Uniprocessor Performance Enhancement through Adaptive Clock Frequency Control

Augustus K. Uht, *Member*, *IEEE*

Abstract—Uniprocessor designs have always assumed worst-case operating conditions to set the operating clock frequency and, hence, performance. However, much more performance can be obtained under typical operating conditions through experimentation, but such increased frequency operation is subject to the possibility of system failure and, hence, data loss/corruption. Further, mobile CPUs such as those in cell phones/internet browsers do not adapt to their current surroundings (varying temperature conditions, etc.) so as to increase or decrease operating frequency to maximize performance and/or allow operation under extreme conditions. We present a digital hardware design technique realizing adaptive clock-frequency performance-enhancing digital hardware; the technique can be tuned to approximate performance maximization. The cost is low and the design is straightforward. Experiments are presented evaluating such a design in a pipelined uniprocessor realized in a Field Programmable Gate Array (FPGA).

Index Terms—Hardware, emerging technologies, better-than-worst-case performance, adaptive system design, computer design.

1 Introduction and Background

Ever since synchronous digital systems were first proposed, it has been necessary to make the operating frequency of a system much less than necessary in typical situations to ensure that the system operates correctly assuming worst-case conditions, both operating and manufacturing. The basic clock period of the system is padded with a guard band of extra time to cover extreme conditions. There are three sources of time variation requiring the guard band. First, the manufacturing process has variations which can lead to devices having greater delay than the norm. Second, adverse operating conditions such as temperature and humidity extremes can lead to greater device delays. Last, one must allow for the data applied to the system to take the worst delay path through the logic.

However, none of these extremes is likely to be present in typical operating conditions. The only known method to still obtain typical delays in all cases is to change the basic model to an *asynchronous* model of operation [6]. But, this is undesirable: Asynchronous systems are notoriously hard to design and there are few automated design aids available for asynchronous systems.

This paper proposes a *Timing Error Avoidance* technique (*TEAtime*) to realize typical delays using standard synchronous design methodologies; the typical-delay operation achieves better-than-worst-case performance. TEAtime uses a test circuit composed of a delay-augmented 1-bit wide version of the system's critical path(s). The system's clock frequency is continuously increased until a failure occurs in the test circuit, but before a failure can occur in the real logic; the frequency is then reduced. TEAtime is thus able to take advantage of typical operating conditions and realize substantially higher-than-normal performance. The extra cost is small. The technique is applicable to any synchronous digital system. Correct results are ensured if the design guidelines are followed. Neither the base cycle time nor the cycle count are affected by TEAtime. It is also easy to modify current designs to take advantage of TEAtime.

In order to demonstrate TEAtime's capabilities and correct operation, we implemented a simple CPU and memory on a Xilinx FPGA (Field Programmable Gate Array) and ran it under various operating conditions. Over a wide range of temperatures, TEAtime demonstrated performance improvements of about 34 percent over the baseline machine's worst-case specified performance. TEAtime adapted automatically to changing conditions, always stabilizing to a steady operating clock frequency.

The remainder of this paper is organized as follows: Related work is reviewed in Section 2. In Section 3, the basic ideas of timing error avoidance are presented, using our test CPU as a case study. Our experimental methodology is described in Section 4, with the experimental results presented in Section 5. We conclude in Section 6.

2 Related Work

There has been prior work somewhat similar to ours, but nothing that encompasses all of the attributes of our technique or actually demonstrates its functioning and characteristics with a real prototype. The closest work we are aware of is [17]. In this work, a microcontroller has been modified so that it can self-tune its clock for "maximum" frequency. It does this by periodically pausing computation for up to 68 cycles, during which time it forces extreme inputs (1 and all 1s) into the ALU. (The ALU has the longest critical path.) The output of the adder is checked: If it is correct, the frequency is increased; if incorrect, the frequency is decreased by a safety margin, at which time the computation resumes. This scheme takes advantage of

• *The author is with the Department of Electrical and Computer Engineering, Microarchitecture Research Institute, University of Rhode Island, 4 East Alumni Ave., Kingston, RI 02881.*
E-mail: uht@ele.uri.edu.

Manuscript received 19 Oct. 2003; revised 17 June 2004; accepted 18 Aug. 2004; published online 15 Dec. 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0181-1003.

some attributes of typical delays, but must pause operation to perform its tuning, reducing its performance gains. It also assumes that the critical (longest) delay path is through the adder, which is relatively easy to check; what if the critical path is through some other part of the circuitry?

In TEAtime, one or more copies of the worst-case path(s) through the system, similar to *replicates*, are used to determine whether the clock frequency should be increased or decreased. While replicates have been used before (see, for example, [11], [15], [22]), they have been primarily used to improve such things as power consumption for a given and fixed frequency, i.e., *dynamic voltage-scaling* in [11], [21], or to provide self-timing for high-speed subsystems [22]. Such methods are also more complex than TEAtime's. Last, it is not clear if the latter methods allow better-than-worst-case operation; certainly, it was not their aim with respect to improving a system's performance.

Another related approach in earlier work is to reduce overall energy consumption by operating a processor at lower than peak frequency during idle or less-demanding periods. It does this by determining the necessary frequency of a microprocessor to execute a given application in a given amount of time, then adjusting the supply voltage to achieve that frequency [3]; this reduces energy consumption, again via dynamic voltage scaling. While the work is impressive (the system was actually built on custom chips), it is not directly related to TEAtime: In [3], the maximum throughput remains the worst-case maximum throughput, so there is no improvement in performance, just energy efficiency. Further, the operating system is used both to determine the desired operating frequency and to control it. TEAtime operation requires no software inputs or interaction. Also in [3], a ring-oscillator is used as a substitute for the worst-case path. A ring oscillator does basically adapt to varying temperatures and process variations, but is an imperfect approximation to the worst-case path through a system, especially under a wide range of operating conditions. Further, the oscillator is open-loop; errors are not detected. It is also not easily expandable to handle multiple worst-case paths without reducing the largest performance possible (the safety margin must be greater than with a TEAtime approach).

