Abstract

Embedded systems require control of many concurrent real-time activities, leading to system designs which feature multiple hardware peripherals with each providing a specific, dedicated service. These peripherals increase system size, cost, weight, power and design time. Software thread integration (STI) provides low-cost thread concurrency on general-purpose processors by automatically interleaving multiple (potentially real-time) threads of control into one. This simplifies hardware to software migration (which eliminates dedicated hardware) and can help embedded system designers meet design constraints.

This paper introduces automated methods for planning and performing the code transformations needed for integration of functions with more sophisticated control flows than in previous work. We demonstrate the methods by using Thrint, our post-pass thread-integrating compiler, to automatically integrate multiple threads for a sample real-time embedded system with fine-grain concurrency. The sample application generates an NTSC monochrome video signal (sending out a stream of pixels to a video DAC) with STI to replace a video refresh controller IC. Using Thrint reduces integration time from days to minutes and, by reclaiming idle time, speeds up graphics rendering by 3.2x to 3.8x in the three functions examined.

1. Introduction

Embedded systems have multiple concurrent activities which must meet their deadlines or else the system will fail. These activities are usually implemented in hardware to guarantee they occur on time, as most microprocessors suffer when trying to perform multiple threads concurrently at a fine grain while meeting deadlines. Adding this hardware complicates system design whether added as external ICs or as modules on a microcontroller or system-on-chip. External components increase system size, weight, power, parts cost and design time. Integrated hardware peripherals increase design time and also fracture the chipmaker’s market (which leads to increased cost through reduced volumes). In the end, both on-chip and external hardware solutions increase costs.

1.1 Hardware to Software Migration Challenges

These costs have led to many efforts to implement the concurrent activities in software in order to ride the wave of falling compute costs driven by Moore’s law. There are two difficulties with making generic microprocessors adept at executing multiple concurrent threads.

First, the processor must switch easily among contexts, saving and restoring registers with each switch. There are many techniques (register banks and windows, coarse- and fine-grained multithreading, simultaneous multithreading, multiprocessing) to allow quick switches [14][22][21][4][20][3]. Some of these techniques are available in embedded processors, though not all. However, if any sort of operating system or scheduler is used to switch threads, it will incur a penalty of tens or hundreds of cycles (if not more) as it executes. This prevents sharing the processor on a time scale of less than twice this delay, limiting the effective concurrency of the processor (as measured in context switches per second).

Second, the processor must execute the right instructions from the right thread at the right time. The general solution is to divide a thread into coarse-grain pieces wherever idle time significantly exceeds double the context-switching delay. Each coarse piece is made of concatenated fine-grain pieces. Scheduling the fine-grain pieces is done statically (at compile time) using padding instructions (e.g. nops) to generate a given time delay. Although some coders have painstakingly managed to inject by hand instructions which perform useful work for another part of the program, the resulting programs are brittle and difficult to maintain. Furthermore, this approach is very poorly suited to systems which require frequent real-time activity with accurate timing. The double context switch time mentioned above determines the level of granularity at which the coder must switch from coarse-grain thread pieces (which allow the
processor to be shared) to padded and concatenated fine-grain thread pieces (which monopolize the processor).

Despite these difficulties, there is an abundance of articles and application notes from makers of microcontrollers describing how to extract concurrency from their generic processors (see [9] for examples). These efforts primarily target two classes of applications: video signal generators and communication protocol controllers. This paper is in the first area. The second area is even more demanding, and we are currently extending our STI concepts to support it. STI support for this latter area opens the door to low-cost implementations of application-specific protocols with optimized power, medium access delay, error detection, reliability or other characteristics.

1.2 HSM with STI

We have developed and continue to enhance our compiler-based approach to providing fine-grain concurrency. We have developed a compiler Thrift which automatically creates an implicitly multithreaded function from two functions, one with real-time requirements on specific instructions. It implements many of the time-driven code transformations which we developed for integrating threads while maintaining control, data and timing correctness. With this technology an off-the-shelf uniprocessor can efficiently perform multiple concurrent functions without any special support for rapid context switching, scheduling or concurrency. This in turn makes hardware to software migration (HSM) viable for general-purpose processors by eliminating much of the run-time overhead of context-switching or nop-based static scheduling.

