DynaTune: Circuit-Level Optimization for Timing Speculation Considering Dynamic Path Behavior

Lu Wan Deming Chen
Electrical and Computer Engineering Department
University of Illinois at Urbana Champaign
{luwan2, dchen}@illinois.edu

ABSTRACT

Traditional circuit design focuses on optimizing the static critical paths no matter how infrequently these paths are exercised dynamically. Circuit optimization is then tuned to the worst-case conditions to guarantee error-free computation but may also lead to very inefficient designs. Recently, there are processor works that over-clock the chip to achieve higher performance to the point where timing errors occur, and then error correction is performed either through circuit-level or microarchitecture-level techniques. This approach in general is referred to as *Timing Speculation*. In this paper, we propose a new circuit optimization technique "DynaTune" for timing speculation based on the dynamic behavior of a circuit. DynaTune optimizes the most dynamically critical gates of a circuit and improves the circuit's throughput under a fixed power budget. We test this proposed technique with two timing speculation schemes - *Telescopic Unit* (TU) and *Razor Logic* (RZ). Experimental results show that applying DynaTune on the Leon3 processor can increase the throughput of critical modules by up to 13% and 20% compared to the timing-speculative and non-timing-speculative results optimized by Synopsys Design Compiler, respectively. For MCNC benchmark circuits, DynaTune combined with TU can provide 9% and 20% throughput gains on average compared to timing-speculative and non-timing-speculative results optimized by Design Compiler. When combined with RZ, DynaTune can achieve 8% and 15% throughput gains on average for above experiments.

Categories and Subject Descriptors

B.6.3 [**Hardware**]: Design Aids **–** *Optimization.*

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

Timing speculation, dual threshold voltage, throughput, performance, BDD, logic synthesis, timing analysis.

1. Introduction

There are many microarchitecture and circuit-level techniques to target higher chip performance in the research community. One recent trend in the microarchitecture domain is the application of *Timing Speculation*. Timing speculation refers to the idea of increasing the processor's clock frequency to the point where timing errors begin to occur and to equip the design with circuit or microarchitectural techniques for detecting and correcting the resulting errors. The proposed techniques include enhanced latches, a checker module, or an additional core that works in a cooperative manner. Representative works include Razor logic [1] and Paceline [2]. Razor [1] works by latching the output values of a microarchitectural pipeline stage twice: once in the normal pipeline latches, and a fraction of a cycle later in *shadow latches*. The paths to the normal latches are timing-speculative and can run into timing errors. The shadow latches are guaranteed to receive the correct values. At the end of each cycle, the shadow and normal latch values are compared. If they agree, no action is taken. Otherwise, the values in the shadow latches are used to repair the pipeline state. Paceline [2] uses two processor cores (a leader core and a checker core) to perform timing speculation. The leader is clocked at a frequency higher than the safe error-free frequency f_0, while the checker is clocked at f_0. The leader sends branch results to the checker and also pre-fetches data into a shared L2, allowing the checker to keep up. The two cores periodically exchange checkpoints of architectural state. If they disagree, the checker copies its register state to the leader for error correction. The overhead of error correction of Paceline (usually hundreds of cycles) is much higher than that of Razor (e.g., five cycles, depending on the pipeline depth of the architecture). Both of these techniques provide frequency gain that goes beyond worst-case critical path delay.

On the other hand, there is parallel effort on VLSI circuit design targeting higher throughput (generally defined as the amount of computation performed in a time unit). Representative works include the *Speculative Completion* [3] and the *Telescopic Unit* [4]. Reference [3] proposed the detection of the computation completion for asynchronous units. It associates multiple *speculative* delay models with a unit, in such a way that the completion of an operation is detected in parallel with the unit itself. The multiple models account for different (e.g., worst-case verses best-case) speeds of early completion. In [4], the authors focused their attention on synchronous circuits, which originally implement arbitrary combinational functions in a single clock cycle. They transform such units into variable-cycle implementations, which have data-dependent latency. Their clock rate can be sped-up to match the *common case*, i.e., the critical-path delay of most computations that can still be achieved in one clock cycle. Longer computations will be split over two (or more) cycles. The motivation is that a fixed-latency unit completes execution with the latency of its longest possible computation. However, a variable-latency unit adapts its latency to the length of the computation it is performing. Average throughput can be improved if the probability of a long-latency computation is much smaller than that of a short-latency one. The single-cycle frequency can be improved as well [4][6].