There has been a large amount of work on asynchronous systems. See, for example, [13] for a description of the first asynchronous microprocessor and [6] for a brief tutorial on modern asynchronous circuit design. Such design techniques either use much more hardware than synchronous ones (*self-timed* circuits) or are very hard to design (*delay matching*) [20]. The latter is not helped by the dearth of robust design tools for asynchronous systems, although work is continuing. Attempts have been made to adapt existing synchronous tools for asynchronous system use, but with limited success [10].

There have been many methods created to improve the performance of synchronous circuits. The main approach is to *retime* [12] the registers or latches so as minimize the worst-case necessary clock period. This is done by a variety of methods, including moving the registers or latches in the circuit. Software pipelining has been applied to synchronous digital circuits to generate optimal clocking schemes [2]. However, worst-case delays between storage elements must still be maintained.

Multisynchronous systems [7] have also been proposed in which the circuitry on a chip is divided into semi-autonomous modules, each with its own clock. All of the clocks have the same frequency, but may be out of phase. This addresses part of the worst-case timing problem, but only at the system level, handling part of the chip clock drive problem. In the general case, *Globally-Asynchronous Locally-Synchronous Systems* (GALS) [4] have been widely studied. GALS methods could be employed in a TEAtime-based system to interconnect any necessarily fixed-frequency elements with the TEAtime-based elements. This would be particularly helpful in larger systems.

Wave pipelined arithmetic units have been proposed, but have implementation difficulties [5], [16], including the inability to easily stall the pipeline, since it depends on time-of-flight data storage (like a mercury delay-line). The design of such devices is also difficult; it is hard to ensure that signals arrive at the same time.

In a common existing method used in laptop computers, the temperature of the processor is measured and fed back to control (throttle) the operating frequency. This only adjusts for one parameter and usually the frequency is not increased above the nominal operating frequency. In AMD's PowerNow method [1], processing power can be supplied on demand, but overall performance is not improved. Intel also has power-saving technologies such as Centrino [8] and SpeedStep [9]; likewise, overall processing performance is not improved. In [14], a control technique is given that does allow the frequency to improve. However, it is an open-loop system, errors in any form are not explicitly detected, and the temperature and voltage changes are only estimated. No prototype was built. Our approach subsumes many of the benefits of such systems and can take advantage of more of the typically valued parameters in a system.

In [19], a hybrid synchronous/asynchronous system is proposed having an on-chip clock generator whose frequency tracks changes in operating temperature and voltage. Therefore, the system is able to partially take advantage of typical operating and manufacturing conditions. However, it is an open-loop system: Errors are not detected or modeled; this limits its effectiveness. Its approach to the possibility of metastability is to stop its clock for an indefinite period; this is not desirable, especially in real-time systems. It is also an expensive system, requiring specialized clock buffering. No prototype was built.

In prior work, we devised a system called TIMERRTOL [23] in which timing errors were actually detected and tolerated by using two specially wired copies of the pipeline. However, while adapting to existing conditions, it was quite expensive, required substantial redesign of the target system, and required high fan-in comparators, potentially impacting the nominal cycle time. TEAtime greatly improves on TIMERRTOL.

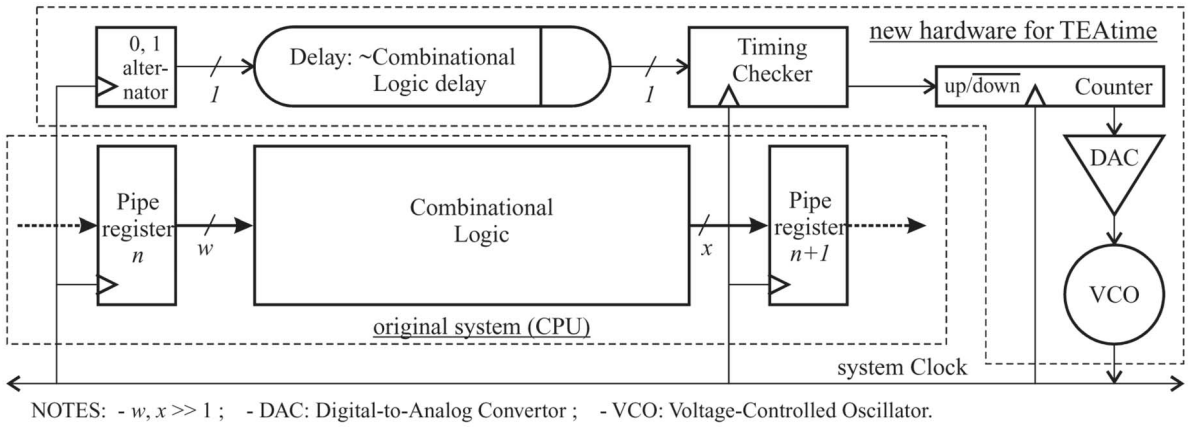

Fig. 1. Timing Error Avoidance (TEAtime) block diagram.

3 Timing Error Avoidance

3.1 Crux of the Idea

The basic idea of Timing Error Avoidance is to use extra logic with the delay of the longest path between pipeline registers, or the equivalent, to test on a cycle-by-cycle basis whether or not the system Clock is too fast or too slow. That is, if a signal applied to the input of the *delay test logic* (similar to a replicate) appears at the output of the test logic within the time of the machine's slowest path, speed up the Clock or, if it is greater than that of the critical path delay, minus a safety margin, slow down the Clock. Thus, since the delay test logic's characteristics (delay, etc.) mirror those of the main logic (they are realized close together on the same chip), the system Clock adapts both to dynamic environmental conditions, including temperature and operating voltage, as well as to statically varying manufacturing conditions.