1.2.1 Design Space Target. HSM with STI is an implementation method which helps some applications more than others. It is important to note that although STI isn’t useful for every class of application, it can be quite helpful for a subset. In particular, it excels for cost-constrained applications with deferrable work and fine-grain concurrency. In this paper we target one such application -- video generation.

A common misconception is that the timing variability of modern high-performance CPUs and memory hierarchies eliminates the temporal determinism which STI requires. This is a non-issue; STI targets applications which neither need nor can afford these CPUs and memory systems. For perspective, in 1999 81% of the 5.3 billion microprocessors sold were four- and eight-bit units (9% were 16-bit). These MCUs run applications which are not computationally intensive, and do not need more parallelism or faster clock rates. They lack sophisticated microarchitectures and memory systems. Instead these applications are constrained by other issues such as functionality, cost, power dissipation, design time and use of commercial off-the-shelf products.

HSM with STI allows the designer to address these issues efficiently.

The fact that microarchitectural features such as superscalar execution and memory caches complicate the static timing analysis upon which STI relies is irrelevant for these applications, as they do not need the performance provided by these features. In fact, often these applications cannot afford the additional cost of such an enhanced processor.

We target applications with only one hard-real-time thread running at a given time. All other threads must accept being potentially delayed until the completion of an integrated thread. This limits the composability of real-time modules to a single hard real-time thread, which is adequate for many low-end embedded applications.

1.2.2 Paper Overview. Our past work developed concepts and methods not only for software thread integration (STI) but also how to use it for HSM, in which a real-time guest thread replaces the dedicated hardware and is integrated with one or more host threads from the application [6][7][5][9].

In this paper we present the algorithms used to plan and perform integration, using previously developed integration transformations as building blocks. These methods are essential to make STI practical, as manual integration is tedious and error prone. Furthermore, they enable the automatic integration of code with overlapping loops, which are present in most non-trivial code. We use two phases to enable the evaluation of a variety of integration plans using estimates of MIPS recovered, guest response latency and code expansion. This enables the compiler or the designer to select the best approach for integration. An NTSC video refresh application is used to demonstrate the automated use of the integration planning and execution methods presented here. We use our compiler Thrift for the integration; it provides static analysis of control- and data-flow and timing, code transformation and visualization. We examine an application and use Thrift to perform the video refresh work, creating efficient integrated functions.

This paper has the following organization. Section 2 gives an overview of how STI works. Section 3 introduces the planning and transformation algorithms and data structures used in Thrift. Section 4 examines the integration of a sample application. Section 5 summarizes the results and their broader implications.

2. Software Thread Integration Overview

Figure 1 presents an overview of how STI works. A hardware function is replaced with software written by a programmer. This code consists of one or more guest threads (represented by the black bar) with real-time requirements. When the threads are scheduled for execution on a sufficiently fast CPU, gaps will appear in the schedule...
of guest instructions, as illustrated by the white gaps in the black bar. These gaps are pieces of idle time which can be reclaimed to perform useful host work. STI recovers fine-grain idle time efficiently and automatically.

STI uses a control dependence graph (CDG, a subset of the program dependence graph [1][10][12][13][16][17][19]) to represent each procedure in a program. In this hierarchical graph (please see Figure 2), 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 uses a control dependence graph (CDG, a subset of the program dependence graph [1][10][12][13][16][17][19]) to represent each procedure in a program. In this hierarchical graph (please see Figure 2), 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.

Using STI for HSM involves moving guest code into the correct position within the host code for execution at the correct time. The first stage in this code motion is called degenerate integration; the programmer manually appends the guest procedure code to the end of the host procedures. The resulting procedure is then automatically integrated by moving guest nodes left in the CDG to locations which correspond to the target time ranges. A tight target time range may fall completely within a host node, forcing movement down into that node or its subgraph.