The major difference between Razor Logic and Telescopic Unit exists in how they handle rare cases: Razor logic *detects* error while Telescopic Unit *predicts* possible error (Section 2). And when error is detected or predicted, they have different ways to handle the error with different performance penalties. On the other

hand, they both try to speed up common cases and allow the delay of rare-case computation exceed one cycle. In that sense, both Razor Logic and Telescopic Unit can be classified as circuit level timing speculative techniques.

There are other related works. Reference [5] proposed a paradigm for low-power variation tolerant circuit design. The principal idea is to (a) isolate and predict the set of possible paths that may become critical under process variations, (b) ensure that they are activated rarely, and (c) avoid possible delay failures in the critical paths by dynamically switching to two-cycle operation (assuming all standard operations are single cycle), when they are activated. This can provide an advantage of operating the circuit at reduced supply voltage while achieving the required yield. Reference [10] proposed On-demand Selective Biasing (OSB) and Path Constraint Tuning (PCT) to do circuit-level optimization for timing speculation by leveraging a commercial tool flow. The probability property of signals is not an integral part of the commercial tool. The work reported a throughput gain of 8% with OSB and 6% with Razor Logic + PCT. However, the overall optimization flow is very time consuming (e.g., one week) due to the extensive simulation tasks incorporated into the flow.

In this study, we develop a logic optimization procedure that utilizes dual V_t (threshold voltage) cells to optimize a circuit in such a way that the most dynamically critical gates of a circuit are detected, analyzed, and optimized for higher frequency and throughput. Our contributions are as follows:

- We show that the traditional timing optimization techniques are not suitable for timing speculation. Thus we need new timing optimization techniques that can consider the dynamic behavior of a circuit.
- Given the increasing acceptance of the dual-V_t design methodology, we investigate the timing speculation schemes with dual V_t cells under a power budget.
- We propose an optimization engine *DynaTune* to improve circuit frequency and throughput using the dynamic behavior information derived from signal probabilities.
- We evaluate two timing speculation schemes – Razor Logic and Telescopic Unit – before and after DynaTune optimization, and report their individual applicability for timing speculation.

The remaining of this paper is organized as follows. In Section 2, two timing speculation schemes are reviewed. In Section 3, the motivation of DynaTune is presented. In Section 4, behavior curve is introduced and circuit optimization with DynaTune is presented. In Section 5, modules extracted from Leon3 processor [13] and a set of MCNC benchmarks are optimized by DynaTune, and the results are analyzed. Section 6 concludes the paper.

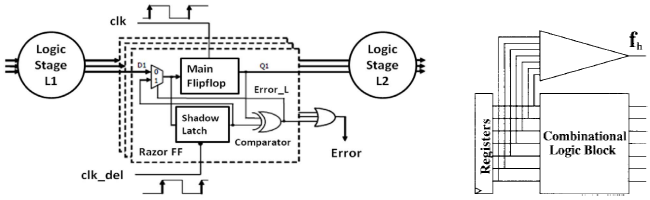
2. Timing Speculation and Definitions

Due to process and environmental variations, traditional worst case oriented design methodology uses aggressive guardbanding to put a conservative safety timing margin to a circuit to guarantee error-free operations. This mechanism, however, is pessimistic and inefficient when those worst case scenarios would only happen infrequently. *Timing Speculation* is an idea to work around this, which over-clocks the chip to achieve higher performance to the point where timing errors occur, and then error detection and correction are performed to guarantee reliability. We introduce two timing speculation schemes: Razor Logic and Telescopic Unit.

Table 1. Terminologies and abbreviations

Optimization method	
syn	Static optimization with Synopsys Design Compiler (DC), ver. 2007.12
dyn	DynaTune optimization
Timing speculation modes	
TU	Telescopic Unit timing speculation
RZ	Razor Logic timing speculation
Throughput configurations	
TRA	Synopsys DC optimized circuit working in non-timing speculative mode. The longest path L determines the cycle time.
TU+ syn	Apply TU directly on DC-optimized circuit.
RZ+ syn	Apply RZ directly on DC-optimized circuit
TU+ dyn	Apply TU on DynaTune optimized circuit
RZ+ dyn	Apply RZ on DynaTune optimized circuit
Other Terminologies	
P(signal)	The static probability of a signal: the probability of being logic 1 observed over a unit time
L(ps)	Longest path delay in pico second
TP	Throughput
T	Operating cycle time
P	The probability that the circuit can produce the correct results within T
F	Operating frequency
NL	Number of lowV_t cells in the design
Gain	Gain in percentage over performance of the circuit optimized by TRA

