Uniprocessor Performance Enhancement Through Adaptive Clock Frequency Control

Augustus K. Uht

***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 CPU's 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*— Adaptive clock design, computer design, microarchitecture, high-performance computing.**

I. 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. Lastly, 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[3]. 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 extra cost is very small, while the performance gains are substantial. The technique is applicable to any synchronous digital system. Correct results are ensured if the design guidelines are followed. Neither the base cycle time or 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% 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 II. In Section III the basic ideas of timing error avoidance are presented, using our test CPU as a case study. Our experimental methodology is described in Section IV, with the experimental results presented in Section V. We conclude in Section VI.

II. Related Work

There has been prior work somewhat similar to ours, but nothing that encompasses all of the attributes of our technique, not to mention actually demonstrating its functioning and characteristics with a real prototype. The closest work we are aware of is [10]. 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 1's) 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

This work was supported in part by the U.S. National Science Foundation under Grants Nos. MIP-9708183, DUE-9751215 and EIA-9729839, the University of Rhode Island Office of the Provost, the Xilinx Corp., and the Mentor Graphics Corp. Patent applied for.

Augustus K. Uht is with the University of Rhode Island, Department of Electrical and Computer Engineering, 4 East Alumni Ave., Kingston, RI 02881 USA (telephone: +1-401-874-5431, e-mail: mailto:uht@ele.uri.edu).

scheme takes advantage of 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?

There has been a large amount of work on asynchronous systems. See, for example, [7] for a description of the first asynchronous microprocessor and [3] 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*) [13]. 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[5].

There have been many methods created to improve the performance of synchronous circuits. The main approach is to *retime* [6] 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[1]. However, worst case delays between storage elements must still be maintained.

Multisynchronous systems[4] 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.

Wave pipelined arithmetic units have been proposed, but have implementation difficulties [2, 9], 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 one existing method used in some 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 [8] 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 [12] 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[14] 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. This paper greatly improves on TIMERRTOL.

III. Timing Error Avoidance

A. 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 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:

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, time-wise) path through that logic.
2. 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.