Figure 2 shows a graphical sample of previously developed CDG transformations [7][8][9] which can be applied repeatedly and hierarchically, enabling code motion into a variety of nested control structures. For example, moving a single guest event node into a host code node requires splitting the code node (a basic block). This is shown in the diagram as single event case b. Moving into a conditional (a predicate) requires guest replication into each case (single event case c) as well as potential padding to equalize case durations. Moving into a loop requires loop splitting (single event case d1) or guarded execution (single event case d2) on a specific iteration. The transformations also support the integration of guest loops with host loops. Loop fusion, guarding and unrolling are used to match the host loop’s work with the available idle time within one or more guest iterations. For more detail please consult [9]. Most of the transformations are implemented in our thread integrating compiler Thrint.

STI automatically ensures semantic and timing correctness with its transformations. The variety of integration methods and decisions enable Thrint to 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. We limit integration to threads without true, output or anti-dependencies between themselves. This is the case for threads with disjoint data-flow graphs, which result from separate data sets. STI only needs to handle false data dependences when integrating threads; no other data dependency issues arise because each individual thread remains in order internally. 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. To date, the 32 registers of the Alpha architecture have been adequate for the two threads simultaneously, eliminating the need for spilling. The Alpha architecture does not use condition code flags; if it did code motion would need to take them into account.

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 [15][18], with best and worst cases predicted. Hardware and software both conspire
to make this a difficult problem in the general case. However, we focus on applications without recursion or dynamic function calls, and processors without superscalar execution, virtual memory or variable latency instructions. We assume locked caches or fast on-chip memory and no pipelining. As mentioned previously, these restrictions have little impact on the design space targeted.

### Single Event Integration

**Key**

- Code
- Loop
- Predicate
- Padding Ovhd.
- Generic
- Guard
- Int. Loop Test

To change the node with minimum error time

Unacceptable error time: b,c,d

Acceptable error time: a

**Code Transformations**

- **b. Best node is code**
  - Split code node and retry
  - Next pass will move node

- **c. Best node is predicate**
  - Retry for each condition with predicate as parent
  - Future passes will copy and move node

- **d. Best node is loop node**
  - Option 1: Split loop
  - Code memory overhead
  - Next pass will move node
  - Option 2: Guard guest in loop
  - Run-time overhead

**Figure 2. Examples of Single and Looping Event Code Transformations for STI**

During integration, timing directives (supplied by the programmer) guide integration. Timing jitter from uneven predicates in the host thread is automatically reduced (using padding instructions) 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 allows the tailoring of STI transformations to a particular embedded system’s constraints.
3. Planning and Performing Integration

Integration requires several stages of preparation. Time-critical guest code is identified based upon user directives, the host code’s execution schedule is predicted for both best and worst cases, and temporally deterministic segments within the host are identified as targets for integration (as described in [8]). Next, integration planning takes place. Thrint plans the integration transformations needed to integrate the guest function with each of the temporally deterministic segments identified previously. The guest code is then integrated with one or more of these temporally deterministic segments.

The guest thread consists of nodes (code, loop, predicate), some of which may have timing requirements associated with them (code and loop). These are sub-thread timing requirements; thread-level timing requirements are dealt with elsewhere. We use timing directives to specify the target time (with a user-defined tolerance) for the start of the node’s execution, as represented as a delay from the beginning of the integrated thread’s execution. We call guest nodes with these timing requirements explicitly specified guests (or simply explicit guests); the other guest nodes are called implicit guests. Thread integration must place the explicit guests in the host code based on the timing directives, while the implicit guests are merely constrained to be moved to ensure in-order execution between the explicit guests.

The integration requirements are defined in a text file as shown in Figure 3. This file specifies when basic blocks must execute and with what timing tolerance. For example, the four Video_PixN basic blocks each output a pixel of data to the display and must be integrated into the loop at the correct location, based on execution time. The integration requirements are loaded into a data structure which duplicates the CDG structure of the explicit guest nodes, as shown in Figure 4.