2.1 Razor Logic (RZ)



(a) Razor Logic (b) Telescopic Unit

Figure 1: Razor Logic and Telescopic Unit

Figure 1(a) demonstrates the concept of Razor Logic (RZ) [1]. Under normal operation where timing errors do not happen, the main FF can latch in the correct data. However, when timing errors take place, the comparator is used to catch the error. The advantage of Razor is that in most cases (with probability *P*) the main FF can latch in the correct value. When an error is detected, at the architecture level, an auxiliary logic will initiate the recovery phase by stopping the incorrectly latched value from propagating into trailing pipeline stages. The recovery phase usually takes several cycles because it needs to flush the whole pipeline stages. In our study, we use 5 cycles (recovery penalty factor r = 5) as RZ's recovery cost for the study of the Leon3 processor, which has a shallow pipeline. The throughput of RZ can be estimated below:

$$TP_{RZ} = \frac{1}{T}\left(P + \frac{1-P}{5}\right) \qquad (1)$$

where T is the clock period, and P is the probability the circuit can produce the correct results within T. Note that for each T, there exists an associated P. If (T, P) pairs are plotted by varying T, we can get the *dynamic behavior curve* of a circuit. This will be discussed in Section 4. Although RZ can tolerate timing errors, it is desirable to reduce the probability of the occurrence of these timing errors to improve the *TP* (throughput).

2.2 Telescopic Unit (TU)

TU eliminates the requirement for error recovery by predicting timing errors before the errors propagate through the FF boundary [4]. Among all the possible input vectors, some can produce the correct results in one cycle (class C_1), and some in two cycles (class C_2). As shown in Figure 1(b), a piece of auxiliary logic f_h is tagged on the primary inputs. f_h will be asserted when the input vector falls in the C_2 class, indicating a two-cycle computation. This is equivalent to a recovery penalty factor r = 2. The throughput of TU can be similarly estimated below:

$$TP_{TU} = \frac{1}{T}(P + \frac{1-P}{2}) \qquad (2)$$

2.3 Generalized Throughput Model

The throughput computation can be summarized as follows:

$$TP = (\frac{1}{T}P) + (\frac{1}{T} * \frac{1-P}{r}) \qquad (3)$$

The first term ($\frac{1}{T}P$) can be viewed as a primary throughput, and the second term $\frac{1}{T}(\frac{1-P}{r})$ can be viewed as a secondary throughput with a timing speculation penalty factor r. We observe several facts: (1) When P is close to 1, the primary throughput dominates the overall throughput. (2) For a given T, the larger P is the higher the overall throughput will be. (3) For traditional non-timing-speculative design, P = 1, however, its throughput can be much lower than timing speculative designs because its T can be much larger by using the worst-case delay as the clock period. Motivated by these observations, our algorithm, DynaTune, will try to accelerate the part in a circuit that contributes to the primary throughput by reducing T and increasing P to achieve higher overall throughput. Such optimization will also reduce the chance of recovery and increase secondary throughput as well.

Note that DynaTune is different from previous logic optimization techniques used for TU, such as in [4][6], in two ways. The first is that previous techniques focus on the error prediction logic "f_h" in Figure 1(b). In contrast, DynaTune is directly applied on the "Combinational Logic Block" in Figure 1(b) to make it timing speculation friendly. Secondly, we extend the use of TU towards microarchitecture optimization using Leon3 as our architectural model, while previous works have not touched on such topics. Similarly, for the RZ scheme, DynaTune can be applied on the "Logic Stages" between Razor FFs. Therefore, DynaTune optimized circuit can either work with TU or RZ for timing speculation.

For convenience, the definitions, terminologies and their abbreviations used in this paper are summarized in Table 1.

3. Motivation for DynaTune Optimization

Multiple threshold voltage (V_t) cell assignment is a popular technique used for timing and power optimization. Fast LowV_t cells are usually deployed along critical paths to reduce delay while slow HighV_t cells deployed on non-critical paths to reduce leakage power. However, a critical path wall can be formed by balancing all paths. We observe from experiments that the critical path wall created by such timing optimization technique can defeat the purpose of timing speculation. Therefore a new dynamic behavior-aware circuit optimization technique is needed for the purpose of timing speculation.