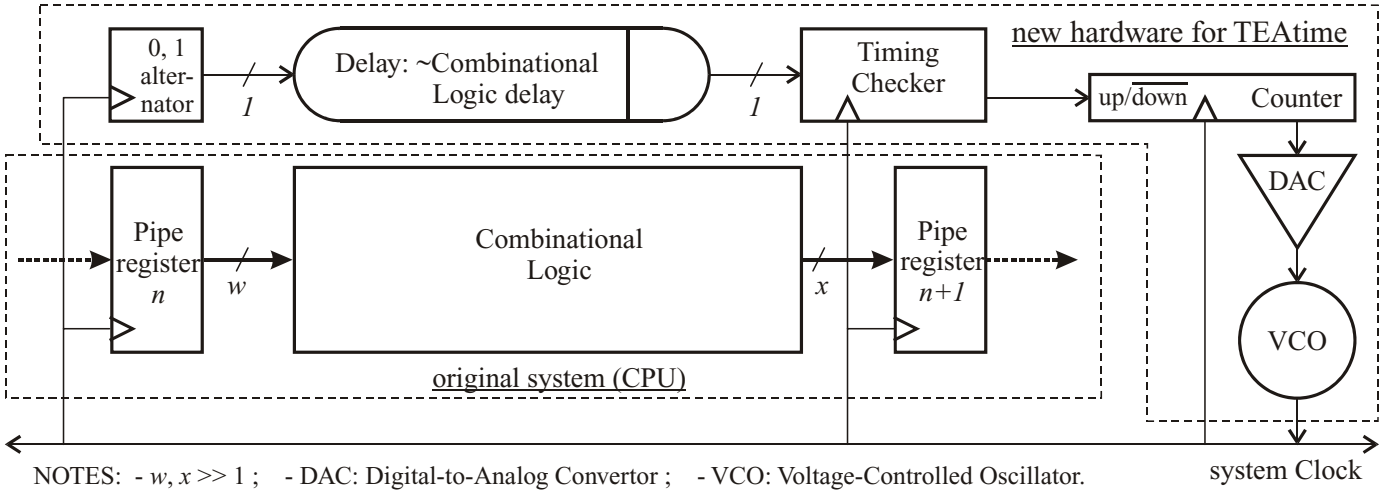

Fig. 1. Timing Error Avoidance (TEAtime) block diagram

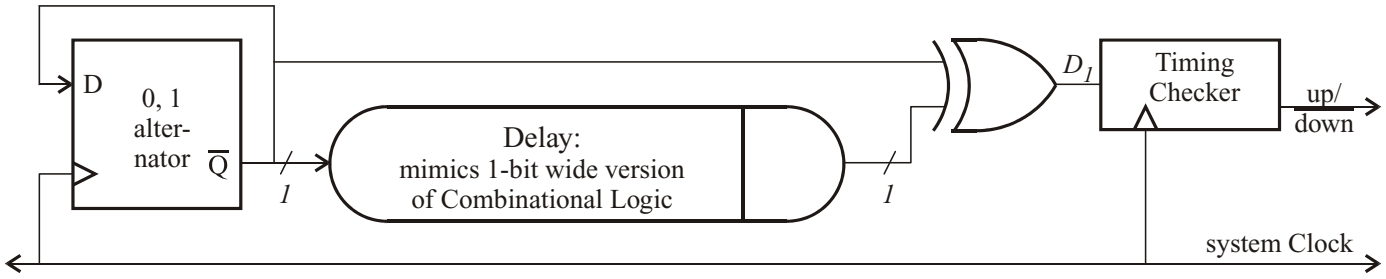

Fig. 2. Details of the main components of TEAtime circuitry.

.

3. Drive the delay test logic with alternating 1's and 0's, 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.
4. 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.

B. 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 1's and 0's are created by a Flip-Flop 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 multiplexor, the CPU's register file, the bypass multiplexor 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 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. From this simulation, 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.

C. 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 start 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 solution to this problem is shown in the detailed Timing Checker design of Fig. 3, with the corresponding timing shown in Fig. 4. The basic idea is to sample the delay test signal, D1, at two different times. Then, for a single cycle, only one of flip-flops Q1 or Q2 can possibly be in a metastable state; they cannot both be metastable in the same cycle, since D1 only changes value at most once in a cycle. The output of the logic looking at Q1 and Q2 to determine up or down Clock frequency changing is only sampled long after a metastable condition can begin, as long as the frequency change increment is kept suitably small. The construction of this logic ensures that no metastable condition can propagate past the sample point. See the timing diagram, Case 3, for an example of the handling of a metastable condition. (Cases 1 and 2 show more typical frequency increasing and decreasing, respectively.)

We must also point out that the actual connection of the clock circuitry is slightly different than shown in the prior figures: the VCO actually generates the "Earlier Clock", while the actual system Clock is an output of the clock delay chain of the timing checker.

D. 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 little more cost, but this is quite small.

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

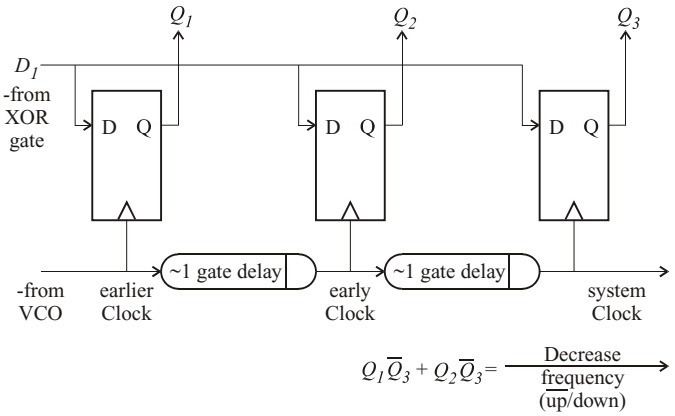

Fig. 3. Timing Checker detailed design.