Each integration directive node holds a pointer to its explicit guest node, as well as a list of pointers to the implicit guests which precede and follow the explicit guest. Currently we limit explicit guests to be code or loop nodes which are at the first or second level of the CDG. We have found this to be adequate for a variety of applications. An implicit guest may be arbitrary code, provided that it is structured.

At this point integration planning begins, using the temporally deterministic segments [8] identified elsewhere (each of these is a contiguous subgraph of the host CDG).

The algorithm Plan_Integration (Figure 5) is used to identify which transformations are needed to integrate the guest code with the host segment. Plan_Integration creates an integration plan based on the integration directives data structure created previously, and then steps through each explicit guest within it.

Figure 3. DrawSprite integration directives file specifies timing requirements for guest thread components

Figure 4. Initial DrawLine control dependence graph with integration directives data structure marking explicit guests

Figure 5. Plan Integration finds hosts, planning loop transformations as needed to fuse loops
Integration for code guest nodes is handled by calling Find_Hosts (Figure 6). This algorithm identifies which node(s) will be executing during the guest’s target time range, and determines which transformations (previously presented in Figure 2) are needed to ensure that if the guest is placed there, control-flow and timing requirements are met. This may involve determining where to split a loop or how many padding nops to use for balancing a predicate node.

Figure 7 shows which target host nodes are identified for the example presented later in this paper.

Figure 8. Diagram showing example of loop integration planning with fusion, splitting and peeling

Looping guest events require a more sophisticated approach, which is also in Figure 5 (an example is graphically presented in Figure 8). The technique attempts to perform loop fusion if a guest and host loop overlap for multiple iterations. The goal of this loop fusion (as seen in Figure 2) is to match guest loop idle time with host loop body work through unrolling.

Figure 9. Integrate algorithm implements code transformations planned previously

Guarded clean-up loops can be added following the fused loop body to accommodate extra iterations or unknown
loop counts. The portions of loops which do not overlap are handled differently depending upon type. Host loops are split to separate the overlapping and non-overlapping iterations. Guest loops have iterations peeled off, with each explicit guest in the unrolled iteration integrated as a non-looping guest node.

This completes the planning for integration, allowing the evaluation of interesting evaluation plans to trade off code memory expansion for increased performance. One or more such integration plans may be selected for actual integration, as described in [9].

Integration, presented in Figure 9, performs padding, loop splitting, unrolling and other transformations previously planned and then copies the guest nodes to the appropriate locations in the host code. Note that each explicit guest may be assigned multiple hosts, and each explicit guest may have multiple implicit guests.

4. STI for a Video Application

Our previous work has developed concepts, code transformations and analytical methods for performing STI especially for HSM. Previous thread integration results reflect manually integrated code and a mix of manual and automatic analysis. In this paper we demonstrate automatic thread integration using our post-pass compiler Thrint, which implements automatic thread analysis, visualization, and integration by using techniques of static control-flow, data-flow and timing analysis, code transformations and register reallocation. We first examine the application, then evaluate idle time within the guest thread and temporal determinacy within the host threads. We then analyze automatically integrated code for system efficiency and memory expansion.

4.1 Target for Hardware to Software Migration

To demonstrate the benefits of STI for HSM we use an NTSC video refresh controller application (for driving a CRT). We replace a video generator chip with a software version. The processor must generate an NTSC-compatible monochrome video signal [11], summarized in Figure 10.

Although the beam scans 525 times per frame (in two interlaced passes (fields) per 33.3 ms frame), only 494 rows are visible and require video data, corresponding to 75.8% of the processor’s time. There are additional features in a video signal (vertical sync, serration and equalization pulses) but these can be generated easily with standard methods (ISRs triggered by an on-chip timer) so we do not examine them in this work. The video data portion of the signal is the most demanding, as a pixel of video data must be generated every 100 ns (for 512 pixels per row). With a 100 MHz CPU this corresponds to ten clock cycles per pixel, which is very frequent and offers little time for context switching or scheduling.