Consider a circuit in Figure 2. Its function is to implement a multiplexer controlled by signal sel: "o=(sel==0)?(a|b):(a|~b)". The delay for HighV_t and LowV_t types are labeled above each cell, respectively. In the following discussions, we will append the gate name with a postfix "L" if the gate is assigned LowV_t. Assume initially all gates are in HighV_t and the leakage power budget allows at most three cells to be assigned to LowV_t. Traditional optimization tries to optimize all paths by assigning U5, U6 and U7 to LowV_t. As a result, the output o of this optimized circuit can become stable at time 1.2ns via {a, U1, U5L, U7L, o} when sel=0, or at 1.3 ns via {b, U2, U3, U6L, U7L, o} when sel=1. We consider the following two modes next:

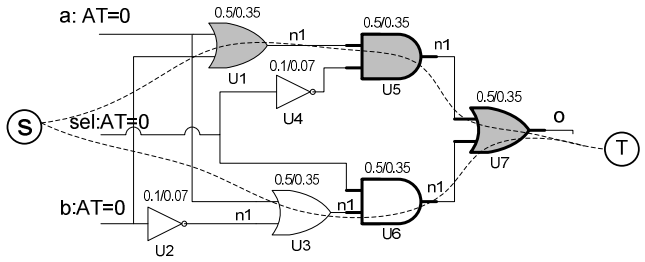


Figure 2: Example circuit to illustrate problem of static optimization for timing speculation

(1) *TRA*: This is a mode without timing speculation. The critical path (T=1.3ns) determines the minimal cycle time. Then TP = 1/T = 1000/1.3 = 769.2 MOPS.

(2) Timing speculation mode (*TU+ syn* / *RZ+ syn*): since the output can only be stable at 1.2ns or 1.3ns, we can at most over-clock this circuit with T=1.2ns. Assume we observe *P(sel)*=0.1 (the probability of sensitizing the top path is 0.9=1-0.1), then the TP can be calculated with equation (3).

TU+ syn : TP = 1000/1.2*(0.9+0.1/2) = 791.7 MOPS
RZ+ syn : TP = 1000/1.2*(0.9+0.1/5) = 766.7 MOPS

When timing optimization is done in a balanced manner, even in the best case where *TU+ syn* combination is used, the throughput can only improve by 3% over that of *TRA*.

On the contrary, with the knowledge of *P(sel)*=0.1 a priori, DynaTune can optimize the circuit in a probabilistic way to achieve a better solution for timing speculation by assigning U1,U5,U7 to LowV_t. It produces two unbalanced paths, 1.05ns {a, U1L, U5L, U7L, o} and 1.45ns {b, U2, U3, U6, U7L, o}. If the circuit is clocked at T=1.05ns, o will produce correct result whenever the top path is activated. The throughputs for *TU+ syn* and *RZ+ syn* are 18% and 14% higher than *TRA*:

TU+ dyn: TP = 1000/1.05*(0.9+0.1/2)=904.8 MOPS (18% gain)
RZ+ dyn: TP = 1000/1.05*(0.9+0.1/5)=876.2 MOPS (14% gain)

We formulate our problem as follows: given (1) an initial circuit implemented with HighV_t cells, (2) a leakage power budget, and (3) the signal probabilities of primary inputs, assign LowV_t cells within the leakage power budget with the objective to obtain the highest peak throughput through timing speculation.

4. DynaTune Optimization

The main idea of DynaTune is to change the dynamic behavior of a circuit by reducing the delay of long path that are commonly exercised and judiciously allowing some rarely exercised critical path to fail the one cycle timing requirement. A *behavior curve* is used as the guidance throughout the DynaTune optimization. A behavior curve is a curve with axis T and P (refer to the definitions in Table 1). By varying T, one can plot all (T, P) pairs to get the behavior curve representing the dynamic behavior of a circuit. An example for a 32-bit adder is shown in Figure 3. A *throughput curve* can be derived from behavior curve using equation (3). And by varying T along the x axis, we can determine

T^* and its associated F^* for the peak throughput. The pair of (T^*, F^*) will be called the *peak operating point*.

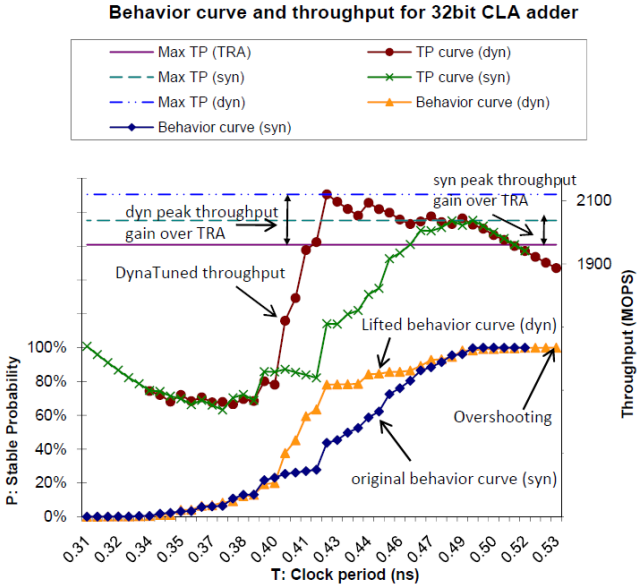


Figure 3: Behavior and throughput curves in TU timing speculation for DynaTune and Synopsys DC optimized circuits

