiple consumers. [DS98b] uses this technique for a handheld computer to share graphics rendering work between application code and an integrated LCD refresh function.

![Fig. 4.5. Line drawing integrated with LCD refresh](image.png)

When a function in the application needs to draw a line, instead of directly calling the DrawLine function it instead inserts the arguments (line endpoints and style) into the buffer LineQueue. This queue is serviced by both the original DrawLine function as well as an integrated version DrawLine_LCDRow which refreshes the LCD while drawing the line. DrawLine is called if DrawLine_LCDRow is not called often enough to keep up with line drawing requests. This can be implemented by calling DrawLine when the queue is
4.1. Guest Triggering Methods

about to overflow, or periodically checking to see if the queue is too close to full and calling DrawLine directly to speed rendering.

When the LCDRow guest is triggered, the dispatcher examines a table such as Table 4.2 and chooses to execute one of the integrated threads (DrawLine_LCDRow or FillPolygon_LCDRow) which has data ready. With no data ready, DedicatedLCDRow is the only option for guest service.

This type of solution requires manual source code modification to decouple work and is application dependent. First, in order to minimize overhead, candidates for decoupling should operate on data which is easily transferred via the queue (either by value or reference). Second, the post-consumer of that data (i.e., the consumer of the data produced by the queue’s consumers) must have sufficiently loose timing requirements, as the processing delay will depend upon consumer call frequency. In this example, the post-consumer of the line data is the computer user’s visual system, so the deadline of line drawing is reasonably loose and depends upon persistence of vision and LCD characteristics.

4.1.2.2. Deferred Service

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 4.4b and Figure 4.6 show how the recognizer sets a flag indicating the guest is ready to start, but also sets a watch-dog timer (WDT) to expire near 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
4.2. Code Timing Analysis and Host Selection

WDT expires, its ISR executes a dedicated guest function (which is implemented with busy waiting).

Deferred service uses a state flag to control guest execution, as shown in Figure 4.6. 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 4.7 illustrates this situation.

![Fig. 4.7. Timeline for deferred service (multiple guests)](image)

4.1.2.3. Dedicated Last-Resort Guest Service

In both the immediate and deferred service cases, a dedicated guest thread triggered by a watchdog timer 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 when it cannot be guaranteed such a function will be called soon enough to meet any deadlines.

4.2. Code Timing Analysis and Host Selection

The choice of host functions for STI dramatically affects 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.
4.2. Code Timing Analysis and Host Selection

We begin by analyzing the computing and timing requirements of the guest code. We then 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.

4.2.1. Guest Analysis

The goals of guest analysis are first determining if a guest thread is worth integrating and then finding its timing characteristics, which are derived 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 the deadline has passed.

Not all potential guests should be integrated, as described in Chapter 5. 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
4.2. Code Timing Analysis and Host Selection

lightly, reducing the need for STI. Finally, a guest may have long sections of idle time, minimizing context-switching penalties.

Figure 4.8 presents an example guest thread with timing requirements and characteristics annotated. The guest code is prepared for analysis by padding so the execution time is consistent. For example, the first two predicates are padded with 7 and 2 nops. 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.

This example is not unreasonable; in applying STI to the software-implemented CAN protocol bridge (Chapter 6 and [DG99]), we found the idle time to be of fine granularity (averaging 20.8 and 26.5 cycles duration), as shown in the weighted histogram of Figure 4.9.

![Figure 4.9. HSCAN idle time bubble size during message transmission and reception](image)

4.2.2. Host Analysis

We next perform timing analysis for potential host functions in the application, extracting information about deterministic execution segments within the functions. This segment timing information is then used to help select hosts for integration. Thr!/nt examines potential host functions to identify segments of code which are sufficiently deterministic. These
4.2. Code Timing Analysis and Host Selection

segments are good locations for integration because their low timing jitter eliminates or reduces the need for padding instructions, which reduce system efficiency and increase program memory.

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. These segments are then measured to find 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 4.10 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 $\sum_{\text{min}}$, $\sum_{\text{max}}$. Thrint evaluates the cost of nop padding to regularizing the segment’s timing; this requires determining cycle cost and program memory cost. Thrint finds the incremental costs for adding each subgraph as well as the cumulative costs.

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 ex-
ceed 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. If the guest thread is too long to fit within a segment it may be possible to partition the guest, but this is left as future work.

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 last no longer than the segment. Second, a WDT must be used to ensure a back-up dedicated guest service thread begins on time.

### 4.3. 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 integrated threads off-load work from the dedicated guest (which is inefficient because it uses busy-waiting), 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 Chapter 2, which essentially is the ratio of loop durations.

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 for 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.

4.4. Experimental Results

To investigate the suitability of STI in a real application we examine a high-resolution MPEG [LeGa91] video player. We use the Berkeley MPEG-1 video decoder mpeg_play [Rowe94] 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 conservative, given high-end embedded processor product announcements from manufacturers [ARM98], [Cata99]. 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 processing method, combined with substantial on-chip memory (or locked cache), will make system timing analysis tractable.

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 at least 1 MByte of SRAM onboard, with a 100 MHz off-chip bus, as shown in Figure 4.11.
4.4. Experimental Results

We use this future microcontroller as a platform for a digital video (MPEG) playback device. STI enables software 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 5% of the speed of the CPU core.

The CPU’s external bus runs at 100 MHz, which reduces 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 [Shar99]. On-chip memory consisting of at least 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 optimization (-O3) and then process it with Pedigree [NNS94], which performs control-flow analysis, and `Thrint`, our tool which per-
forms static timing analysis, integration planning and data visualization. Thrint uses profiling information derived from gprof to help plan integration.

4.4.1. Guest Analysis

`LCD_Refresh_DRow` 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 HSnap 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 HSnap 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 4.3 and Figure 4.12 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.

![Figure 4.12](image_url)

**Figure 4.12. Guest thread timing diagram**

<table>
<thead>
<tr>
<th>Guest Trigger Event</th>
<th>Deadline</th>
<th>WCET</th>
<th>Idle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LCD_Refresh_DRow</code></td>
<td>Internal Timer, 17.01 kHz</td>
<td>58.8 µs</td>
<td>14.4 µs, 10.8 µs</td>
</tr>
</tbody>
</table>
4.4. Experimental Results

A loop inside this function repeatedly loads the LCD driver’s shift register. This loop will be unrolled as needed to fit into host segments, allowing integration to adjust the guest thread size based on host segment size and hence take advantage of most of each host segment.

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).

4.4.2. 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 and optimized 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 as its dataflow information is too complex for Pedigree. The analysis reveals a large amount of code which can easily be made deterministic and used to support thread integration.

<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>j_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>ReconBMBlock</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 4.4 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 4.5 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.
Figure 4.13 shows the maximum jitter of the most deterministic segments found within the host functions. Each data point on the plot represents the growth of a segment by adding a subgraph in the CDG. The jitter is equal to the cost of making a segment deterministic, measured by the number of cycles which must be wasted through padding. \textit{OrderedDitherImage} is not plotted, given its extremely long and fully deterministic path. Only segments with under 1000 cycles of jitter were selected. \textit{ReconBiMBlock} and \textit{ReconIMBlock} both offer long, highly deterministic segments with very low or no padding costs. \textit{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.
4.4. Experimental Results

We find the total CPU throughput spent executing deterministic code by scaling the segment size by call frequency. Figure 4.14 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 of determinism suggests that integration will produce efficient code which can mask a significant amount of idle time.

4.4.3. Predicted Performance Results

Table 4.6 presents the amount of guest work which will be executed by integrated threads. Predicting this value requires finding how many guest loop iterations fit into the host’s deterministic segments. These iteration counts are then weighted by the segment and host execution frequencies. For example, in \texttt{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.
4.4. Experimental Results

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 more than 82 of these row periods, so the remaining 44.4 μs of deterministic code per row period are unavailable.

![Table 4.6: Predicted Thread Integration Results](image)

<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>ReconIMBlock</td>
<td>53</td>
</tr>
<tr>
<td>ReconBiMBlock</td>
<td>47</td>
</tr>
<tr>
<td>ReconBMBlock</td>
<td>6</td>
</tr>
<tr>
<td>ParseReconBlock</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 4.15 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.
The variations in code memory expansion result from two host code characteristics. First, deep conditional structures within the host code force integration to replicate guest code for each condition. *ReconBMBlock* (Figure 4.16) and *ReconPMBlock* both spend most of their execution time in loops contained in conditionals nested up to 5 levels deep, leading to the need for up to $2^5 = 32$ instances of a guest thread fragment within the host, one to cover each possible condition combination.