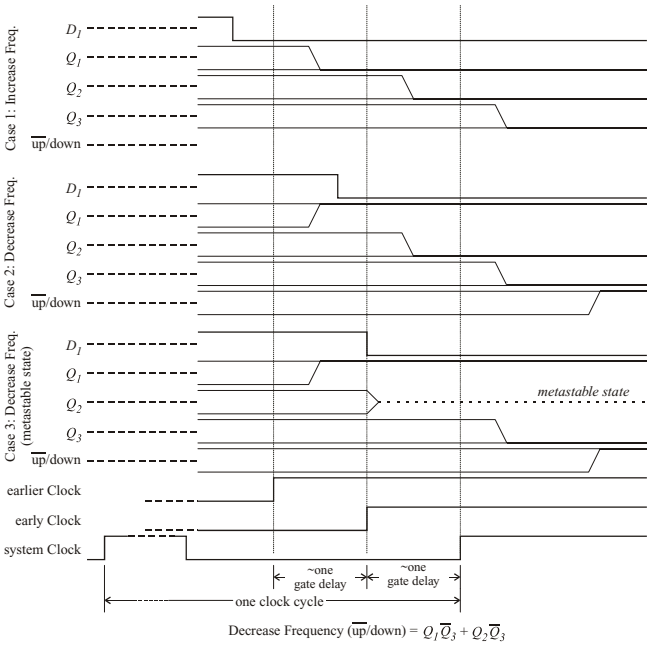

Fig. 4. Timing Checker / TEAtime detailed timing.

IV. 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 [11]. Three experiments were performed: basic functionality and stabilization testing with performance evaluation, and two experiments investigating the adaptive abilities of TEAtime to temperature changes.

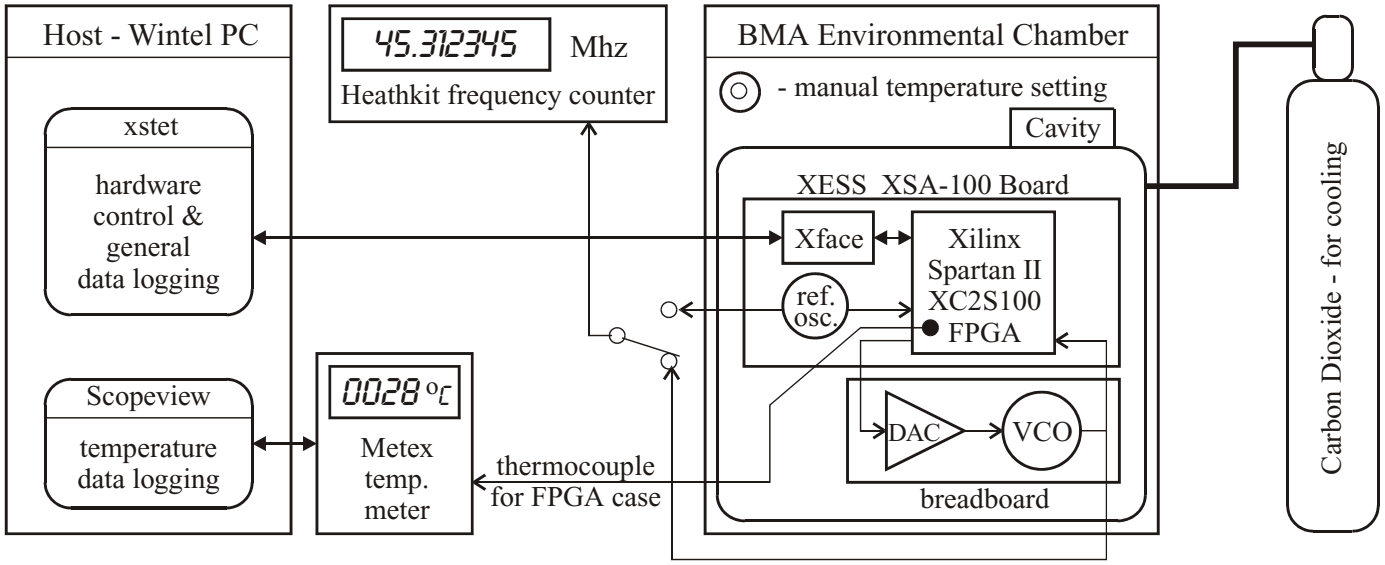

Fig. 5. TEAtime evaluation experimental setup.