In greater detail, the delay test logic is composed of the following design/construction and operation elements; see Fig. 1:

Determine the critical path between register elements within a digital machine. In the case of a pipelined CPU, this means to determine the slowest (clock-period determining) stage and the critical (longest, timewise) path through that logic.

Construct a one-bit wide version of that logic in which a change at the one-bit version's input from a logic 0-to-1 or a 1-to-0 propagates all the way through to the end of the logic. This *delay test logic* is not connected to any of the regular logic of the machine. However, the delay test logic nominally has the same delay as the worst-case path through the machine.

Drive the delay test logic with alternating 1s and 0s, the latter synchronized with the system Clock. The location of this test input corresponds to the output of the beginning pipeline register of the slowest pipeline stage in a CPU.

At the end of every cycle, if the test data has not reached the output register of the pipeline stage before the system Clock edge, then the system is operating slower than it might and the system Clock frequency is increased. If, however, the test data has reached the output register, then the system Clock frequency is getting close to the system's limit and, thus, the system Clock frequency is reduced.

The basic components of the variable system Clock are also shown in Fig. 1. They are standard logic and analog elements: an up/down counter to drive the DAC, which in turn generates an analog voltage to drive the VCO; hence, the counter sets the frequency of the system Clock. (In the example system, the counter is always changing and, by at most 1, up or down.) With advances in VLSI technology, all of these elements should be realizable on the same chip as the system. Note that, since there is an explicit feedback loop from the system Clock to the counter's setting, the absolute value of the counter is not important, only that it be able to go up and down with the timing checking circuitry's commands.

3.2 Other TEAtime Details

In order to show the simplicity of the main TEAtime circuitry, we provide low-level details of its realization in Fig. 2. The alternating 1s and 0s are created by a Flip-Flop

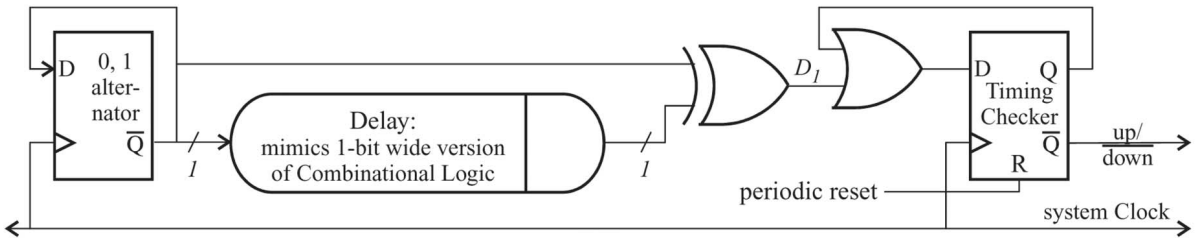

Fig. 2. Details of the central TEAtime circuitry. Other than the delay test logic, only two gates and two flip-flops are needed. See Fig. 3 for the corresponding timing diagram.

wired for toggle operation. The delay test logic (not shown) as used in the example system herein consists of a one-bit slice through an address multiplexer, the CPU's register file, the bypass multiplexer used for operand forwarding in the CPU (to reduce data dependencies), and a zero-detecting comparator (across the whole data path width).

The exclusive-OR gate normalizes the delayed signal so as to present a signal to the timing checker with the same polarity, regardless of the output of the toggle flip-flop. The following OR gate is used to latch a "decrease Clock frequency" signal if it occurs at any check time over the course of hundreds or thousands of cycles. The Clock frequency is then changed appropriately and a "periodic reset" signal is applied to the TEAtime circuitry.

The delay of the delay test logic is adjusted at system design time to be slightly greater than that of the aforementioned critical path to give a suitable safety margin. This is a relatively simple procedure when a high-quality logic simulator is used in the design process. In the case of our example CPU system, a structural simulation was performed on the CPU running the test program. This produced detailed timing information for every path in the CPU. From this information, we obtained both the worst-case operating frequency for a non-TEAtime (baseline) CPU and checked the performance of the TEAtime logic to ensure that the system Clock frequency was reduced before the timing constraints of the regular CPU logic were violated; hence, Timing Error Avoidance was guaranteed.

3.3 Dealing with the Possible Metastability in the Timing Checker

So far so good, everything seems pretty simple and straightforward. However, there is one place in the TEAtime system as so far described where system failure can occur; this is at the input of the Timing Checker, where the delayed signal is latched into a flip-flop. Since the delayed signal can be positioned anywhere in time and is not synchronized with the system Clock, there is the possibility that the delayed test signal could change value at the same time as the signal is being latched in the Timing Checker. This can result in metastability at the output of the timing checker in which the physical value of the logic output signal of the Checker's flip-flop is neither 0 or 1. It is well-known that metastable signals can stay in this state indefinitely, leading to misinterpretation of the signal's value by the rest of the system's logic; hence, system malfunction would ensue.

Our approach to this problem is to reduce the likelihood of metastability to a negligible level. (This is a common approach since there is no known way to eliminate the possibility of metastability in a synchronous system with one or more asynchronous inputs. In practice, every synchronous system falls into this category.)

In TEAtime, the frequency is only changed infrequently, roughly once for hundreds or thousands of cycles. Therefore, since D_1 in Fig. 2 is sampled hundreds if not thousands of times in between system clock frequency changes and it only has to indicate an error once during that time to force a reduction in the frequency, the likelihood of metastability occurring can be reduced to an arbitrarily low level, that is, a negligible level.