Second, host loop bodies which are much longer than the guest loop iteration time lead to significant guest loop unrolling. *ReconPMBlock, ReconBiMBlock, ReconBMBlock*, and *OrderedDitherImage* all have long loop bodies, with the guest loop being unrolled 44 times in this last function.
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 4.17 shows cumulatively how required CPU throughput falls and total code memory rises as the guest thread is integrated into more and more 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.

4.5. Conclusions

This chapter presents system-level issues in software thread integration for hardware to software migration. Guest execution can be triggered by polling or interrupts, and to some extent these two methods can be interchanged. A guest’s service can be immediate (due to deadline pressures) or deferred (in hopes of raising system efficiency).

Code analysis helps the system designer to select which threads to integrate. The idle time within a guest is identified to determine whether it is appropriate for reclamation through STI. The application is profiled to find the most commonly executed functions, and these potential hosts are analyzed to detect segments of deterministic code which meet the re-
quirements of the guest threads. The performance results as well as memory and timing costs of integrating the guests and hosts are then predicted based on CDG structure, loop counts and timing requirements, without requiring the actual integration. The designer can then select the host functions for integration which provide sufficient performance at the lowest cost.
Guest Triggering and Host Selection
Chapter 5

Constraints on Using STI for Hardware to Software Migration

Successfully using STI as a design alternative for embedded systems requires understanding the constraints limiting its use and comparing them across all migration options. Each method has its own strengths, resulting in different “sweet spots” within the design space. This chapter identifies the relevant factors and characterizes them for the different migration methods using single instruction stream processors.

In order to compare HSM methods we use the two hardware architectures presented in Figure 5.1. In the first, the function of interest is implemented in hardware, while the second uses software guest code for the function. Either system may consist of discrete modules or be integrated onto a single chip; the partitioning is not critical with respect to HSM. A software-based design (Figure 5.1b) can be implemented with traditional HSM
(context-switching and busy-waiting) or STI. For traditional HSM, a programmable timer generates interrupts after a software-defined delay. These interrupts can trigger real-time threads in the guest function, simplifying implementation. The guest code is structured as a single thread written as a finite state machine which advances through its states when triggered.

This chapter first quantifies the processor throughput requirements for the different HSM methods. We first measure the guest software’s maximum CPU utilization, add in the overhead for the different HSM methods, show how to choose a method, and then predict the system’s average CPU utilization. The second section discusses the effects of STI on the memory subsystem within the embedded system. The third section discusses program timing determinism, which is necessary for the integration of real-time threads, yet is reduced by software and hardware features.

5.1. Processor Throughput

The first question raised when considering implementing a real-time function in software is how fast the processor must be to perform the required work while meeting deadlines. We use the metric of utilization remaining \( U_{Rem} = 1 - U_{Max} \) to evaluate system performance for a processor of a given speed.

This section mathematically quantifies the computational cost of a set of real-time software operations, for example from HSM. It first finds the peak loading without considering the concurrency mechanism, and then adds in the effect of the different mechanisms. It then uses these computation costs to define the “sweet spots” for the different HSM methods. This section concludes by quantifying the computational cost for a HSM implementation taking into account multiple host threads with different call frequencies.

5.1.1. Peak Loading

![Fig. 5.2. Operation timing definitions](image)

As presented in Figure 5.2, we assume the Guest \( G \) consists of a set \( \Omega \) of \( I \) operations \( \omega_i \), where each operation contains one or more basic blocks and at least one instruction with
5.1. Processor Throughput 89

real-time constraints. There may be timing constraints between operations, specifying minimum \( \delta_{\text{Min}}(\omega_i, \omega_j) \) or maximum \( \delta_{\text{Max}}(\omega_i, \omega_j) \) delays permitted between those operations. During peak guest activity, each operation is invoked at a frequency of \( f_{\text{Max}}(\Omega) \). We assume all operations \( \omega_i \) in \( \Omega \) are in a repeating schedule and therefore are invoked at the same frequency \( f(\Omega) \). Each operation (which contains one or more instructions) requires a number of cycles of computation ranging from \( C_{\text{Min}}(\omega_i) \) to \( C_{\text{Max}}(\omega_i) \), with an average of \( C_{\text{Ave}}(\omega_i) \). The worst case CPU loading by the guest is \( C_{\text{Max}}(\Omega) \):

\[
C_{\text{Max}}(\Omega) \equiv f_{\text{Max}}(\Omega) \cdot \sum_{\forall i} C_{\text{Max}}(\omega_i)
\]

This computation requirement does not include overhead for switching contexts or scheduling delays, which will be considered in the next section. The computation imposes a utilization \( U_{\text{Max}}(\Omega) \) on a CPU with a clock rate of \( f_{\text{CPU}} \):

\[
U_{\text{Max}}(\Omega) \equiv \frac{C_{\text{Max}}(\Omega)}{f_{\text{CPU}}}
\]

The CPU instruction cycles which remain after executing the guest threads are likely to be fragmented and too small to use easily. For example, consider the behavior of HSCAN (the software implementation of a 10 kbps CAN protocol bridge in Chapter 6 and [DG99]) when sending or receiving an eight byte message. To send or receive a bit, the CPU must perform bit stuffing, cyclic redundancy check calculations, and other activities as well as moving the bit between the CPU and the CAN bus. When scheduled for an 8051 microcontroller running at 22 MHz (1.83 million instruction cycles per second), processing each bit requires about 89% or 92% of a bit time (sending and receiving, respectively). The remaining time is not used, resulting in a bubble of idle time in each bit period (every 183 cycles), with durations distributed as shown in Figure 5.3. This plot presents idle time distribution and shows the majority of the idle time comes in short bubbles (on average 26.5 cycles long when sending messages and 20.8 when receiving).
Reclaiming these bubbles requires some method of allocating the CPU among multiple operations. The concurrency mechanism chosen typically imposes a performance overhead on top of the guest loading. An ideal mechanism for concurrency would impose no penalty for switching between different threads. STI is able to reach this limit under the best circumstances.

5.1.2. Context Switching Costs

Context switching requires saving and restoring processor state information, which typically happens each time a separate thread needs to run. This occurs to varying degrees with control transfers such as interrupt service routines, task switching, procedure calls and coroutine calls. In a processor with \( N_{\text{Registers}} \) state registers (not partitioned), a memory access time of \( C_{\text{Memory}} \) cycles, and an interrupt vectoring delay of \( C_{\text{Vector}} \), the context switch cost \( C_{\text{CS}} \) can reach \( 2*N_{\text{Registers}}*C_{\text{Memory}} + C_{\text{Vector}} \) cycles. If an operating system handles interrupts, \( C_{\text{Vector}} \) can become much larger than the memory access time. We assume that hardware interrupts are used to trigger context switches directly without intervening scheduler actions (which increase delays and reduce performance).

There can be many variations in context-switching activities, leading to different delays:

\[ C_{\text{CS}} = 2*N_{\text{Registers}}*C_{\text{Memory}} + C_{\text{Vector}} \]
• In certain cases, a thread may have no state to restore. For example, an interrupt service routine (ISR) may run to completion each time it is triggered, so it loads all values from memory as needed, rather than before execution. In this case, $C_{CS} = N_{Registers} \cdot C_{Memory} + C_{Vector}$, as registers are only saved (upon ISR triggering) but none are restored.

• Another alternative is one in which the ISR saves only the registers it will use, in which case $C_{CS} = N_{RegistersSaved} \cdot C_{Memory} + C_{Vector}$.

• A hardware timer may be used to trigger the execution of the next guest operation, often with a very low initialization cost $C_{TimerSetUp}$ (e.g. 7 cycles on an 8051 microcontroller [Phil96]). This provides flexibility of implementation, yet even this low cost may be excessive for applications with very fine-grain idle time.

In any case, an idle time bubble can be reclaimed for other use only if it is longer than two context switches (one at the beginning and one at the end, $2 \cdot C_{CS}$). For the HSCAN example, using an timer-based interrupt to interleave processing would lead to a total overhead of 24 cycles per interrupt. This overhead would waste most of the idle time when sending (86%) or receiving (91%) messages, making this approach too inefficient for reclaiming idle time.

We define two separate context switch cycle costs: $C_{CS1}$ represents the cycles required to respond to the guest, and $C_{CS2}$ represents those for the guest to yield the processor back to the preempted procedure. These times may vary because guests need only save the registers which they will use.

The penalty for switching contexts can become significant as the event frequency and therefore context-switching frequency rise. This reduces the size of the idle time bubbles, reducing the number of bubbles large enough to be worth reclaiming as well as total idle time.