In Figure 3, behavior curves and throughput curves for the adder optimized by Synopsys DC and DynaTune are plotted (Table 1 for definitions). From top to bottom, three horizontal lines indicate the peak throughput for *dyn, syn,* using TU as the timing speculation scheme, and *TRA*. The lowest two curves are the behavior curves of *syn* (marked with ♦) and *dyn* (marked with ▲). DynaTune optimization can create a separation in the middle (0.40ns-0.46ns) of the behavior curves by sacrificing some uncommonly exercised critical paths (the longest critical path for *syn* is 0.52ns while the longest path of *dyn* is 0.54ns). The lifted probability for the common cases in the middle range of the behavior curve can be translated into throughput gain according to equation (3). As can be seen from the upper two throughput curves, the peak of the throughput curve (marked with •), also the corresponding *F,* of *dyn* are higher than those on throughput curve (marked with ×) of *syn*. In fact, in this example, *dyn* circuit can produce the highest throughput (2160 MOPS). Throughput curves for RZ scheme can be derived in the same way using equation (3) with a penalty factor $r = 5$.

One can get the behavior curve through timed simulation, but that would be very time consuming. To avoid this, we derive the behavior curve using *Timed Characteristic Function* (TCF).

4.1 Timed Characteristic Function

TCF was proposed in [7]. It was originally used for ATPG purposes. The goal of deriving TCF is to find an input vector which can sensitize a circuit and make a given output stable before (after) a given timing requirement *t*. The local TCF function for a gate output ***n*** is denoted as: *TCF(**n**, t-)*. *TCF(**n**, t-)* contains all input vectors which make the output ***n*** stable no later than *t*. Local TCF at a given point can be derived recursively: starting from node ***n*** and back-tracing the circuit in ***n***'s fan-in cone until the primary inputs are reached. Instead of using the TCF formulas from [7], where the stable value (logic 0 or 1) is explicitly used as one parameter of the TCF, we are only interested in the stable time for the gate output *c*, and the actual stable value of *c* does not matter to us. Thus, we derive the TCF in the following forms:

The local *TCF(c, t-)* for an AND gate (c=a∧b) can be written as:

$$TCF(c, t-) = CF(a') * CF(b) * TCF(a,(t-d)-)+$$
$$CF(a) * CF(b') * TCF(b,(t-d)-)+$$
$$CF(a') * CF(b') * [TCF(a,(t-d)-)+TCF(b,(t-d)-)] +$$
$$CF(a) * CF(b) * TCF(a,(t-d)-) * TCF(b,(t-d)-) \quad (4)$$

Where *CF(x)* is the Characteristic Function for the output *x*. Characteristic function *CF(x)* is a logic representation for *x* which contains all primary input vectors that make *x* stable at logic 1, while *CF(x')* contains all primary input vectors that eventually make *x* stable at logic 0. *CF* can be computed using BDD for each gate [8]. In Equation (4), each term in the summation actually encodes one possible input combination for the gate. For example, the first term "*CF(a') * CF(b) * TCF(a,(t-d)-)*" encodes all primary input vectors which make output *c* stable at *t-* under the condition *a*=0 and *b*=1. Since *a* has the controlling value of AND gate (logic 0), if and only if *a* is stable at (*t-d*)-, output *c* can be stable after a gate delay *d*, and then *TCF(c,t-)* is guaranteed. Similarly, the other three input combinations are encoded in the other three terms, respectively.

TCF(c,t-) for an OR gate (c=a∨b) can be written in a similar way:

$$TCF(c, t-) = CF(a') * CF(b) * TCF(b,(t-d)-)+$$
$$CF(a) * CF(b') * TCF(a,(t-d)-)+$$
$$CF(a') * CF(b') * TCF(a,(t-d)-) * TCF(b,(t-d)-) +$$
$$CF(a) * CF(b) * [TCF(a,(t-d)-)+TCF(b,(t-d)-)] \quad (5)$$

This recursive TCF derivation can be extended for complex gates.