A. CPU Aspects

The CPU has a 32 bit data path to memory and 32-bit wide instructions. (It was originally designed in the RISC style for use in the URI computer engineering curriculum.) 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 employed full register operand forwarding for minimal data dependencies

The test program for the CPU was written to be small (about 30 static instructions) so as to fit in the equivalent of a small 32 4-byte word cache memory on the FPGA (1 cycle access time). 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 forwards and backwards conditional branching, subroutine calling, and logic and arithmetic functions.

B. The Experimental Equipment and Setup.

The experiment setup is shown in Fig. 5. 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 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 absolute range: a junction, not ambient, temperature of 85 degrees maximum with no minimum, so our results could be further improved upon.

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 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.

C. Experimental Operation

A self-explanatory flowchart of the operation of the xstet program and hence the experiments is shown in Fig. 6. Each pass through the main loop took about 12 milliseconds, and was primarily due to Host PC/interface delays. Once the results from each run of the test program were read and checked, the memory results 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.)

V. TEATIME EXPERIMENTS

A. Experiment 1: Basic Operation and Stabilization

In this experiment the environmental chamber was not used. All of the runs were 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%, 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 experiment are shown in Fig. 7. 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% 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 to, as well as how it would operate starting from an elevated frequency. The results of this part, part B, are shown in Fig. 8. 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 it rose in part A, and stabilizes at the same frequency as in part A.

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%. As implied earlier, a safety margin is still needed, so that that specific performance would not actually be realized under these operating conditions.

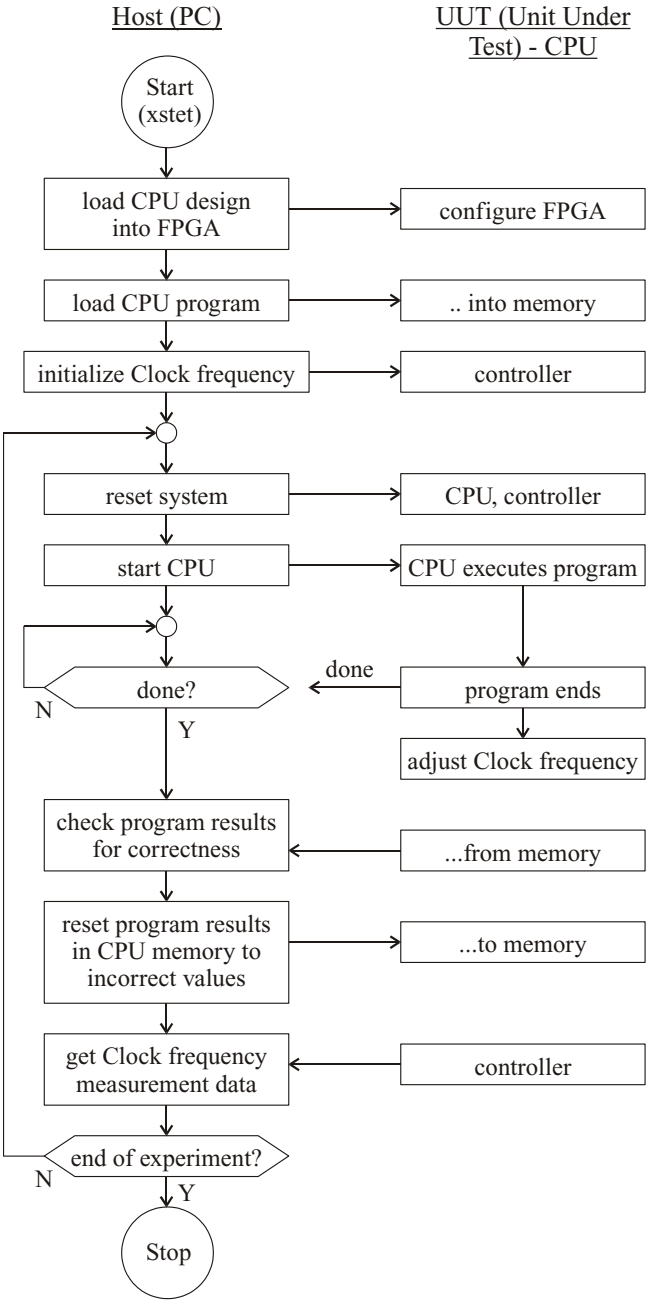

Fig. 6. Experiment Operation.