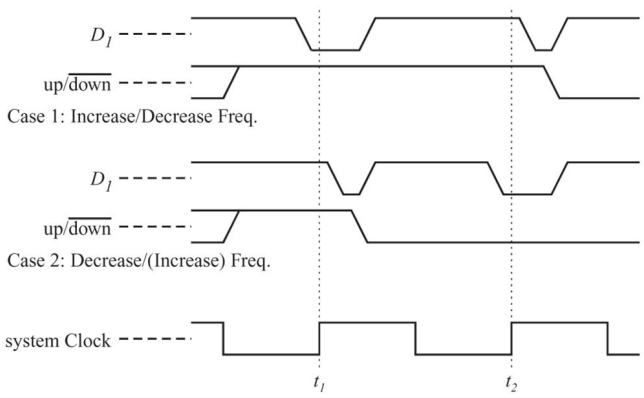

Fig. 3. Timing Checker/TEAtime detailed timing. (Refer to Fig. 2 for the generating logic.) Briefly, the system Clock frequency is decreased if, for any cycle between Timing Checker resets, the system is going too fast, as indicated by the delay test logic. See the text for the detailed explanation of the timing.

3.4 TEAtime Timing

The detailed timing of the TEAtime logic of Fig. 2 is shown in Fig. 3. The timing of the D_1 1-to-0 transition with respect to the active clock edge determines whether the system is going too fast or too slow. The detailed operation is as follows: At some time prior to t_1, the Timing Checker was reset, so the system can now potentially increase the system Clock frequency. In Case 1, at t_1, the delay test logic's timing has not been violated, that is, the 1-to-0 D_1 transition has come before the active clock edge, so the Timing Checker still wants to increase the system Clock frequency. However, at t_2, the delay test logic's delay is now greater than the system Clock period, that is, the 1-to-0 D_1 transition comes after the active clock edge and, thus, the timing has been violated, so the Timing Checker switches to cause an eventual decrease in the system Clock frequency.

In Case 2, the (almost) reverse happens. At t_1, the system is running too fast, so the Timing Checker switches to cause a decrease in system Clock frequency. Now, at time t_2, the delay test logic's delay is less than the Clock period, so, theoretically, the system Clock frequency could be increased. However, the Timing Checker has already been latched into the "decrease Clock frequency" state and, thus, stays in that state until the Checker is again reset after the system Clock frequency is decreased, possibly hundreds or thousands of cycles later. The system Clock frequency is only increased if the delay test logic's delay is less than the system Clock period, that is, is safe, for every cycle between Timing Checker resets.

3.5 TEAtime Summary and Analysis

The TEAtime logic is very cheap and conceptually straightforward. For a given CPU, say 32 bits, the hardware cost of the delay test logic is less than 1/32 of the cost of the slowest pipeline stage. The variable frequency oscillator adds a lttle more cost, but this is quite small.

Should a CPU or other digital system have two or more pipeline stages of similar delay, they are all treated as described herein for the single stage case, with a "decrease Clock frequency" signal from any of them having priority

TABLE 1
UUT (Unit-Under-Test) CPU Characteristics

CPU Architectural Features	Size or Description
General CPU type	RISC (DLX-style)
Data word width	32 reduced to 8 bits
Instruction word width	32 bits
Memory bus width	32 bits
Number of instructions in the architecture	25
Number of General Purpose Registers (GPR)	16
CPU Microarchitectural Features	**Size or Description**
Data path type	Pipelined
Number of stages	5
Stage types	IF/ID-OF/EX/MEM/WB[1]
GPR accessing	Write then Read in same cycle (DLX-style)
Stage interconnections	Full data-forwarding
Branch prediction	Static, always 'not taken'

1. Instruction Fetch; Instruction Decode & register Operand Fetch; EXecute operation; load or store MEMory operation; register Write Back.

for the setting of the Clock frequency. Put another way, multiple TEAtime circuits are used to handle a complex microprocessor or system, such as the Intel Pentium 4, that might have hundreds or thousands of critical or near-critical paths. In such a case, a TEAtime circuit, Fig. 2, is made for each of these paths and is placed with its associated path as usual. If any one of these TEAtime circuits indicates a "decrease Clock frequency" signal, the system clock frequency is decreased. Thus, there is a chip-wide OR-circuit to generate the overall "decrease" signal. Since there are hundreds or thousands of system clock cycles per system clock frequency change, the OR circuit construction is not critical; there is plenty of time for such a large fan-in circuit to produce its output. A wired-OR implementation is likely.

4 Experimental Methodology

Our experimental goals were to realize TEAtime on a reasonably complex real digital system (a CPU) to demonstrate TEAtime's functionality, performance, and adaptability to changing conditions.

The TEAtime ideas were tested and evaluated on a 32/8 bit pipelined CPU designed in the mold of that in [18]. Two sets of experiments were performed:

1. Basic functionality and stabilization testing with performance evaluation.
2. Adaptive abilities of TEAtime to temperature changes.

4.1 CPU Characteristics

The CPU has a 32-bit data path to memory and 32-bit wide instructions. It was originally designed in the DLX [18] RISC style for use in the URI undergraduate computer engineering curriculum. See the "ICED" website for detailed information on the source CPU [24]; a summary of the CPU's salient features is given in Table 1. An FPGA was used to realize the CPU, its memory, the system controller, and the TEAtime logic. Mentor Graphics and Xilinx CAD tools were used for the design process. The internal data paths of the CPU were reduced to 8-bits wide to keep the design small and allow for rapid design changes. (The instruction width stayed at 32 bits.) The CPU was pipelined and employed full register operand forwarding for minimal register data dependencies.

4.2 Test Program Characteristics

The test program for the CPU was written to be small (28 static instructions) so as to fit in the equivalent of a small 32 4-byte word cache memory on the FPGA (one cycle access time). Pseudocode of the Assembly program is shown in Table 2; a listing of the Assembly code itself may be found on the Web [25]. The function of the test program was to go through a small randomized set of positive and negative numbers and create two sums: one of the positive numbers and one of the negative numbers, with these results stored in the externally accessible cache. While the test program was quite small, all of the usual functions were performed, including forward and backward conditional branching, subroutine calling, and logic and arithmetic functions. The worst-case critical path through the CPU's logic was exercised.