4.2 Behavior Curve Generation

The global TCF function for a whole circuit is defined as *TCF(C,t-)*, which contains all primary input vectors that make all outputs stable no later than time *t* for the circuit *C*. We use lower case letter in local TCF and upper case letter in global TCF. We can derive global *TCF(C,t-)* in the following manner. First, early arrival time (min_AT) and late arrival time (max_AT) are calculated for every gate in circuit *C*. Then local TCF for each PO is derived using recursive relations mentioned in Section 4.1 by calling the function *TCF(g,t)* in Algorithm 1 for different gate types. The recursion call can terminate at gate *g* early if the timing requirement *t* either can never be satisfied *(t < min_AT(g))* or can always be satisfied *(t ≥ max_AT(g))*. At last, the local TCFs of all POs are *AND'ed* together to produce the global TCF for the circuit. Note that global TCF is only used for deriving the behavior curve to guide DynaTune optimization. None of the aforementioned TCF functions with the global AND operation will become part of the final optimized circuit. Thus they won't incur any overhead for the final result. The probability of *TCF(C,t-)* being evaluated to logic 1 (the static probability of *TCF(C,t-)*) represents the probability that the circuit would produce all stable output signals no later than *t*. *TCF(C,t-)* is represented in ROBDD. We use an algorithm proposed in [12],

Algorithm 1 TCF

Inputs: *g* : gate
t : operating point (timing requirement)
Output: an *ROBDD* representing *TCF(g,t-)*

Begin
if (*TCF(g,t)* exists) **then return** *TCF(g,t)* ;
if (*t* < min_AT(*g*)) **then return** *bdd_zero* ;
if (*t* ≥ max_AT(g)) **then return** *bdd_one* ;
switch (*g*)
case AND: recursively call TCF using Equation (4)
case OR: recursively call TCF using Equation (5)
…
End switch

which is widely used in vector-less power analysis, to calculate the static probability of *TCF(C,t-)*, which is captured by the probability of the ROBDD evaluating to logic 1. By varying the operating point *t*, the desired (*t,P*) pairs can be computed for the behavior curve, where *P = Prob(TCF(C,t-))*.

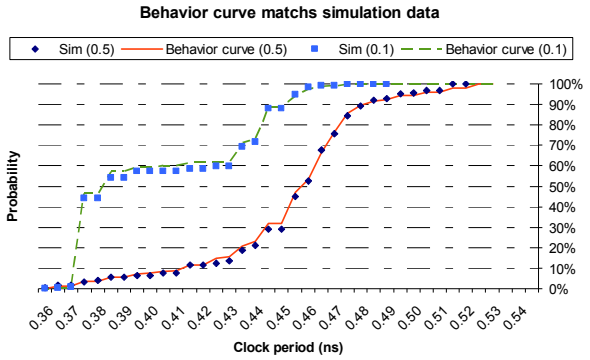


Figure 4: Behavior Curve for a 32bit Adder

Figure 4 shows the derived behavior curves and the simulation data for a 32-bit Carry-Look-Ahead (CLA) adder. We use the NCSim simulator from Cadence to do timed simulation for this adder in floating mode [11], under which the internal gates and nets take arbitrary value "X" before a single input vector is applied to settle down their values. We set the static probability for PIs as 0.5 and 0.1 for this study. Intuitively, when the static probability is 0.5, the carryout signal has more chance to propagate for a longer distance on the critical path along the carry chain, thus making the whole circuit stable late. Conversely, when it is 0.1, there is more chance the carry bit will be terminated early, enabling the circuit to stabilize early. This is verified by the data shown in Figure 4. Also, the derived behavior curves match the simulation data very closely.

4.3 Step-wise Optimization

DynaTune optimizes a circuit by iterating three steps: 1) Scan the behavior curve. 2) At the operating point *t* where the behavior curve shows a noticeable probability drop, DynaTune investigates the cells' individual contributions to such a probability drop at *t*. The motivation behind this is that we will use low V_t for those cells that cause non-trivial probability drops so these cells can be sped up to work against the probability drops to improve the overall throughput and frequency of the design. 3) A min-cut max-flow algorithm is used to find the cells mentioned above. We introduce the details next.