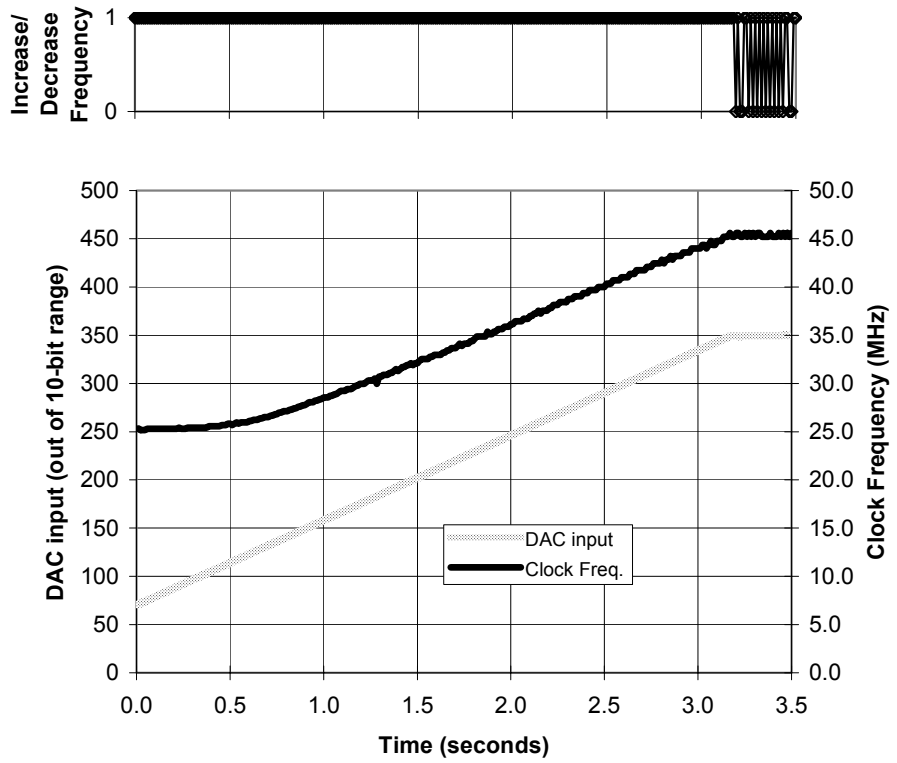

Fig. 7. Experiment 1, part A: TEAtime automatically rises to a high frequency and then stabilizes at that point.