![Figure 11. Hardware architecture of system lacks a video refresh controller](image)

We target a 32 bit scalar RISC processor running at 100 MHz with on-chip single-cycle memory access and instruction execution and no virtual memory. The processor executes 100 million instructions per second. The hardware is structured as shown in Figure 11.

![Figure 12. Video data flow overview](image)
display refresh code. The queues hold parameters for drawing lines, circles or sprites and are fed by other functions in the application. The ISR selects one of the three integrated functions (if data is present in the queue) or else a dedicated busy-wait refresh function. The chosen thread then reads video data from the frame buffer in memory and sends it out to the CRT through the DACs.

The ISR and the queues are not implemented for this paper because they are straightforward to implement and analyze. Instead we focus on the integration and analysis of the refresh/render threads which are integrated by the compiler.

4.2 Experimental Method

Our compiler Thrint processes functions compiled for the Alpha instruction set architecture. Although it is not representative of most embedded systems, it was chosen to leverage the compiler PCOM from another tool suite (Pedigree [16]). The Alpha ISA is a clean load-store architecture with an ample register set and is a suitable target for this work. We assume a microarchitecture with easily predicted performance: scalar execution, single-cycle memory system or lockable cache, a predictable pipeline and no virtual memory. As explained in the introduction, the bulk of the applications targeted by this research neither need nor can afford the high throughputs provided by sophisticated and complex microarchitectures.

The guest (VidRef) and host functions (DrawLine [2], DrawSprite and DrawCircle) are written in C++ and used for initial degenerate integration (the guest function body is concatenated with the host function body, and automatic variables are added). The new functions are compiled with gcc 2.7 with -O1 optimization. Basic block labels are added to the resulting assembly language functions to identify instructions with specific real-time requirements. The functions are then processed by PCOM into CDG-structured assembly language. These functions are analyzed and integrated by Thrint, which creates an output assembler file as well as visualization support files (e.g. Figure 14). Data symbol information is added to the assembler file (after having been deleted during processing) and then assembled. The object file is linked with an X-windows-based driver program to allow execution-based verification of correct program operation. Timing correctness is verified by static timing analysis in Thrint after all code transformations have been completed. As the target machine architecture is highly predictable (completely predictable pipeline, scalar instruction execution, single-cycle memory access), timing verification through execution or simulation is not performed.

4.3 Guest Thread

The video signal has events which must occur within tight time ranges to create a compatible video signal. As mentioned previously, we assume an interrupt service routine triggered by a programmable timer generates the signal transitions needed for the horizontal and vertical sync, front and back porches, and equalization and serration pulses.

![Figure 13. Idle time in video signal generation software is mostly fine-grain (under 10 cycles long)](image)

The video data is read out of a frame buffer and sent out through an 8 bit DAC by the previously mentioned VidRef function. When the software needed to perform these events is executed by the 100 MHz processor (without any context switching or scheduling overhead considered), the resulting idle time is distributed as shown in Figure 13.

The idle time in the large bubbles (compared with the overhead of setting up a timer and performing two context switches (e.g. 30 cycles)) is best recovered through context switching. The idle time in smaller bubbles is recovered through STI. Figure 13 shows that over half the processor’s idle time for this video refresh application is in fine-grain pieces (four, seven and eight cycles long), making this type of application a good fit for HSM using STI.

4.4 Host Threads

![Figure 14. Initial DrawLine control dependence graph](image)

Figure 14 shows the control structure of DrawLine after the guest code has been appended and the file assembled. DrawLine takes two endpoints and a color code as argu-
ments and scan converts the line into the frame buffer. Two conditionals (predicates) in the beginning of the function determine line direction, then the code determines increment values and finally a loop sets pixels in the frame buffer. The conditionals within the loop selectively update x or y counters and error variables. The guest code consists of pointer initialization code and then a loop which loads 32-bit words from memory and sends them out to the video display one byte at a time.