The maximum context switch frequency $f_{CSMax}$ occurs when all of the processor’s time is spent switching contexts:

$$f_{CSMax} = \frac{f_{CPU}}{C_{CS}}$$
The maximum event rate for guest events handled by an operation $\omega_i$ interrupt service routine is limited by two context switches as well as the work which must be done:

$$f_{\text{EventCSMax}} = \frac{f_{\text{CPU}}}{\sum C_{CS} + \sum C_{Max}(\omega_i)}$$

### 5.1.3. Traditional Migration

Traditional migration uses both context switching and busy waiting to schedule operations. Predicting the performance of such a system during worst-case traffic first requires finding the worst-case idle time between guest operations by scheduling for a given system (CPU and memory with their respective timings). These operations $\omega_i$ are then joined into subthreads $\theta_j$ based on timing and performance goals.

![Fig. 5.4. Operations and subthreads with traditional HSM](image)

Figure 5.4 above shows how operations $\omega_3$ and $\omega_4$ (separated by a delay $\delta_{\text{min}}(\omega_3, \omega_4)$ < $2C_{CS} + C_{\text{Timer}}$) are concatenated into a subthread $\theta_3$, as the idle time is too short to be used. This subthread is internally self-timed [Chou95], i.e. statically scheduled through busy-waiting. Subthreads are triggered by interrupts (either time- or event-based) or through a coroutine mechanism. It may be desirable to concatenate operations separated by larger idle times. For example, if the interrupt response time features significant jitter, operation timing requirements may be violated with interrupt triggering, suggesting instead additional concatenation and padding.

We set an indicator function $N(i)$ to one to show when an operation $\omega_i$ begins a new sub-thread $\theta_j$. For an operation $\omega_i$ which is concatenated with its predecessor $\omega_{i-1}$ to extend the current subthread the indicator function is set to zero.
5.1. Processor Throughput

Figure 5.4 presents various operations transformed into subthreads. Even in the worst case, operations $\omega_1$ and $\omega_2$ are separated enough in time (by $\delta_{\text{Min}}(\omega_1, \omega_2)$) to allow context switching, so they form separate subthreads $\theta_1$ and $\theta_2$. Operations $\omega_3$ and $\omega_4$ are too close, so they are joined to form a single subthread $\theta_3$ with the idle time between them filled with busy waiting instructions. Subthread computation costs range from $C_{\text{Min}}(\omega_j)$ to $C_{\text{Max}}(\omega_j)$, with an average of $C_{\text{Avg}}(\omega_j)$.

We can now use these subthreads to evaluate system performance. Under the heaviest loading conditions (worst case execution and triggering), the processor utilization from both guest work and overhead is $U_{\text{MaxCSBW}}(\Omega)$:

$$U_{\text{MaxCSBW}}(\Omega) = \frac{f_{\text{Max}}(\Omega)}{f_{\text{CPU}}} \cdot \sum \left( C_{\text{Max}}(\omega_i) + N(i) \cdot (C_{\text{CS1}}(\omega_i) + C_{\text{CS2}}(\omega_i)) + (1 \leq N(i)) \cdot d_{\text{Min}}(\omega_i \leq 1, \omega_i) \right)$$

To simplify, we can characterize loading using the subthreads $\theta_j$ rather than the operations $\omega_i$. Note that any padding between operations is reflected in the term $C_{\text{Max}}(\theta_j)$.

$$U_{\text{MaxCSBW}}(\theta) = \frac{f_{\text{Max}}(\theta)}{f_{\text{CPU}}} \cdot \sum \left( C_{\text{Max}}(\theta_j) + C_{\text{CS1}}(\theta_j) + C_{\text{CS2}}(\theta_j) \right)$$

A utilization greater than one means a faster processor is needed to implement the function in software. A value of one or less indicates the processor has enough throughput to handle the work (without considering deadlines); a faster processor may still be needed depending upon the real-time requirements.

5.1.4. Migration with Software Thread Integration

STI schedules guest operations within one or more host threads, essentially filling guest idle time with host work. The host thread must initially be processed to regularize its timing enough to meet guest timing accuracy requirements. This involves inserting padding instructions (busy waiting) to compensate for program paths of different durations. In addition, guard operations may be added to limit guest code execution to specific circumstances (e.g. certain host loop iterations). Register pressure may force variables to be spilled to and restored from memory. Host threads should be selected to minimize the
amount of such overhead. Figure 5.5 presents how the guest operations \( \omega_i \) might be integrated with a host thread \( H_j \) to form \( \Theta_k \) and shows various types of overhead.

It is important to note that most of the context switches are eliminated; we compute the average utilization for STI implementations later in this chapter. If the adjustments made to the host thread create less overhead than these switches, STI provides a more efficient solution. In practice we have found that STI results in marginal performance overhead, as there is a large amount of usable temporal determinism in typical applications.

Because the integrated thread has real-time constraints, it is assumed to be uninterruptable. This lack of preemption increases the CPU’s maximum interrupt response latency by the worst-case duration of the integrated function. Any other real-time functions requiring a response faster than this must be implemented in hardware.

5.1.5. Migration Method Selection

Examining the peak event rate of the guest and the context switch delay on a given processor indicates when to use which form of migration, as illustrated in Figure 5.6. For event rates from \( f_{EventCSMax} \) (the maximum event rate for a context-switching solution) to \( f_{Ops} \) (\( f_{CPU}/C_{Max}(\Omega) \), the maximum rate at which the CPU can execute operations) all CPU capacity is used for real-time event processing; this is the upper part of thread integration’s sweet spot when considering event and processor operation rates. Using STI for lower event frequencies increases system processing efficiency by reducing context switches.
5.1. Processor Throughput

Setting a limit $L$ for processor loading from context switches defines an event frequency lower bound for the STI sweet spot:

$$f_{\text{EventLowerBound}}(L) \equiv \frac{L \cdot f_{\text{CPU}}}{C_{CS1} + C_{CS2}}$$

The sweet spot for context switching lies below this event frequency, as the performance overhead is acceptable (below $L$). For a 10 MHz CPU with a 25 instruction cycle context switch and an efficiency limit of 10%, the sweet spot for STI ranges from event frequencies of 20 kHz to 10 MHz. This sweet spot covers such varied applications as audio sampling and reconstruction, low to moderate speed bit-banged network and liquid crystal display refresh. The sweet spot for traditional migration with context switching lies below 20 kHz.

5.1.6. Average Loading

It is useful to predict a system’s long-term average performance to get a sense of how much processor capacity remains for background tasks which can be deferred during peak processing. We assume that events occur with a fixed frequency but with a phase jitter which varies uniformly and hence spreads events evenly over time.

The utilization for the traditional HSM method is simply the average computation cost of the subthreads (as well as the context switch cost per subthread) weighted by event frequency.
The STI utilization is more complex; if an STI host function $H_j$ is not called often enough to meet the event frequency or deadline requirements, a last-resort traditional (busy-waiting and context-switching) dedicated service routine is used. The first case applies to operation with the event frequency $f(\Omega)$ less than or equal to the host function invocation frequency $f(H_j)$. In this case the integrated threads are called to service the guest events, so the guest loading consists of the guest operations $\omega_i$, any guard code $G_i$ for those guests, and potentially the context switching costs for the host function. If the temporal jitter in the host exceeds the guest’s timing error tolerance, it is necessary to pad away this jitter. This padding is represented by the term $C_{AveJitterPad}(\omega_i, H_j)$.

$$U_{CSBW}(\theta) = \frac{f(\theta)}{f_{CPU}} \cdot \sum_{\forall i} (C_{ave}(\theta_i) + C_{CS1}(\theta_i) + C_{CS2}(\theta_i))$$

The second case applies to operation with the event frequency $f(\Omega)$ greater than the host function invocation frequency $f(H_j)$. Here the first term describes the loading from the integrated thread, while the second term covers the dedicated context-switch/busy-wait subthreads for overflow guest events. Note that each $C_{AveCSBW}(\theta_i)$ subthread computation cost term may include busy-waiting needed to schedule concatenated operations.
5.1. Processor Throughput

For illustration, consider the guest operations described in Figure 5.7, which performs message transmission on a serial data link. A complete message lasts 5 ms (50,000 cycles) on a 10 MIPS processor. However, only 3600 cycles of instruction execution are needed, so the guest requires only 7.2% of the processor capacity during message transmission.

The guest operations $\omega_0$ to $\omega_{2399}$ are listed in the table in Figure 5.7. The idle time is distributed between the guest operations as shown in the plot, with bubble duration ranging from 16 to 24 cycles. A context switching delay of longer than 12 cycles will prevent the recovery of this idle time using context switching.