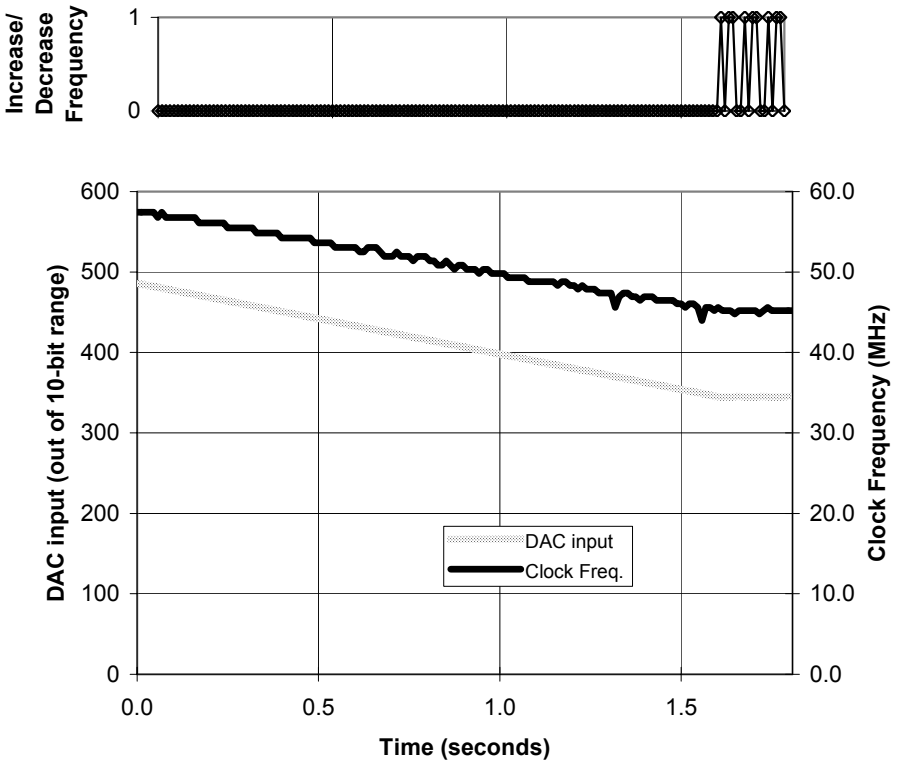

Fig. 8. Experiment 1, part B: Decreasing-frequency stabilization.

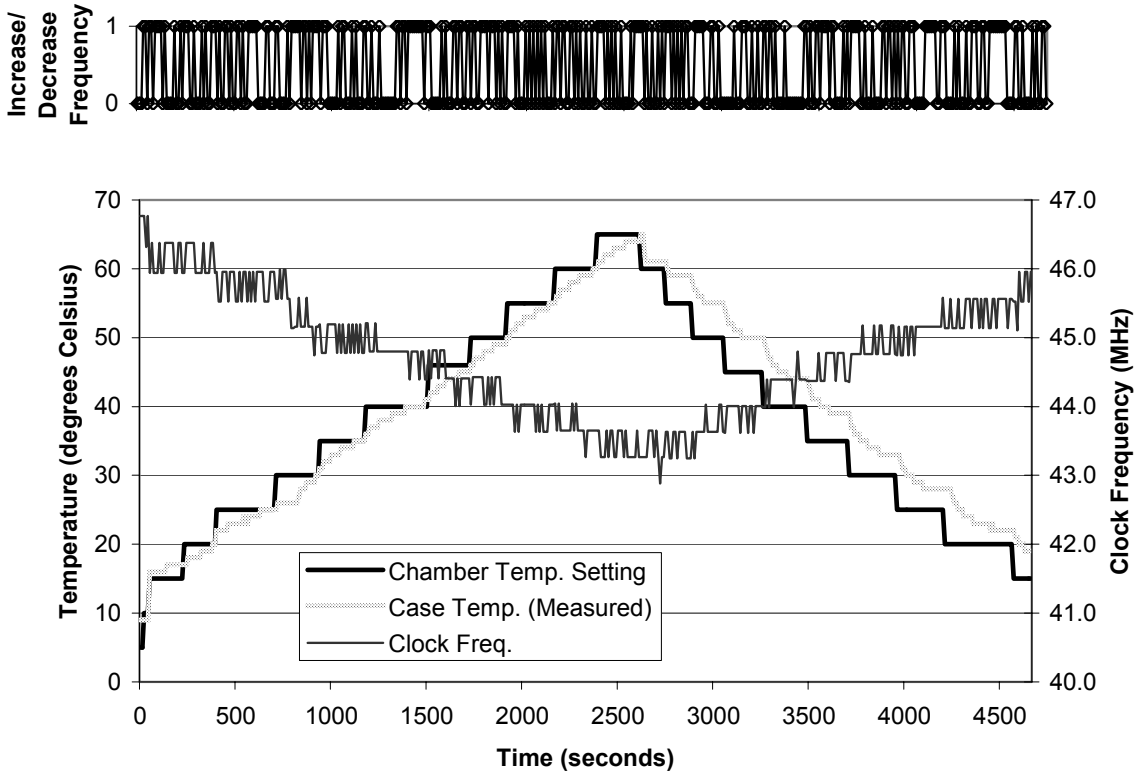

Fig. 9. Experiment 2: TEAtime gradual temperature change response. Each datum is a run sample taken every 10 seconds; the CPU always runs, however.