4.3 The Experimental Equipment and Setup

The experiment setup is shown in Fig. 4. The major components include the 100,000 gate equivalent Xilinx FPGA mounted on the XESS Corporation evaluation board, the XESS board itself, additionally containing the Host PC interface and the reference oscillator (~25.0 MHz). The latter operated at a frequency well below the system Clock frequency and was used to drive the interface and the CPU controller so as to make sure they did not affect the results.

The reference oscillator was also used to measure the system Clock frequency. The reference clock cycles were counted during test program execution; since the test program ran in a constant 194 system Clock cycles, the system Clock frequency was readily computed from the reference clock cycle count and its frequency.

The breadboard was a home-brew affair, containing the 10-bit DAC and the 25-100 MHz VCO, as well as supporting circuitry. The DAC and the VCO were each one integrated

TABLE 2
C-Like Pseudocode of the Test Program

Label	Instruction	Comment
	negsum = 0;	//initialize running sum of negative numbers
	possum = 0;	//initialize running sum of positive numbers
	mask = 0x80;	//mask for MSB of a byte
	i = 0;	//initialize loop counter (will loop for 8 data)
	dataptr = &data;	//initialize pointer to data
	stackptr = &stack;	//initialize pointer to stack
loop:	datumptr = dataptr + i;	//create pointer to current data byte
	temp = *(datumptr +0);	//load first data byte to a temp variable (register)
	*(stackptr +0) = temp;	//put the temp on the stack
	call elmnt;	//jump to the subroutine (with branch-and-link instr.)
	i++;	//update the loop counter
	test = i – 7;	//see if done with looping
	if (test != 0) goto loop;	//backwards conditional branch forming loop
	*(dataptr + 8) = negsum;	//save the sum of the negative numbers in memory
	*(dataptr + 9) = possum;	//save the sum of the positive numbers in memory
	halt;	//end of program, pass control back to host
elmnt:	temp2 = *(stackptr +0);	//start of subroutine; get the datum off of the stack
	temp3 = temp2 – 0;	//dummy instruction
	temp3 = temp3 & mask;	//test the MSB (to see if it's a negative number)
	if (temp3 == 0) goto pos;	//branch if it's a positive number
	negsum = negsum + temp2;	//update the running sum of negative numbers
	if (0 == 0) goto ret;	//equivalent to unconditional branch to return statement
pos:	possum = possum + temp2;	//update the running sum of positive numbers
ret:	return;	//indirect branch to return from elemnt subroutine
data:	0xf00238f9	//pseudo-random data to be tested and added
	0x0ffde700	// 8 bytes, total; 4 negative numbers interspersed
		// with 4 positive numbers
results:	0x00ffeedd	//where the two sums go (2 LS bytes); init. to garbage
stack:	0xffffffff	//one-word (4 byte) stack

This program uses 32 4-byte words of storage, including the input data and the results.

circuit. (Normally, of course, these would be on the same chip as the digital system, but this is not essential.)

A thermocouple was attached to the center of the top of the FPGA with thermal grease and was used by the METEX meter to measure the case temperature of the digital system.

The XESS board and the breadboard were mounted in a cavity within the BMA environmental chamber. An insulated hole in the chamber side provided access to the contained circuitry for the PC and temperature meter. Only a fraction of the possible temperature range of the chamber was used. Since many of the non-FPGA components in the chamber were only specified for operation at ambient temperatures between 0 and 70 degrees Celsius, the chamber temperature was kept to within 5 and 65 degrees Celsius. The chamber used an electric coil for heat and carbon dioxide gas for cooling. Note that the FPGA is specified for a much wider temperature range: A junction, not ambient, temperature of 85 degrees Celsius maximum to 0 degrees Celsius minimum for recommended operation and 125 degrees Celsius absolute maximum with no absolute minimum, so our results could be further improved upon.

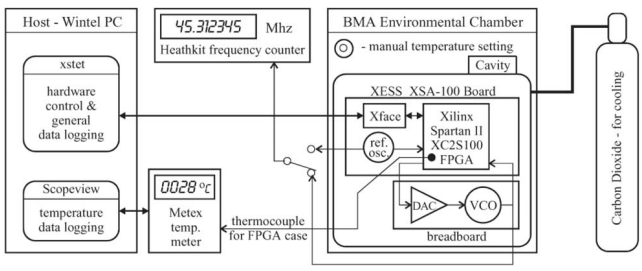

Fig. 4. TEAtime evaluation experimental setup.

The frequency counter was used to precisely measure the reference oscillator's frequency at different temperatures, to provide system Clock frequency calculation corrections. The counter was also used to validate the on-chip system Clock frequency measurements by measuring the system Clock frequency directly. (It had no electronic interface, so it could not be used for the actual high-speed data gathering.)

The Host PC ran the experiments with a home-brew control program: xstet. xstet was also used for all of the data logging, except for case temperature; the latter measurements were logged by a separate program provided with the METEX temperature meter.

4.4 Experimental Operation

A self-explanatory flowchart of the operation of the xstet program and hence the experiments is shown in Fig. 5. Each pass through the main loop took about 12 milliseconds and