Figure 5.8 shows the system performance of three different design options for the guest functionality. The vertical axis shows processor capacity remaining after guest processing $I-U$ as a function of guest event frequency $f(\Omega)$. The hardware option is shown in blue, while the two software options are shown in orange and yellow.

Using traditional context-switching and busy-waiting techniques involves concatenating all 2400 operations into a single subthread and filling the idle time with nops. This raises the system utilization during guest execution to 100%. As seen with the orange plot, when the event frequency $f(\Omega)$ reaches 200 Hz, the processor spends all of its time executing either the guest operations or the busy-wait padding, so the remaining capacity for other work is zero.
5.2. Processor and Memory System Impact

STI allows the recovery of most or all of the idle time. The yellow STI line in Figure 5.8 shows the maximum performance possible for these guest operations. We assume the host function is called at 100 Hz, so the guest can be serviced up to 100 times per second using an integrated thread. Above this rate another service function must be used. For example, the dedicated guest service thread described in the previous paragraph can be used, resulting in the reduced system efficiency showed by the sharply dropping yellow STI plot. Performance in this region can be improved by integrating the guest with additional host functions. As stated previously, this plot shows the maximum performance for an STI solution. Guard instructions and other execution-time overhead can keep a system from reaching this limit.

5.2. Processor and Memory System Impact

Software Thread Integration directly affects an embedded system’s memory system by increasing the code size. There may be an indirect impact if a faster CPU is needed, as it will need faster memory to operate. Microcontrollers with on-chip memory are well-suited to mitigate the memory-related drawbacks of STI.

Many of STI’s code transformations copy or create instructions, leading to code expansion. This will increase system cost if a larger or additional memory array is needed. However, when the code increase is considered with respect to the total code size, it becomes
less significant. The examples used in this dissertation have code expansion ranging from under 2% (I$^2$C communications for a cellular telephone) to 32% (for the high-temperature CAN bridge of Chapter 6). In the final analysis, how much code expansion is acceptable depends upon economics of a given embedded system.

The costs of implementing a real-time function in hardware or software are relatively clear if the system’s CPU is fast enough to support the integrated threads’ additional load. However, the situation becomes more murky when the CPU is not fast enough. Speeding up the CPU may also require speeding up the memory, leading to two sources of increased cost.

Increased CPU performance can come from semiconductor processing improvements, which reduce delays and hence increase clock rates. Faster memories are more expensive, and raising the bus speed may lead to other problems with signal integrity and power dissipation. Designers of high-reliability avionics systems try to keep CPU/memory bus speeds at a minimum to limit potential problems (e.g. 25 MHz instead of the 66 MHz common in the personal computers of the time [Beac95]). A memory system’s throughput can be raised by increasing the bus width, which raises chip count, cost, power and board area. All of these factors make it more difficult to predict system cost when considering a faster CPU for an STI implementation.

Performance can also come from architectural and microarchitectural features. These features may adversely affect the memory system or STI process. For example, pipelining increases CPU throughput but also increases memory bandwidth requirements, requiring a faster memory system. A cache increases average performance but requires a more elaborate STI solution, such as cache performance prediction during static timing analysis or else cache locking during execution.

Microcontrollers, as they include memory on-chip, are well positioned to mitigate some of these penalties. DSPs often provide multiple data address spaces which can be accessed simultaneously. Separate on-chip program and data memories support pipelining and wider data buses, and on-chip memory sizes continue to grow. Very high speed microcontrollers and DSPs use fast on-chip memory to reach their maximum instruction processing rates. Typical devices may include on-chip cache, on-chip memory, or a configurable block which can perform either function. The processor core runs at full speed when accessing this local storage but must slow down for off-chip accesses. Some microcontrollers even eliminate the external bus for system cost reasons, making the issue moot.
5.3. Determinism

In order to schedule time-critical guest code correctly, thread integration requires accurate timing information describing the program’s run-time behavior. Both the system’s software and hardware can complicate program execution time prediction [MML97]. Many microarchitectural features which improve performance do so inconsistently, introducing timing variability and occasionally briefly reducing performance. These variations force real-time embedded system designers to add performance margins which increase system costs.

Caches, pipelines, superscalar instruction dispatch, virtual memory and variable latency instructions will increase the variability of system performance with different data sets. Embedded real-time systems often lack many of these features because of cost or temporal determinacy constraints. Pipelines are the most common microarchitectural performance enhancement for embedded processors, and need to be addressed by modeling pipeline behavior when predicting program execution behavior.

We use path analysis to predict the timing behavior of the host and guest software. Path analysis requires that the threads evaluated have bounded loop iterations and no recursive or dynamic function calls [PK89]. We have developed methods to identify and use sufficiently deterministic segments of code within application programs. These appear in Chapter 4. Much embedded software meets these requirements and lies in this “sweet spot” including all the examples presented in this thesis. For example, Chapter 4 reveals that an MPEG decoder spends a third of its time executing code which is deterministic enough to integrate with a thread refreshing a large-format liquid crystal display.

Any variability in timing which exceeds application limits needs to be eliminated through code transformation techniques such as code motion to or from predicates and padding with nops. These changes increase memory requirements and decrease program efficiency.

An interesting trend has emerged recently among high-speed microcontrollers. Some designers of high-volume embedded processors have created microarchitectures for their next-generation devices which back away from superscalar instruction processing in order to reduce clock cycle times [ARM98], [Cata99]. This swing of the work-per-clock pendulum simplifies the microarchitecture and hence simplifies static timing analysis, making STI more feasible for high performance embedded applications.
5.4. Conclusions

The utility of implementing a real-time guest function in software using STI depends upon several system characteristics. First, the guest must offer sufficient idle time to be worth reclaiming, distributed with a fine enough granularity to make a context-switching solution too wasteful. Second, moving a function from hardware into software will increase the system’s processing throughput requirements, possibly forcing the use of a faster (and more expensive) processor and memory system. There is a bus speed wall beyond which performance gains become too expensive; this wall is much lower for most embedded systems than for desktop PCs (which operate in controlled environments). Finally, there must be host functions which execute with sufficient determinism to allow integration with guest functions. The functions must follow predictable control flow paths and the hardware upon which they run must execute instructions at predictable speeds. Within these constraints STI offers an automatic method to move real-time functions from dedicated peripheral hardware into software running on a generic microprocessor.
Constraints on Using STI for Hardware to Software Migration
Chapter 6

Hot Soft CAN: An HSM Prototype

This chapter describes an experimental implementation of a communication network protocol bridge for a high-temperature (185° C / 365° F) application. Software thread integration is used to interleave multiple real-time program threads for efficient concurrent execution, allowing an off-the-shelf, high-temperature, low-speed 8 bit microcontroller to translate messages between a CAN bus and a serial UART data link. The prototype system, which has been built and tested, demonstrates the feasibility of using STI in real-world applications. STI provides the techniques needed to automate the tedious hand-scheduling of time-sensitive code from multiple simultaneous threads so common in cost-driven microcontroller-based applications.

6.1. Introduction

We use STI to create a program for an 8051-compatible microcontroller which functions as a communication bridge between a 10 kbaud CAN network [Bosc91] and a 9.6 kbaud serial link. Our previous work in STI has focused on integrating functions in multimedia applications from simulated systems using high-end 32- and 64-bit microprocessors. The work described in this chapter presents a complete application which has been built and tested at high temperature using a low-end 8-bit microcontroller, demonstrating STI's suitability for a wide range of realistic applications.

Designers of high-temperature embedded systems (HTES) are constrained by the scarcity of semiconductor components capable of operating in a harsh environment (in this case 185° C / 365° F). Low production volumes lead to high prices due to the lack of economies of scale. In addition, these components tend to be mature designs, which prevents HTES designers from benefiting from recent advances in technology. A common alternat-
6.2. Experimental Prototype

tive to designing exclusively with HT parts is to provide a cooling system to moderate the environment, allowing standard-temperature components to be used. These solutions range from simple (using a larger heatsink) to complex (cooling electronics with jet fuel in an aircraft engine). Many applications are unable to bear the added cost, size, weight, complexity or reliability risk of such a cooling system, leaving high-temperature components as the only option.

Moving a real-time function from dedicated hardware to software (hardware to software migration) running on a microcontroller allows a wide variety of processing to be performed, limited only by the speed of the microcontroller and the performance of the software. The potential benefits of eliminating hardware components include less reliance on component availability, higher reliability, smaller size, lower weight, lower cost and lower power. Using more software can increase function availability and flexibility, and reduce time to market and system redesigns resulting from component obsolescence.