The overall control structure for DrawSprite and DrawCircle with the guest code appended is similar. DrawSprite takes a pointer to a sprite (a 16x16 pixel array) with a position and draws the sprite in the frame buffer. The code consists of a loop which iterates across sprite rows, and conditionals within that body which handle various position cases of the sprite. DrawCircle scan converts a circle given center and radius.

4.5 Integration Process

The code is prepared for integration by marking time-sensitive instructions in the guest thread assembly code (called explicit guests) with labels, and then specifying timing requirements for those basic blocks. Figure 3 presents the integration directives file used for integrating DrawSprite. A timing error tolerance of 0 cycles is specified, so Thrint pads away all timing jitter (leading to increased code size). Only explicit guests (nodes with specific timing requirements) need to be defined in this file; the intervening nodes are called implicit guests and are automatically handled. At this point Thrint is run to perform integration.

Figure 15 shows the CDGs of the integrated function; DrawLine’s host loop is fused with the video refresh loop, and guest code is replicated into one host conditional. The loop control tests are fused with a logical AND to control loop execution. After the fused loop finishes, a guarded dedicated guest loop completes any remaining video refresh work. It is guarded to keep it from running if there is no more work. Following that loop is a guarded replica of the host loop to finish drawing long lines which were not completed in the fused loop. DrawSprite and DrawCircle have similar CDGs, with a fused loop followed by two clean-up loops.

4.6 Performance

Using STI to reclaim the idle time within the video refresh portion enables the system to perform more useful work such as line, sprite or circle drawing. Figure 16 shows graphics rendering throughput (drawing speed) of the processor using the discrete and integrated versions of these functions. The discrete version uses nops to fill the idle time within the video refresh function. Integration reclaims most of this idle time and more than triples the rendering speed for each function. This remarkable performance increase reflects the large amount of CPU time wasted when performing discrete video refresh and the efficiency of STI’s reclamation of the idle time.

![Figure 15. Integrated DrawLine CDG](image)

![Figure 16. Integrated threads reclaim CPU for useful work, improving system performance by more than 3x](image)
4.7 Code Expansion

Figure 17 shows how STI affects function size for the examples used. DrawLine grows by 96% (from 144 to 282 instructions), DrawSprite grows by 211% (from 341 to 1062 instructions) and DrawCircle grows by 153%. The bulk of the code expansion for DrawLine() (57 instructions) comes from padding to equalize timing variations among paths and to statically schedule the dedicated guest clean-up loop. Next are 47 instructions from splitting the host loop (marked “Replicated Host” in the graph). The guest requires some replication into conditionals, adding 18 instructions.

Next, the guest loop is unrolled three times, and much of that code must be replicated into each path of the three-conditional deep host loop. In addition, many padding instructions are added to statically schedule the guest code and reduce timing variations in the host code. The combination of deep conditionals and loop unrolling leads to code explosion; it would be logical to examine the DrawSprite function and replace the conditionals (which allow sprites to be drawn at positions unaligned with word boundaries) with computation. DrawCircle grows because of loop unrolling, replication and padding.

This code expansion only applies to functions which are integrated. In a typical application there would only be a few, so the overall impact on code size would be slight.

5. Conclusions

We have developed an automated solution to the hardware-to-software migration challenge called software thread integration. Our compiler Thrint automatically integrates multiple threads into a single highly efficient implicitly multithreaded flow of control which executes on a standard uniprocessors without special support for scheduling or fast context switching. We have also developed design methods for using software thread integration to perform hardware to software migration quickly and efficiently.

In this paper we present the data structures and algorithms needed to plan and perform software thread integration, using the transformations developed in previous work. These methods have been implemented in our research compiler Thrint. We demonstrate the results of automatic integration of a sample application with fine-grain concurrency and analyze the resulting code expansion.

We use Thrint and STI to replace a video refresh controller with a software implementation. We reduce integration time from days to minutes, paying a minor penalty in memory size while boosting performance by more than 3x by reclaiming large amounts of processor idle time.

Acknowledgments

The author thanks the many reviewers for their helpful suggestions.

6. References