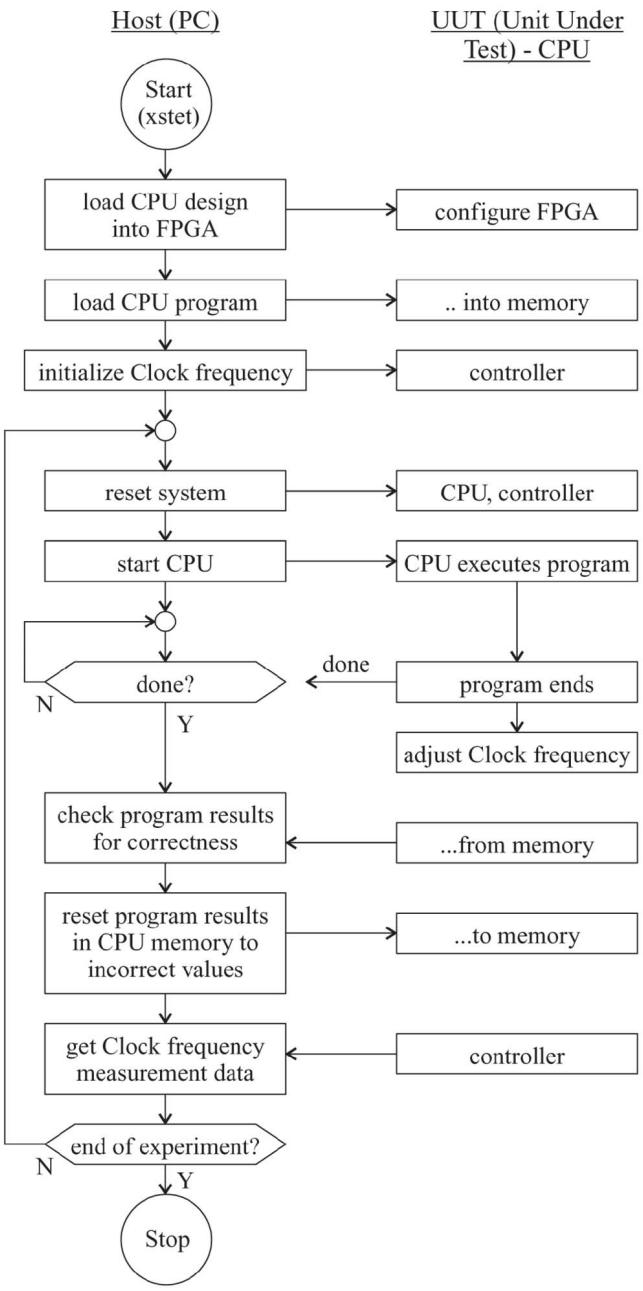

Fig. 5. Experiment operation.

was primarily due to Host PC/interface delays. Once the results from each run of the test program were read and checked, the results' memory locations were overwritten with junk to ensure that the following run produced correct results.

Note that the actual system Clock frequency changing, after initialization by the Host, was solely directed and caused by the CPU and TEAtime hardware. Also, although the system Clock frequency was only changed between test program runs, this would not be necessary for a production unit with a suitably designed system Clock having no glitches during frequency changes. (The latter may be the case with our oscillator, but, since it was not the point of the study, we did not investigate it.)

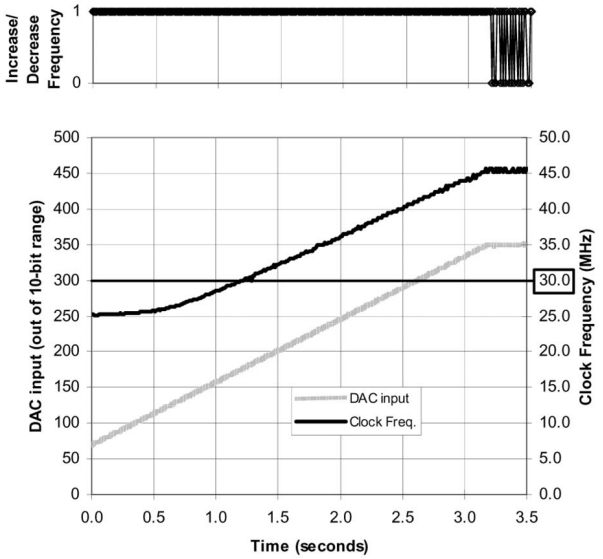

Fig. 6. Experiment Set 1, part A: TEAtime automatically rises to a high frequency and then stabilizes at that point. The horizontal line at 30 MHz indicates the approximate baseline worst-case operating frequency.

5 TEAtime Experiments

5.1 Experiment Set 1: Basic Operation and Stabilization

In these experiments the environmental chamber was not used. All of the data were taken at room temperature ($T_{ambient} = 22$ degrees C).

The worst-case operating frequency of the CPU, as determined through the structural simulations mentioned earlier, was 35.7 MHz (period of 28 ns.). With a design safety margin of 20 percent, this equates to what would be a non-TEAtime specified operating frequency of about 29.8 MHz (period of 33.6 ns.). The latter are our baseline conditions.

The results of part A of the first set of experiments are shown in Fig. 6. The top subplot shows the value of the up/down control line over time; this subplot is horizontally aligned with the main subplot below it. It is seen that, shortly after the starting frequency of about 25 MHz, the system Clock rises steadily to its final stabilization frequency of about 45.1 MHz. From a time (period) perspective, this is a performance increase of 34.0 percent over the baseline worst-case system Clock frequency. The frequency also stabilizes rapidly, at a rate of about 8 MHz/sec. Note that, in both parts of this experiment, all of the data is shown, so adjacent data in the plots were adjacent during the experiments.

The main conclusions from this part are that TEAtime works and provides a substantial performance gain under typical operating conditions.

While part A represents a normal operating condition, we were also curious as to both how high a frequency TEAtime could go as well as how it would operate starting from an elevated frequency. The results of this part, part B, are shown in Fig. 7. As is seen from the plot, the system Clock frequency can go as high as about 57.3 MHz and TEAtime will still continue to function and produce correct results. The frequency drops at the same rate as it rose in part A and stabilizes at the same frequency as in part A.