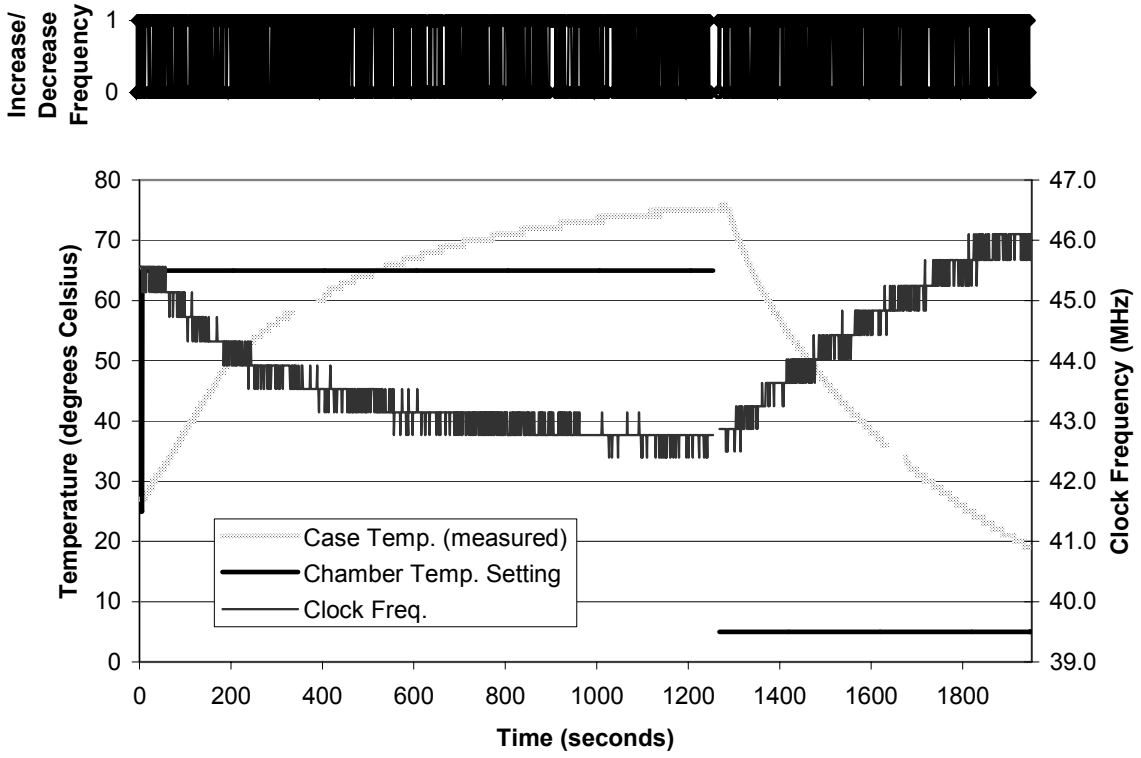

Fig. 10. Experiment 3: TEAtime rapid temperature change response. Each datum is a run sample taken every second; the CPU always runs, however.

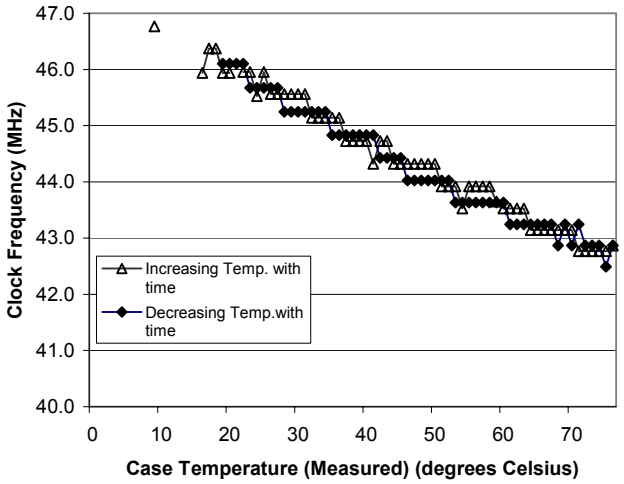

Fig. 11. System Clock Frequency vs. Case Temperature. Only one datum per degree C. is plotted.