Two factors limit the use of HSM for high-temperature embedded systems. First, the microcontrollers available offer very limited performance (under two million instructions per second), constraining the range of real-time functions which can be moved into software. Second, the traditional techniques for HSM are complicated and produce inefficient code, further reducing the feasibility of migration. An interrupt-driven approach not only incurs context-switch penalties which limit efficiency with frequent events, but can also introduce significant timing jitter in other tasks, which may be real-time and hence very sensitive to such variations. A busy-waiting solution is tedious to implement, as the code must be implemented in assembly language for timing precision and then measured and padded to ensure events are timed correctly.

6.2. Experimental Prototype

A prototype high-temperature automotive controller (“Smart Power Distribution Box”) being developed at UTRC (United Technologies Research Center, East Hartford, CT) needed a CAN (Controller Area Network) interface, but no appropriate high-temperature hardware solutions exist. Hardware to software migration with traditional software techniques would lead to an unacceptably low CAN data rate on the available high-temperature microcontroller. STI was selected to implement the CAN interface in software running on the high-temperature microcontroller.
6.2. Experimental Prototype

Figure 6.1 shows the hardware architecture of the prototype HSCAN (Hot Soft CAN) system, consisting of a Honeywell HT83C51 high-temperature microcontroller [Hone98a], a memory system and a CAN physical layer interface. The microcontroller operates at 22 MHz, executing at most 1.83 million instructions per second. External memory contains an EPROM, which transfers the program to a high-temperature SRAM (HT6256) [Hone98b] for actual execution. The CAN physical layer is based on a standard room-temperature circuit [Phil95b] and modified to use high-temperature components. Figure 6.2 shows the HSCAN prototype board mounted on the controller board.
The software for HSCAN consists of two threads, one for bit-banged 10 kbps CAN communication (denoted CAN interface thread) and one for communicating with the controller across a 9600 baud serial link (denoted controller interface thread) as seen in Figure 6.3. The threads jointly manage a pair of message queues for message transmission and reception.

6.2.1. CAN Interface Thread

Controller Area Network (CAN) [Bose91] is a multimaster communication network protocol designed by Bosch for use in real-time embedded systems. Messages are short (0 to
8 bytes of data payload), contain an identifier which determines bus access through non-destructive arbitration, and provide error detection through a 15 bit cyclic redundancy check code and bit stuffing. CAN is popular in the European automotive market and industrial control applications world-wide. Bit rates can reach 1 MBaud. Designing CAN into a real-time distributed system is simplified by the lossless prioritized arbitration method, which allows the network to be modeled as a priority queue. Many protocol controller devices are available for operation at standard temperature ranges, but not at the extreme temperature required in this project.

In addition to simply sending and receiving messages, a software implementation of CAN must coordinate these activities with buffer management, error handling and idle bus detection. In HSCAN an executive function performs these functions; it monitors the bus when idle, it calls a message receive function after receiving a valid message start bit, and finally it calls a message transmit function when the bus is idle and a message is waiting in the transmit message queue. In addition, the executive transfers control from message transmit to receive functions upon losing arbitration.

To be a good candidate for STI, there must be enough idle time in the CAN software to allow other real-time threads to execute and meet their deadlines. We measure the idle time granularity within the three CAN threads to determine suitability for integration. The CAN bus monitor samples the bus at three times the bit rate (about every 33 µs) to ensure that when a message is received, sampling begins in the middle third of the bit. This timing requirement, combined with the CPU processing speed and code size, results in 22 to 30 µs (40 to 56 cycles) of idle time per 33 µs loop iteration. We use software thread integration to reclaim part of this idle time for controller communication.

Figure 6.4 presents the distribution of idle time in the other two CAN threads when an eight-byte message is being transmitted or received. The CAN message transmit and receive functions repeatedly call send and receive bit functions, which also perform protocol requirements such as CRC computation, bit stuffing or destuffing, and bus error detection. When transmitting an eight byte message, the CPU is idle for up to 66 µs (121 cycles) per 100 µs bit, for a total of 2619 cycles out of the message’s duration of 21777 cycles. However, this idle time is fragmented, with such a fine granularity as to make reclamation difficult with context switching, given its seven-cycle overhead. We reclaim these idle times through software thread integration. Note that these idle times fall as the desired CAN bit rate rises or the CPU speed falls.
6.2. Experimental Prototype

6.2.2. Controller Interface Thread

The controller interface thread responds to commands from the serial port to transmit or receive messages and manage the message queue, as shown in Figure 6.5a.

- A transmit command causes the thread to load the transmit queue with a message from the UART following the command.
- A receive command causes the thread to examine the receive queue for a complete incoming CAN message. If one exists, it is sent through the UART to the controller.
- A queue status request leads the thread to reply via the UART with a sequence of four bytes, indicating queue message counts and free space.
- Two flush commands allow either the transmit or receive queue to be erased immediately.

The first three of these operations repeatedly access the UART, creating idle time which is large enough to be worth reclaiming.

If an interrupt is used to detect incoming bytes, the service routine will create timing jitter (unless specific compensation code is added). During the longest CAN message, an interrupt service routine (which must be at least 8.7 µs long) could be triggered 13 times, lead-
6.2. Experimental Prototype

According to timing errors reaching 113.1 µs (which is greater than a bit-time). This error will corrupt an incoming or outgoing CAN message. Instead, STI is used to integrate the controller interface routine with three functions of the CAN thread. STI allows the software to be structured in an easily developed and maintained form.

The controller interface thread monitors the UART for incoming message activity. When a byte is received, it must be unloaded before the next arrives to eliminate the possibility of receive buffer overrun. With a 9600 baud serial link, the controller interface thread must be called often enough to service the UART at least every 1.04 ms.

We convert this thread into subthreads, which are similar to the states of a finite state machine. Each time the controller interface thread is called, a complete subthread executes, possibly enabling a different subthread to run at the next call. This simplifies integration in two ways. First, the switching among subthreads restores the control-flow flexibility lost through integration. Second, a subthread can easily be partitioned into smaller subthreads, allowing a thread to be adapted to fit into short sections of idle time as well as eliminating blocking due to non-deterministic inputs.

In fact, the controller interface thread uses both of these techniques. Each subthread is partitioned to last no more than 78 cycles and contains no more than one potentially blocking UART access. Each subthread with such an access is guarded to prevent its execution if the UART is not ready. This partitioning increases the number of subthreads between UART receive buffer servicing, so the thread must be called more frequently to avoid overruns (three times per 1.04 ms, as at most two subthreads separate receive buffer servicing).

We convert this thread into subthreads (Figure 6.5b), which are similar to the states of a finite state machine. Each time the controller interface thread is called, a complete subthread executes, possibly enabling a different subthread for the next call to the thread. This simplifies integration in two ways. First, the switching among subthreads restores the control-flow variability lost through integration. This makes it much easier to allow the individual threads to execute at their own paces, rather than in lock-step. Second, a subthread can easily be partitioned into smaller subthreads, which allows a thread to be adapted to fit into short sections of idle time as well as eliminating blocking due to nondeterministic inputs.
6.2. Experimental Prototype

6.2.3. Software Thread Integration

After the controller interface thread is converted into subthreads it is integrated into three of the CAN functions which make up the CAN interface thread. As mentioned above, the CAN interface thread spends its time monitoring an idle bus or sending or receiving a message. The controller interface thread is integrated to meet its timing requirements in each of these three conditions.
When monitoring the idle bus, the CPU has from 40 to 56 cycles of idle time per 33 µs iteration. The controller interface subthreads are too long (78 cycles) to fit into this gap, so the bus monitor loop is unrolled once to double the amount of idle time. In the midst of this idle time the bus must be sampled, so this code is integrated with the controller interface by replication into each of the subthreads at the appropriate time. The bus sampling code is replicated 5 times leading to code expansion of 45 bytes in addition to the 496 bytes of the duplicated controller interface, as shown in Table 6.1. This replication is less than the total number of subthreads (9) as several share the same padding code into which a copy of the bus sampling code is integrated. If loop unrolling is not possible, the subthreads would need to be further partitioned to fit into the available idle time.

Table 6.1: Code Expansion (in bytes) from Integration with Controller Interface

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Size</th>
<th>Thread B</th>
<th>Size</th>
<th>Integrated Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>controller_interface</td>
<td>496</td>
<td>idle bus sampling</td>
<td>9</td>
<td>541</td>
</tr>
<tr>
<td>controller_interface</td>
<td>496</td>
<td>send_bit</td>
<td>185</td>
<td>684</td>
</tr>
<tr>
<td>controller_interface</td>
<td>496</td>
<td>receive_bit</td>
<td>157</td>
<td>654</td>
</tr>
</tbody>
</table>

Integration of the controller interface with send and receive messages functions is straightforward. Because each bit must be sent or sampled at a specific time, the message functions provide a scheduled framework into which the controller interface thread is integrated. The repeated calls to the send_bit() and receive_bit() make them logical candidates for integration. The idle time per bit while transmitting a message varies from 91 to 121 cycles. The minimum idle time of 91 cycles is longer than the controller interface
thread, so degenerate integration with send_bit() is sufficient; the entire thread is appended to the end of the function. The procedure is similar for receive_bit(). This duplication of code leads to significant code expansion.