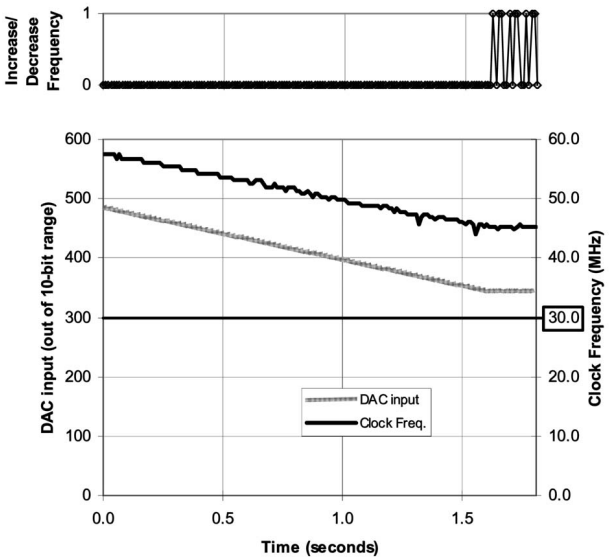

Fig. 7. Experiment Set 1, part B: Decreasing-frequency stabilization. The system stabilizes to the same frequency as in part A, this time with a decreasing frequency. The horizontal line at 30 MHz indicates the approximate baseline worst-case operating frequency.

The high possible starting frequency in part B indicates both that this realization of TEAtime has a big safety margin built into its design and that the delay test logic could possibly be tuned for less delay to get more performance and still maintain a good safety margin. Considering the performance aspect, if operated at the 57.3 MHz frequency, the TEAtime performance improvement would be about 48.1 percent. As implied earlier, a safety margin is still needed so that that specific performance would not actually be realized under these operating conditions.

5.2 Experiment Set 2: Temperature Change Response

In this experiment (Part A), the chamber temperature setting was changed twice: at time 0 from 25 to 65 degrees C and at about time 1250 seconds from 65 to 5 degrees C. The results are shown in Fig. 8. Both the system Clock frequency and the CPU case temperature were measured as the temperature was varied.

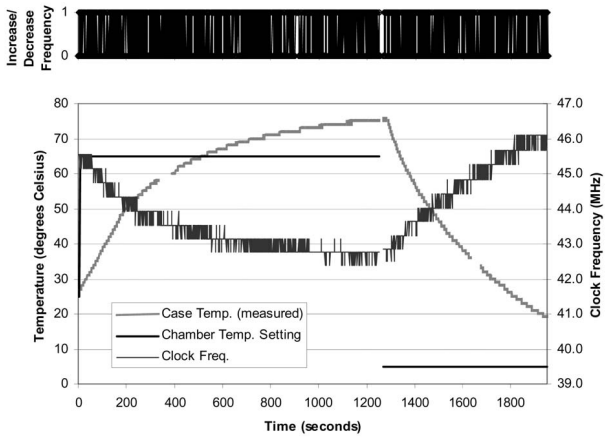

Fig. 8. Experiment Set 2, Part A: TEAtime temperature change response. Each datum is a run sample taken every second; the CPU runs continuously, however.

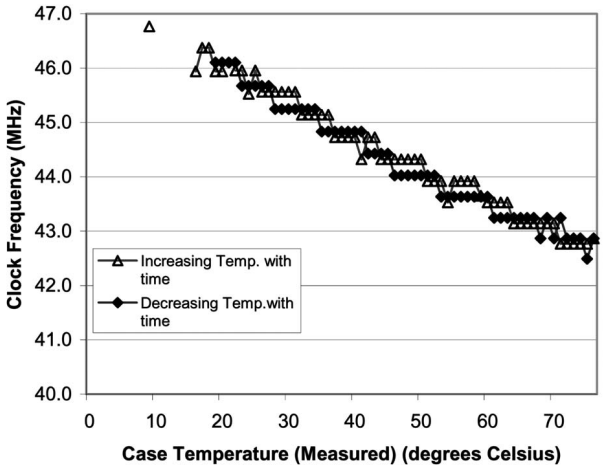

Fig. 9. Experiment Set 2, Part B: System Clock Frequency versus Case Temperature. Only one datum per degree C is plotted.

As the results in Fig. 8 show, the system Clock frequency tracks the changes in operating temperature. Note the change in frequency scale from the prior plots; also note the large granularity of system Clock frequency change. The latter is an artifact of the coarse frequency measuring system employed on the chip; the actual frequency steps are finer, as can be seen in the prior plots.

We also note that, as expected, performance improves with lower temperatures. These results also show that TEAtime readily adapts the system Clock frequency to existing environmental conditions (at least for temperature changes). We also see that TEAtime can adapt to quickly changing temperatures, as well, at least as fast as 0.1 degree C/second.

(The two small gaps in the case temperature data were due to temporary case temperature-data logging omissions.)

Last, for Part B, data from experiment Part A was combined with other similarly obtained temperature data to produce the system Clock frequency versus case temperature plot of Fig. 9. In this plot, only one data point (a typical value) for each unit temperature value was plotted. As the plot shows, there is little or no hysteresis in the frequency changes and the relationship between the two variables is approximately linear.

6 Conclusions

In this paper, we have presented a simple and cheap technique for improving performance of many, if not all, synchronous digital systems, both simple and complex. This technique, TEAtime, dynamically changes the system's frequency to enhance performance and adapt to the system's operating conditions. If tuned carefully, TEAtime can come close to maximizing a system's performance.

TEAtime was realized in physical hardware and its functionality, performance gains, and adaptability were verified.

We conclude that this is a viable and useful method to be used in many if not most digital systems, especially

embedded systems requiring high performance under changing and possibly extreme environmental conditions. Further, TEAtime may possibly help to increase manufacturing yields since the system would also be able to adapt to static variations in its own construction.

ACKNOWLEDGMENTS

The author is indebted to Professors Godi Fischer and James Daly of the Department of Electrical and Computer Engineering, University of Rhode Island, for the loan of their environmental chamber and for guidance in its use. Mr. Michael Platek, also of this department, also helped greatly with the chamber's setup. This work was supported in part by the University of Rhode Island Office of the Provost, Xilinx Corporation, and Mentor Graphics Corporation. Patent applied for. A previous version of this paper appeared in [26]. Follow-on work appeared in [27].