B. Experiment 2: Temperature Response-Gradual Change

In this experiment the ambient temperature was varied from 5 degrees C. up to 65 degrees C. and then back down to 15 degrees C., in 5 degree steps. Both the system Clock frequency and the CPU case temperature were measured as the temperature was varied.

As the results in Fig. 9 show, the system Clock frequency tracks the changes in operating temperature almost immediately and with no difficulty. 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).

(As will be seen from the next plot, the operator's judgment of when the chamber temperature had stabilized was not as good as would be desired. However, this does not change the conclusions, especially in light of the results of Experiment 3.)

C. Experiment 3: Rapid Temperature Change Response

In this experiment the chamber temperature setting was changed only 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. 10.

In general, the results are about the same as in Experiment 2, except that here 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-logging malfunctions.)

D. Frequency versus Temperature

Lastly, some data from experiments 2 and 3 were combined to produce the system Clock frequency vs. case temperature plot of Fig. 11. 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, as one would expect.

VI. Conclusions

In this paper we have presented a simple and cheap technique for improving performance of many, if not all, synchronous digital systems. 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.

Acknowledgement

We are indebted to Professors Godi Fischer and James Daly for the loan of their environmental chamber, and for guidance in its use.

References

[1] F. Boyer, M. Aboulhamid, Y. Savaria, and I. Bennour, "Optimal Design of Synchronous Circuits Using Software Pipelining Techniques," in *Proceedings of the 1998 International Conference on Computer Design*, 1998.

[2] M. J. Flynn, P. Hung, and K. Rudd, "Deep-Submicron Microprocessor Design Issues," *IEEE MICRO*, July-August 1999.

[3] S. Furber, "Asynchronous Logic," in *IberChip*. Sao Paulo, Brazil, February 1996.

[4] R. Ginosar and R. Kol, "Adaptive Synchronization," in *Proceedings of the 1998 International Conference on Computer Design*, 1998.

[5] 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/August 2002.

[6] C. Leiserson and J. Saxe, "Retiming Synchronous Circuitry," in *Algorithmica*, vol. 1, 1991, pp. 3-35.

[7] A. J. Martin, "Design of an Asynchronous Microprocessor," Computer Science Department, California Institute of Technology, Pasadena, California, Technical Report CS-TR-89-02, 1989.

[8] A. Merchant, B. Melamed, E. Schenfeld, and B. Sengupta, "Analysis of a Control Mechanism for a Variable Speed Processor," *IEEE Transactions on Computers*, vol. 45, no. 7, pp. 968-976, July 1996.

[9] S. F. Oberman, H. Al-Twaijry, and M. J. Flynn, "The SNAP Project: Design of Floating Point Arithmetic Units," in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, July 1997.

[10] M. Olivieri, A. Trifiletti, and A. De Gloria, "A Low-Power Microcontroller with On-Chip Self-Tuning Digital Clock Generator for Variable-Load Applications," in *Proceedings of the 1999 International Conference on Computer Design*: IEEE, 1999.

[11] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, First ed. San Mateo, California: Morgan Kaufman Publishers, Inc., 1990.

[12] A. E. Sjogren and C. J. Myers, "Interfacing Synchronous and Asynchronous Modules Within a High-Speed Pipeline," in *Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI '97)*, 1997, pp. 47-61.

[13] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720-738, June 1989.

[14] A. K. Uht, "Achieving Typical Delays in Synchronous Systems via Timing Error Toleration," Department of Electrical and Computer Engineering, University of Rhode Island, Kingston, RI, Technical Report 032000-0100, March 10, 2000. Available via http://www.ele.uri.edu/~uht.