Integration raises the total code memory size from 3520 to 4647 bytes, or about 32%, as listed in Table 6.2. This expansion could be reduced through the use of more space-efficient padding techniques. The HT83C51 offers 8 kilobytes of on-chip memory, so this expansion is quite acceptable. In more typical high-volume applications, it is likely that the cost savings of eliminating a dedicated peripheral component will exceed the cost of the microcontroller (if added) and its memory.

### Table 6.2: Code Expansion (in bytes)

<table>
<thead>
<tr>
<th>Function</th>
<th>Original Size</th>
<th>Integrated Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_bit</td>
<td>111</td>
<td>684</td>
</tr>
<tr>
<td>receive_bit</td>
<td>100</td>
<td>654</td>
</tr>
<tr>
<td>can_interface_executive</td>
<td>290</td>
<td>831</td>
</tr>
<tr>
<td>controller_interface</td>
<td>541</td>
<td>0</td>
</tr>
<tr>
<td>Other</td>
<td>2478</td>
<td>2478</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3520</strong></td>
<td><strong>4647</strong></td>
</tr>
</tbody>
</table>

### 6.2.4. Development Time

Software development and testing for HSCAN took about 100 days, with about a quarter dedicated to developing the independent threads without considering real-time requirements. A break-down of the time appears in Figure 6.7. Conversion to real-time operation (through both simulated and actual execution on the target processor) combined with integrating the threads took over half of the total time. The software development effort could be simplified dramatically through the use of an automated tool for STI. More importantly, software maintenance could be made much easier with such a tool, as each thread could be operated upon in its source code format, insulating the user from assembly language concerns, improving portability and extending the useful life of software.
6.3. Conclusions

The prototype system was tested and functioned successfully for an extended period at temperatures ranging from 185°C to 225°C, meeting all project goals. Figure 6.8 shows the boards after testing in the actual high temperature environment. Although the printed
6.3. Conclusions

circuit boards have been browned, the HSCAN prototype system performed flawlessly in passing messages between the CAN network and the UART without errors, demonstrating the feasibility of the HSM and STI techniques.

Whether driven by cost or function availability pressures, embedded system designers use many ad hoc techniques to implement and interleave real-time functions in software on microprocessors. Software thread integration provides a set of methods which simplify hardware to software migration and makes it possible to implement HSM in a systematic and automatic way with a software tool. This tool enables designers to work at a higher level of abstraction without worrying about manual tweaking of assembly code. At this level designers become much more efficient and can implement (and maintain) more sophisticated embedded real-time systems.
Chapter 7

Other Potential Applications for STI

This chapter describes how reliable systems can benefit from STI. In particular, STI can be used to enhance an existing thread by integrating a reliability-enhancing or error-detecting thread. The case of duplicating a thread is described first. This method reduces error detection latency. The case of integrating a thread with a different thread is covered second. This reduces error detection latency and can enhance performance through data locality.

The concurrency provided by software thread integration is well suited for fault-tolerant programs in embedded systems. A monitoring or checking thread can be integrated into application code to offer concurrent error detection, which reduces detection latency. This also allows the exploitation of idling or underutilized machine resources in modern superscalar processors, designed to exploit instruction-level parallelism within a single thread of execution, for dependability-enhancing objectives. Consequently, fault monitoring mechanisms can be automatically incorporated without requiring specialized hardware and without incurring a significant performance penalty. Experimental results using the image processing functions from the SDIO Signal and Data Processing Benchmark Suite are used to demonstrate the compilation tool and the thread integration method.

7.1. Introduction

In many embedded real-time control systems, advances in integrated circuit and printed wiring board reliability have left external sensor and interconnection faults as major contributors to system failures [ADAC89]. Control systems can compensate for input faults through three forms of redundancy. Massive hardware redundancy [Crev56], also called spatial redundancy [Siew95], duplicates sensors and interconnections and provides logic
7.1. Introduction

to choose among the inputs. Temporal redundancy [Reyn76] involves repeating the sampling of a parameter. Analytical redundancy allows calculation of unmeasured parameters based on available information. The latter two approaches can be supported with a recovery block structure [HLMR74] to simplify design. Each of these approaches differs in detection latency, fault set coverage, cost, and processing requirements. Spatial redundancy is well suited to both transient and permanent faults, but increases system cost, size, weight, power and repair frequency. Temporal redundancy is applicable for brief transient faults in a system without tight timing requirements and with available resources. Analytical redundancy can compensate for both transient and permanent faults, but requires higher CPU performance. However, cost pressures typically limit the increase of CPU performance requirements, and preclude the use of these add-on redundancies to increase system dependability. Temporal and analytical redundancy both require additional computation, increasing detection latency or processor throughput requirements. This chapter presents a technique which reduces the amount of computation needed for these methods by merging threads of execution for efficient run-time performance.

Modern CPU microarchitectures exploit instruction-level parallelism (ILP) via the use of pipelined and superscalar machine designs to increase throughput, measured in average instructions per cycle (IPC). These ILP machine resources enable the CPU to speed up processing of parallelizable portions of the program but remain underutilized otherwise. Even with modern compilers most programs are not able to exploit all the CPU resources in every machine cycle due to the dynamic variation of the available parallelism in these programs during their execution. For example, even a highly aggressive superscalar pipelined processor such as the PowerPC 620 achieves an IPC of only 25% to 44% of its potential four instructions per cycle [DNS95]. The corollary of this low average utilization is that many ILP resources are underutilized and available for use by another thread of computation. In many applications, these same resources can be effectively used to enhance an embedded system’s dependability by executing application code that has embedded software redundancy for fault detection and compensation. Such techniques have been proposed and studied in the past [Schu91]; however no compilation tool has been implemented to automate this process. This chapter presents an automatic method of integrating such a thread of fault-tolerance-enhancing computation into existing embedded application programs. This method makes use of a post-pass assembly code compiler to efficiently merge the application thread with the monitoring thread into a functionally
7.2. Integration Methods

Previous research has developed promising secondary threads that can be integrated with the application threads to achieve certain attributes of dependability, e.g. control flow and input fault monitoring and recovery [Schu91]. Our focus is on the automatic and efficient integration of monitoring threads into application threads. The monitoring thread may provide control flow checking [YC80] [Lu82] [Namj92], algorithm-based fault tolerance [HA84], duplication, N-module redundancy with masking [vonN56] [Reyn76], and other techniques. The case of duplication, or temporal redundancy, is used to demonstrate the integration concept. The concept is then extended to provide algorithm-based fault-tolerance, showing how two arbitrary threads can be integrated into a single efficient composite thread to effectively achieve multithreading in a single-threaded machine.

7.2.1. Thread Enhancement through Duplication

The integration of threads increases the number of independent instructions, allowing the compiler to generate a more efficient instruction schedule and hence faster run-time performance. In addition, common control structures in the two threads can be merged, and
7.2. Integration Methods

duplicate instructions implementing them can be eliminated, further improving performance. Duplicated code doubles the number of independent instructions while using an identical control structure, providing an opportunity for improving execution speed. However, this also increases the system’s vulnerability to single-point failures.

We present an example based on a transverse image filtering function (*Trans*) to illustrate basic methods for duplicating and integrating threads. We do not include information on execution schedules or instruction latencies in order to simplify the explanation. However, the code transformation tool which integrates these threads generates cycle-accurate schedules, monitors loop iteration variables and evaluates cost trade-offs in order to derive efficient integrated code.

Figure 7.1 shows how idle CPU resources are used to speed the execution of the duplicated thread. In the baseline duplication case, function *trans* is run twice in order to detect errors and verify correct processor and memory performance. The results from each run are compared in a separate verification function. By merging the two runs of *trans*, the run time falls significantly while providing the same level of error detection.

![Fig. 7.1. Integrated thread execution](image-url)
The first step of the integration consists of duplicating the code of the procedure to be executed. Variables are then totally split (renamed to eliminate register reuse), which allows code to be rescheduled freely. Figure 7.2 a) shows a CDG which is generated for the code; code regions with identical control structures and no data dependencies are then merged to eliminate excess control instructions as seen in part b).