REFERENCES

[1] AMD, "AMD PowerNow!™ Technology Brief,"Advanced Micro Devices, Inc., 2003, http://www.amd.com/us-en/assets/content_type/DownloadableAssets/Power_Now2.pdf.
[2] F. Boyer, M. Aboulhamid, Y. Savaria, and I. Bennour, "Optimal Design of Synchronous Circuits Using Software Pipelining Techniques," *Proc. 1998 Int'l Conf. Computer Design,* 1998.
[3] T.D. Burd, T.A. Pering, A.J. Stratakos, and R.W. Brodersen, "A Dynamic Voltage Scaled Microprocessor System," *IEEE J. Solid-State Circuits,* vol. 35, no. 11, pp. 1571-1580, Nov. 2000.
[4] D.M. Chapiro, "Globally-Asynchronous Locally-Synchronous Systems," PhD thesis, Computer Science Dept., Stanford Univ., Oct. 1984.
[5] M.J. Flynn, P. Hung, and K. Rudd, "Deep-Submicron Microprocessor Design Issues," *IEEE Micro,* July-Aug. 1999.
[6] S. Furber, "Asynchronous Logic," *IberChip,* Feb. 1996.
[7] R. Ginosar and R. Kol, "Adaptive Synchronization," *Proc. 1998 Int'l Conf. Computer Design,* 1998.
[8] Intel Staff, "Get a Notebook that Enables Extended Battery Life,"Intel Corp., 2003, http://www.intel.com/products/mobile technology/battery.htm.
[9] Intel Staff, "Intel® Low-Power Technologies,"Intel Corp., 2003, http://www.intel.com/ebusiness/products/related_mobile/wp021601_sum.htm?iid=ipp_battery+press_lowpow.
[10] A. Kondratyev and K. Lwin, "Design of Asynchronous Circuits Using Synchronous CAD Tools," *IEEE Design and Test of Computers,* vol. 19, no. 4, pp. 107-117, July/Aug. 2002.
[11] T. Kuroda, K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, T. Sakurai, and T. Furuyama, "Variable Supply-Voltage Scheme for Low-Power High-Speed CMOS Digital Design," *IEEE J. Solid-State Circuits,* vol. 33, no. 3, pp. 454-462, Mar. 1998.
[12] C. Leiserson and J. Saxe, "Retiming Synchronous Circuitry," *Algorithmica,* vol. 1, pp. 3-35, 1991.
[13] A.J. Martin, "Design of an Asynchronous Microprocessor," Technical Report CS-TR-89-02, Computer Science Dept., California Inst. Technology, Pasadena, 1989.
[14] A. Merchant, B. Melamed, E. Schenfeld, and B. Sengupta, "Analysis of a Control Mechanism for a Variable Speed Processor," *IEEE Trans. Computers,* vol. 45, no. 7, pp. 968-976, July 1996.
[15] M. Miyazaki, J. Kao, and A. Chandrakasan, "A 175mV Multiply-Accumulate Unit Using an Adaptive Supply Voltage and Body Bias (ASB) Architecture," *Proc. Int'l Solid-State Circuits Conf. (ISSCC),* Feb. 2002.
[16] S.F. Oberman, H. Al-Twaijry, and M.J. Flynn, "The SNAP Project: Design of Floating Point Arithmetic Units," *Proc. 13th IEEE Symp. Computer Arithmetic,* July 1997.
[17] M. Olivieri, A. Trifiletti, and A. De Gloria, "A Low-Power Microcontroller with On-Chip Self-Tuning Digital Clock Generator for Variable-Load Applications," *Proc. 1999 Int'l Conf. Computer Design,* 1999.
[18] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach,* first ed. San Mateo, Calif.: Morgan Kaufman Publishers, 1990.
[19] A.E. Sjogren and C.J. Myers, "Interfacing Synchronous and Asynchronous Modules within a High-Speed Pipeline," *Proc. 17th Conf. Advanced Research in VLSI (ARVLSI '97),* pp. 47-61, 1997.
[20] I.E. Sutherland, "Micropipelines," *Comm. ACM,* vol. 32, no. 6, pp. 720-738, June 1989.
[21] H. Suzuki, W. Jeong, and K. Roy, "Low Power Adder with Adaptive Supply Voltage," *Proc. Int'l Conf. Computer Design (ICCD 2003),* Oct. 2003.
[22] N. Tzartzanis, W. Walker, H. Nguyen, and A. Inoue, "A 34-Word 64b 10R/6W Write-Through Self-Timed Dual-Supply-Voltage Register File," *Proc. Int'l Solid-State Circuits Conf. (ISSCC),* Feb. 2002.
[23] A.K. Uht, "Achieving Typical Delays in Synchronous Systems via Timing Error Toleration," Technical Report 032000-0100, Dept. of Electrical and Computer Eng., Univ. of Rhode Island, Kingston, Mar. 2000.
[24] A.K. Uht, "ICED Canned CPU—IcpuED," http://www.ele.uri.edu/iced/protosys/canned/cpu/iced-canned-cpu.html, June 2004.
[25] A.K. Uht, "TEAtime Prototype Test Program Listing," http://www.ele.uri.edu/~uht/research/TEAtime/BMK2.LST, June 2004.
[26] A.K. Uht, "Uniprocessor Performance Enhancement through Adaptive Clock Frequency Control," *Proc. SSGRR-2003w Int'l Conf. Advances in Infrastructure for e-Business, e-Education, e-Science, e-Medicine, and Mobile Technologies on the Internet,* Jan. 2003.
[27] A.K. Uht, "Going Beyond Worst-Case Specs with TEAtime," *Computer,* vol. 37, no. 3, pp. 51-56, Mar. 2004.

Augustus K. Uht received the PhD degree in electrical (computer) engineering from Carnegie-Mellon University. He is a research professor in the Department of Electrical and Computer Engineering, University of Rhode Island. His research interests include computer architecture, especially the fields of instruction-level parallelism, multipath execution, adaptive computing, and underclocking. He is a member of the IEEE and the IEEE Computer Society.