Given the behavior curve for *C*, we use an outer loop to scan the behavior curve with a fixed interval t_s starting from one end of the operating point (t_{high}=*max_delay*) to the other end of the operating point (t_{low}=*max_delay/2*). Setting t_{low} to *max_delay/2* is because that TU needs to guarantee all signal propagations to finish within at most 2 cycles when f_h is asserted. At each scan step t_{curr}, DynaTune_step (Algorithm 2) is called to investigate individual cell's contribution and to find the node cut-set at a proper optimization point (*t'* in Algorithm 2). To achieve such a goal, a **do** loop is used to find *t'* that has a probability drop on the behavior curve larger than a threshold *min_*Δ (*min_*Δ = 0.2 in our experiment). To save runtime, Algorithm 2 filters out two kinds of cells through the filter_candidate procedure: (1) cells that have been assigned $LowV_t$ and (2) cells whose local TCFs do not change when the circuit is clocked at t_{curr} and *t'*. This is because those cells with no local TCF changes do not contribute to the probability drop. After a flow network is constructed with the remaining cells by procedure "create_network", a cell's individual contribution in this network is investigated by setting it to low V_t temporarily and updating the local TCFs of the affected fan-in and fan-out cones. Then the probability difference on the behavior curve between the two cases (i.e., before and after setting low V_t) is recorded as the cell's probability contribution. We perform such evaluation for each cell in the network. Next the capacity of each cell (i.e., a network node) is set using the inverse of the square root of each node's contribution. We then call the mincut procedure that performs a max-flow min-cut algorithm.

If the mincut procedure returns a non-empty node cut-set (a set of candidates that most likely caused the probability drop), we will perform "LowVt_assign" next, which is the key step to identify the cells that will potentially optimize the most commonly exercised critical paths if assigned $LowV_t$. We do this by comparing a candidate cell's probability contribution to a filtering factor "α×avg_contri". Only when a cell's contribution is larger than this threshold, this cell is finally assigned $LowV_t$. Here, α is an empirically determined value (0.25 in our experiment), and "avg_contri" is calculated as a moving average of the probability contribution from all $LowV_t$ cells assigned by DynaTune so far. Such a filtering process is very important because it only picks the cells that are on commonly exercised critical paths for $LowV_t$ assignment and allows rarely exercised paths to go beyond one cycle timing requirement. Since it is not worth spending precious $LowV_t$ budget on those cells who contribute insignificantly even they are in the cut-set, filtering them out can save leakage power budget for others who could contribute more in later iterations. Note that when α is set to zero (no filtering), DynaTune will degenerate into a traditional timing optimization technique which tries to reduce delays for all paths.

After these steps, timing information as well as the cost of leakage power are updated. The outer loop continues to move to a lower operating point $t_{curr} = t_{curr} - t_s$ until it reaches t_{low}. The loop will also end when the leakage power cost exceeds the leakage power budget.

Algorithm 2: DynaTune_step

Inputs: *C*: a circuit netlist
t_{curr}: timing scan point
Output: {*lowVt_cells*}: a set of cells for low V_t assignment
Begin
p = probability(*C*, t_{curr}); //probability at t_{curr} from the behavior curve
t' = t_{curr} ; Δ = 0;
while (Δ < *min_*Δ)
t' = *t'* – *L/100*; // inner loop to find the probability drop above
// *min_*Δ; *L* is the longest path delay.
Δ = *p* - probability(*C*, *t'*); // probability drop
End while
{*candidates*} = filter_candidate(*C*);
network = create_network({*candidates*});
for each *n* in {*network*} **do**
contri = contribution(*C*, *n*, *t'*);
capacity = 100/sqrt(*contri*);
assign_capacity(*n*, *capacity*);
End for
{*candidate_cells*} = mincut(*network*);
{*lowVt_cells*} = LowVt_assign({*candidate_cells*}, α×*avg_contri*);
return {*lowVt_cells*};
End

We will use the circuit in Figure 2 as an example to explain the algorithm. Initially, all cells are set as $HighV_t$ cells. Two delay values are shown above each cell for $HighV_t$ / $LowV_t$ in unit of nanosecond. We assume that independent PIs a, b and sel have static probabilities of P(a)=0.5, P(b)=0.5 and P(sel)=0.1. When all cells are $HighV_t$, the critical path delay is 1.6ns, consisting of {b → U2 → U3 → U6 → U7 → o}. We compute the behavior curve using all $HighV_t$ cells.

We first show how the behavior curve is computed. The first operating point of interest is t = 1.6ns. Since this is the critical path delay, Algorithm 1 terminates early for all POs with the return value *bdd_one* (logic 1 in the BDD package). Then, the global *TCF(C,1.6-)* = 1, indicating that all input vectors stabilize the circuit by 1.6ns. Thus we get a (*T,P*) pair (1.6,1).