Figure 7.3 shows one example of this merging called loop jamming, in which two loops share the same iteration conditions and are data-independent, allowing the instructions from one loop body to be inserted into the other loop body. The control instructions from only one loop are needed, so those of the other loop are not included. After loop jamming, the code is scheduled for efficient execution on the target processor. Finally, registers are allocated to the variables to complete the integration process. The performance of the integrated code is shown in Section 7.3.

Some functions will benefit more from duplication and integration than others, in part because the potential performance gain from rescheduling instructions is a function of the
size of the control-equivalent instruction pool (i.e., instructions which execute under the same control conditions).

7.2.2. Thread Enhancement through Diversity

The concepts behind this integration can be extended to integrate separate, non-duplicate threads, increasing the generality of the technique. In this case, a function which determines the centroid of a bright image area (AC) is enhanced with a gamma particle detection function (Gamma). Figure 7.4 shows a portion of the CFGs and CDGs of the two functions, in particular the loops which scan across each row of the image. These two pairs of nested loops can be jammed together for simultaneous execution.

In Figure 7.5, the instructions of the loop bodies are placed to execute sequentially, but are further enhanced in Figure 7.6 by filling both sides of AC’s conditional with code from
7.2. Integration Methods

This particular optimization was driven by the free resources predicted in block C41_0 by the processor model.

Other such code transformations are possible to increase the amount of free resources in the host thread. For example, the amount of free resources in a thread can be increased by unrolling or peeling loops. In these transformations, copies of the loop body are duplicated in order to reduce looping overhead; each of these copies may yield additional free resources.

Not all applications are good candidates for thread integration. For example, merging the functions Background (which subtracts an expected background brightness value from the image) with Gamma and forcing them to operate on different images leads to excessive memory traffic and cache misses, reducing performance. Mismatches in thread characteristics can lower execution speed, increase memory requirements and render caches ineffective. Applications which can benefit from thread integration typically offer multiple concurrently operating independent threads, such as reading sensors, providing communication services, interacting with users, controlling actuators, and providing overall control system operation. Threads must share similar invocation and control flow behavior while operating on independent data. Finally, the target processor must offer sufficient resources such as functional units, memory bandwidth and registers to allow improved performance.

7.2.3. Tools and Evaluation Environment

A post-pass compilation software tool suite [NNS94] designed to exploit parallelism in uni- and multi-threaded applications has been enhanced to help integrate threads. This tool suite builds CDGs to represent programs, moves code based upon user guidance, schedules code for efficient execution on the target processor, and provides visualization sup-
7.3. Experimental Results

The tool suite also provides a machine model which by providing accurate execution timing allows for code scheduling.

The target processor chosen for execution is the DEC Alpha AXP 21064 microprocessor, a dual-issue superscalar processor from Digital Equipment Corporation implementing the Alpha instruction set architecture. All code is compiled and assembled with gcc at the highest level of optimization. All performance data are derived from on-chip embedded cycle counters during program execution on a DEC Alpha 3000/400 workstation.

The application code presented is part of the SDIO Signal and Data Processing benchmark suite [NRC91] which contains functions for digital processing of satellite images.

7.3. Experimental Results

Two types of integration are evaluated using functions from the benchmark suite. The first example presents duplication and integration of the function Trans, which performs transverse filtering on an image. The second example presents two integration cases, with Gamma integrated into AC, and Gamma integrated into Back. AC determines the centroid of an area of bright pixels in an image, Gamma filters out noise resulting from gamma particles striking the image sensor, and Back subtracts a predicted background brightness from the image. Performance comparisons use a base case of the cycle count of executing the two threads separately and sequentially, with an additional verification function in the case of thread duplication, as seen in Figure 7.1.

Two sets of tests are performed for Trans to evaluate the performance impact of code duplication and integration, shown in Table 7.1. In the first set, the same input data is used for both threads, representing the situation in which the data processing must be duplicated for on-line error detection. The second set of tests uses different input data sets, for cases in which the input data itself is being tested. The first set of tests requires less communication with memory and hence less performance is lost due to cache misses, resulting in a 9-10% performance improvement.
7.4. Conclusions

*Trans* benefits from thread integration through loop jamming, which eliminates certain control instructions by merging control structures. In addition, the larger pool of instructions in the loop body allows better scheduling, which further improves performance.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Data Set</th>
<th>Base Duplicate Cycle Count</th>
<th>Improvement with Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans</td>
<td>same</td>
<td>103 M</td>
<td>15.9%</td>
</tr>
<tr>
<td>Trans</td>
<td>different</td>
<td>103 M</td>
<td>6.9%</td>
</tr>
</tbody>
</table>

*AC* and *Gamma* are integrated to demonstrate the potential for merging separate control flow threads, with the results shown in Table 7.2. By running the two functions simultaneously, the execution time drops by over one-sixth, decreasing fault detection latency and computing requirements. The same input data set is used for both functions due to a one-image delay required for the *Gamma* function. Performance improves primarily because of loop jamming but also from instruction rescheduling given the larger pool of instructions. Merging *Back* and *Gamma* requires the two functions operate upon separate images, leading to increased memory bandwidth and hence more cache misses. A more sophisticated technique for integrating the threads could improve performance by enabling overlapping operations on the same image, but this is left as future work.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base Cycle Count</th>
<th>Improvement with Loop Jamming</th>
<th>Improvement with Loop Jamming and Rescheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC &amp; Gamma</td>
<td>8.99 M</td>
<td>16.0%</td>
<td>18.9%</td>
</tr>
<tr>
<td>Back &amp; Gamma</td>
<td>14.0 M</td>
<td>-7.9%</td>
<td>-6.4%</td>
</tr>
</tbody>
</table>

These results demonstrate the potential performance impact of software thread integration for function duplication and more general merging for performance enhancement under appropriate circumstances. Code for fault-tolerance often has a structure (e.g. duplication) which benefits from thread integration.

7.4. Conclusions

This chapter demonstrates the use of software thread integration to improve system reliability and performance. The work is especially relevant to embedded control systems.
with cost and reliability concerns as it reduces CPU requirements, simplifying implementation of low-cost software-based fault tolerance. The original application program is used as the host thread, while a fault monitoring and compensation thread is treated as the guest thread and is integrated into the application thread. This additional thread may be a duplicate of the original. The resulting integrated thread is functionally equivalent to the two original threads, yet reduces error detection latency by allowing interleaved execution of the threads. The integrated thread can also cut system run-time by improving instruction scheduling and reducing instruction counts.
Chapter 8

Summary and Future Work

This thesis introduces software thread integration (STI), a compiler technology which interleaves multiple threads of control into one, allowing embedded system designers to reap low-cost concurrency from general purpose, single instruction stream processors. STI's concurrency offers many benefits. System resources can be better utilized, as more instructions are available to be scheduled. Context switching can be eliminated, increasing system efficiency. Real-time software threads can be written to replace dedicated hardware and then integrated with existing application code, allowing efficient migration of functions from hardware to software. Error detecting code can be integrated to provide concurrent error detection.

We have focussed on migrating functions from hardware to software because of its many potential advantages. Replacing hardware with software puts more of a system’s cost on the downward curve described by Moore's Law. Eliminating hardware reduces system size, weight and possibly power requirements. Functions not available in hardware can be created in software. Requirements for on-chip microcontroller peripherals are reduced, enabling the use of a wider range of microcontrollers, possibly including less expensive ones. Using software to perform functions enables greater flexibility of design, in-place function upgrades and faster response to changing markets and standards.

There are two drawbacks to software-based solutions. First, the design produced may be quite inefficient, requiring significant CPU throughput to deal with program overhead such as switching among different function contexts. Although instruction processing is getting less expensive, it is not free. The competitive nature of the marketplace will reward the more computationally efficient and hence less expensive designs, all other things being equal. Second, software is expensive to write and maintain, which reduces its attractiveness. Programmer efficiency has risen thanks to modularity and abstractions offered by high-level and object-oriented styles of design and implementation. Unfortunately,
these methods reduce program efficiency and prevent the programmer from precisely controlling the processor as needed to migrate hardware functions to software. Despite the penalties, time and again one sees that embedded system designers are willing to forego programming efficiency in order to perform hardware to software migration (HSM) and reduce hardware costs.

STI offers savings for both non-recurring (design) and recurring (production) costs. Dramatic programming performance gains result from writing in high-level languages; STI gives this software the timing precision of assembly code, making it suitable for hardware to software migration. Automating the integration of the code shortens the design process, reducing design costs, and shortens debug times by eliminating many coding errors. Automated software tools also simplify debugging and maintaining code in systems which have been deployed. Finally, implementing functions in software reduces component counts, reducing recurring costs.

