1. Introduction

Software Thread Integration (STI) [1] and Asynchronous STI (ASTI) [2] are compiler techniques which interleave functions from separate program threads at the assembly language level, creating implicitly multithreaded functions which provide low-cost concurrency on generic hardware. This extends the reach of software and reduces the need to rely upon dedicated hardware. STI and ASTI are driven by two types of timing requirements: thread-level (e.g. the delay between an event occurring and a service thread running) and instruction-level (e.g. when a specific instruction or code region must begin executing relative to the start of the function or another such instruction or region). These coarse- and fine-grain approach provide a precise method of defining timing requirements. STI provides synchronous thread progress; both functions proceed lock-step. ASTI provides asynchronous (independent) thread progress through the use of lightweight context switches (coroutine calls) between primary and secondary threads. The primary thread has hard-real-time constraints, while the secondary thread is not real-time, or has much longer deadlines.

We assume that instructions take a predictable number of cycles to execute. This implies a straightforward instruction execution pipeline (if used) and a predictable memory system (e.g. the cache is locked, software managed, or not present). These requirements are met for the processors we target: 8 and 16 bit microcontrollers. We target applications with only one hard real-time thread (the primary thread, used for the communication protocol), although recent extensions to STI [3] support multiple hard-real-time primary threads. We have implemented a thread-integrating compiler Thrint which implements many of these analyses and transformations for the AVR architecture, which is 8-bit, load/store, and optimized for embedded C code.

2. STI

STI targets functions which are independent, data ready and can run to completion. This may require buffering work for one function to ensure it is ready to run when the other is released. For example, in a video controller, graphics rendering work is buffered to be available whenever a periodic scan line interrupt service routine runs [3]. STI works by merging two functions into one implicitly multithreaded function, as shown in Figure 1. When used for real-time software, it enables the placement of time-critical instructions from one thread so they execute at a specific time relative to the beginning of the integrated function, regardless of the control or data flow characteristics of either thread.

STI begins with building a control-dependence graph representation of the functions to be integrated. We partition the register file between threads at compile time, but more sophisticated register allocation is possible. The best-and worst-case timing of the code is derived statically. This timing is then regularized; execution paths of uneven duration are padded to last the same amount of time (nops and nop loops are used for space efficiency).

Code from one thread can now be moved to execute at given time in the other, using code transformations such as motion, replication, and various for loops (peeling, splitting, guarding, unrolling and fusion). Integration copies code into each path of control flow in a location which ensures it executes at the correct times regardless of the thread progress. Integration involves control-dependence graph traversal (and is hierarchical), and nested transformations are cumulative. Non-looping primary code regions are handled individually. Moving a region into a conditional requires replicating it into both sides, while entering a loop
execute for each segment. After padding to equalize timing of conditionals and loops modulo $T_{\text{SegmentIdle}}$, the entire secondary thread is broken into segments of duration $T_{\text{SegmentIdle}}$. Intervening primary code within the idle time window is removed and integrated into each segment of the secondary thread, ensuring that running any segment of the secondary thread will still result in the proper intervening primary code executing at the correct times. Coroutine calls are integrated in the secondary thread to ensure it yields control back to the primary thread at the end of the segment, just before the available idle time expires.

4. Applications

In the STIGLitz project [3], STI enables an inexpensive 20 MHz AVR 8-bit microcontroller to generate monochrome NTSC video while servicing a high-speed (115.2 kbaud) serial communication link. This system offers graphics rendering speed-ups of 3.99x to 13.5x by recovering fine-grain idle time in the display refresh function and using it for video rendering.

We used ASTI to create a bridge between an RS232 link and a software-implemented J1850 automotive embedded network protocol controller [2]. Compared to an interrupt-based approach, ASTI results in a secondary thread speed-ups of 1.56x (message reception) and 1.83x (transmission), enabling the use of an 8 MHz AVR 8-bit microcontroller. An interrupt-based approach would require a 14.7 MHz clock to provide the same performance.

5. Conclusion

STI and ASTI extend the performance of commodity low-end microcontrollers through static scheduling techniques, potentially avoiding the need for dedicated hardware or faster clock speeds.

References