8.1. Contributions

This section describes the contributions of this research organized by chapter. It covers the thread integration code transformations, the Thrint tool, guest and host thread selection methods, guest triggering, suitability analysis of STI for a given embedded system, a high-temperature STI prototype and other potential applications for STI.

8.1.1. Code Transformations for Software Thread Integration

We present and demonstrate methods to integrate multiple individual threads, some of which may be real-time. The goal of thread integration is to produce a program with integrated real-time guest thread events which execute at the proper time. In existing systems, this event scheduling can be implemented through busy waiting (the CPU polls inputs to detect event occurrence and executes nops to pass time), or through interrupt-driven context switching (a timer or other signal source triggers execution of the real-time code). Thread integration replaces the filler instructions of the busy-wait version with instructions from the host function, enabling more efficient program execution. Guest events can be either singular or looping, which will determine in part how integration is performed. In some cases multiple techniques can be used to integrate an event, allowing optimization for execution speed or program memory.
8.1. Contributions

8.1.2. The *Thrint* Tool

Software’s flexibility is a blessing and a curse. “Sharp” tools enable a programmer to manage a design’s complexity and benefit from this flexibility while avoiding its traps. This makes it easier to develop and maintain software. Embedded and real-time constraints complicate software development, but tools have been slow to react for various reasons. Hardware to software migration is an area of real-time embedded systems which is valuable enough to perform manually despite its complexity and tedium.

We have created a “sharp” tool, *Thrint*, which automates most of the work in STI, making it much easier to design real-time embedded systems with HSM. *Thrint* operates on assembly code for timing accuracy, using a hierarchical program representation to simplify code manipulation. It uses static timing analysis to guide code transformations which interleave host and guest code into a single thread. It also provides various forms of graphical output to help the developer understand program behavior.

8.1.3. Guest Triggering and Host Selection

We develop and present run-time support techniques to recognize a trigger event and invoke a guest thread, taking into account timing constraints and unpredictable host function invocation. We develop methods (built into the tool *Thrint*) to analyze an application, finding good potential host functions through profiling and static timing analysis. This provides guidance to select threads to integrate and predicts the performance of an integrated application before performing the integration.

Integration often 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.

8.1.4. Constraints on Using STI for HSM

Successfully using STI as a design alternative for embedded systems requires understanding the constraints limiting its use and comparing them across all migration options. Each method has its own strengths, resulting in different “sweet spots” within the design space.
8.1. Contributions

The utility of implementing a real-time guest function in software using STI depends upon several system characteristics. First, the guest must offer sufficient idle time to be worth reclaiming, distributed with a fine enough granularity to make a context-switching solution too wasteful. Second, moving a function from hardware into software will increase the system’s processing throughput requirements, possibly forcing the use of a faster (and more expensive) processor and memory system. There is a bus speed wall beyond which performance gains become too expensive; this wall is much lower for most embedded systems than for desktop PCs (which operate in controlled environments). Finally, there must be host functions which execute with sufficient determinism to allow integration with guest functions. The functions must follow predictable control flow paths and the hardware upon which they run must execute instructions at predictable speeds. Within these constraints STI offers an automatic method to move real-time functions from dedicated peripheral hardware into software running on a generic microprocessor.

8.1.5. Experimental Prototype

We present an experimental implementation of a communication network protocol bridge for a high-temperature (185° C / 365° F) application. Software thread integration is used to interleave multiple real-time program threads for efficient concurrent execution, allowing an off-the-shelf, high-temperature, low-speed 8 bit microcontroller to translate messages between a CAN bus and a serial UART data link. The prototype system, which has been built and tested, demonstrates the feasibility of using STI in real-world applications.

8.1.6. STI for Concurrent Error Detection and Performance

The concurrency provided by software thread integration is well suited for fault-tolerant programs in embedded systems. A monitoring or checking thread can be integrated into application code to offer concurrent error detection, which reduces detection latency. This also allows the exploitation of idling or underutilized machine resources in modern superscalar processors, designed to exploit instruction-level parallelism within a single thread of execution, for dependability-enhancing objectives. Consequently, fault monitoring mechanisms can be automatically incorporated without requiring specialized hardware and without incurring a significant performance penalty.
8.2. Future Work

The STI concepts presented in this work encourage further investigations into HSM, reliability and program execution speed improvement. Many interesting research questions arise, such as:

- Is there a clean set of rules for integrating bit-banged protocol applications such as the HSCAN bridge? If so, the market which could benefit is large.

- Is there an automatic technique for identifying host work which can be restructured into queues as described in Chapter 4? Can the restructuring be performed automatically? This would expand the range of applications which can benefit from STI.

- What is the most practical way of verifying semantic and temporal correctness in an integrated system? The software is designed to be correct, but bugs have a way of appearing in any system.

- Can hardware timers be used to further improve system performance or reduce complexity? They are already useful as watchdog timers for deferred guest service.

- Is it worthwhile to use STI as an alternative “back-end” for an existing hardware/software codesign-cosynthesis tool chain? This would enable the use of standard processors, allowing the leveraging of existing CAD tool chains to target smaller-volume applications which don’t justify custom silicon or hardware.

- Can STI be used to accelerate programs running on the increasingly popular VLIW (Very Long Instruction Word) DSPs (Digital Signal Processors)? The telecommunications industry, for example, is growing quickly yet its RTES design constraints can be tight. And if future performance gains in high-end embedded processors come primarily from LIW (Long Instruction Word), VLIW or EPIC (Explicitly Parallel Instruction Computing), can STI be applied efficiently for them as well?

- Should concurrent error detection concepts be used to protect HSM-implemented peripheral functions? This might be necessary in space-borne applications, which would benefit dramatically from HSM’s cost, size, weight and reconfigurability benefits yet routinely suffer single-event upsets.

STI offers exciting possibilities for automating the difficult parts of HSM and improving both the embedded system product and development process. The product is improved by eliminating hardware and adding flexibility. Cutting hardware improves cost size, weight, power and reliability. Implementing functions in software improves time to market, sim-
plifies upgrades and enables customization. The process of developing software is improved by enabling the system developers and maintainers to work in a high-level language while maintaining the timing accuracy of assembly language.

We look forward to making software thread integration a standard method for hardware-to-software migration in industry, giving embedded system designers a “sharp” tool which lets them work efficiently on the interesting parts of system design and implementation without wasting time on the tedious parts.

8.3. Publications

References

[ADAC89] Die ADAC Pannenstatistik, Motorwelt (5), 1989


[Beac95] Bill Beacham, Hamilton Standard (United Technologies Corp.), personal communication, March 1995


<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
</table>


[Geor97] Scott George, “HC05 Software-Driven Asynchronous Serial Communication Techniques Using the MC68HC705J1A,” Motorola Semiconductor Application Note AN1240

[Glen] Mark Glenewinkel, “Interfacing the MC68HC705J1A to 9356/9366 EE-PROMs,” Motorola Semiconductor Application Note AN1241


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Hita2]</td>
<td>Hitachi Corp., <em>HD61830/HD61830B LCDC (LCD Timing Controller) Data Sheet</em></td>
</tr>
<tr>
<td>[Hone98b]</td>
<td>High Temperature 32Kx8 Static RAM Datasheet, Honeywell, Inc., April 1998</td>
</tr>
<tr>
<td>[IDC00]</td>
<td>IDC, Personal Computer Sales Analysis, 2000</td>
</tr>
</tbody>
</table>

[Nauf] Naji Naufel, “Interfacing the 68HC05C5 SIOP to an I²C Peripheral,” Motorola Semiconductor Application Note AN1066


[Phil95a] Philips Semiconductor, The I²C Bus and How to Use It (including specifications), 1995

[Phil95b] Philips Semiconductors, Databook IC20: 80C51-Based 8-Bit Microcontrollers, P8XC592 Data, January 1995, p. 1200

[Phil96] Philips Semiconductor, 80C51 Family Hardware Description, August 1996


<table>
<thead>
<tr>
<th>Reference</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Rowe94]</td>
<td>Lawrence A. Rowe, “Continuous Media Applications,” <em>Multipoint Workshop</em> held in conjunction with <em>ACM Multimedia '94</em>, San Francisco, CA, Nov. 1994</td>
</tr>
<tr>
<td>[Scen99]</td>
<td>Scenix Semiconductor Inc., “SX18AC75/SX20AC75/SX28AC75/SX28AC100 High-Performance 8-Bit Microcontrollers with EE/Flash Program Memory and In-System Programming Capability,” June 16 1999</td>
</tr>
<tr>
<td>[Topp]</td>
<td>Peter Topping, “An RDS Decoder using the MC68HC05E0,” Motorola Semiconductor Application Note AN460</td>
</tr>
</tbody>
</table>