Assume later on we reach t = 1.55ns. Starting from PO *o* with timing requirement t = 1.55ns, one branch for tracing towards the PI in algorithm 1 is {o→U7→U5→….}. When we reach U5 along this path, we find its late arrival time max_AT(U5) = 1.0ns which is smaller than its required arrival time (RAT) 1.05ns = 1.55 – 0.5. Then this branch of algorithm 1's recursive call can be terminated early with a return value of *bdd_one*, meaning that any controlling value of U7 (logic 1 in this case) produced by U5 can make *o* stable by t = 1.55ns. This condition is ((a+b)×sel'). Similarly, after tracing back from other branches and operating on those local TCFs, the global TCF(C,1.55-) produced by DynaTune is (sel'+a×sel) meaning whenever (sel'+a×sel)==1, *o* can become stable no later than 1.55ns. Since P(sel'+a×sel) = 0.9+0.5*0.1 = 0.95, the associated P for 1.55ns is 0.95. In this way, we get another (*T,P*) pair (1.55, 0.95) on the behavior curve.

Assume the next operating point of interest is t = 1.35ns. Algorithm 1 terminates early with *bdd_zero* when back-tracing reaches U1, because min_AT(U1)=0.5ns is greater than RAT of U1 (0.35ns=1.35-0.5-0.5). Similarly, it also terminates early at U3. In fact the global TCF(C,1.35-) = sel×sel' = Φ, meaning no input vector can make PO stable by 1.35ns. Thus a (*T,P*) pair = (1.35,0) is computed. As one can see, by varying *t*, we can derive the complete behavior curve.

Now, we illustrate how Algorithm 2 works. At t = 1.35ns, we reach a significant probability drop (bigger than *min*_Δ), and we need to recover from this probability drop by turning some cells into faster LowV_t cells. Because turning U2 or U4 into LowV_t makes no difference for the overall probability drop procedure "filter_candidate" will filter them out first. Then a flow network (dashed curves in Figure 2) is constructed. Initially, every cell is in HighV_t. Turning any gate on the top path {a → U1 → U5 → U7 → o}(1.5ns) to lowV_t can reduce this path delay to 1.35ns. In addition, turning U7 into LowV_t can also make the output *o* stable by 1.35ns whenever the path {a → U3 → U6 → U7} is activated with the condition (a×sel=1) (with a probability 0.05). Combining with the probability 0.9 of activating the top path, turning U7 to LowV_t makes the largest probability contribution 0.95. And since the capacity in the network flow is the inverse of the square root of contribution, U7 will have the smallest capacity. Because none of the cells has been assigned LowV_t so far, the avg_contri is 0. Then {U7} is returned by Algorithm 2 and assigned to LowV_t, and the avg_contri is updated to 0.95.

Later on, the flow network will exclude U7 because it has already been assigned LowV_t. And a cut-set {U5, U6} will be returned by "mincut", Since signal sel rarely activates the path {b → U2 → U3 → U6 → U7 → o} (P(sel)=0.1), {U6} will be filtered out by "LowVt_assign" when its probability contribution is compared with α×avg_contri (0.25×0.95). At the end of the DynaTune optimization, U1, U5 and U7 will eventually be assigned LowV_t.

5. Experiments and Analysis

We implemented DynaTune in SIS [14]. Following the TSMC reference design flow [15], the baseline Dual V_t benchmark circuits were compiled by Synopsys Design Compiler (DC) with a TSMC 65nm library (*TRA*). (Refer to Table 1 for various definitions used in this subsection.) The same circuits were used as inputs to DynaTune, but with all low V_t cells converted to high V_t. DynaTune then optimizes the circuit and selectively turns cells to LowV_t within the same leakage power budget used by Synopsys DC. We apply DynaTune on Leon3 modules and MCNC benchmark circuits and report various comparison results.

5.1 DynaTune on Leon3 processor

We use a benchmark suit compiled by John Hennessy from Stanford, including a variety of applications: three sorting programs (quick, bubble, tree); two matrix multiplication programs (intmm, mm); a FFT application (oscar); a permutation program (perm); and several puzzle programs (tower, puzzle and queens). We randomly choose one application from each category to form our training set consisting of quick, intmm, perm, oscar, and queens. We run these five training programs on Leon3 at RTL level with the Cadence simulation tool – "NCSim" to characterize initial static probabilities for PIs. With the simulation results, a Value Change Dump analyzing tool "lpsvcd2tcf.exe" provided by Cadence is used to extract the static probability for the Leon3 modules. Then DynaTune optimizes these modules based on the extracted static probability information. All ten benchmark programs are then used as the testing set on both DC optimized circuit (*syn*) and DynaTune optimized circuit (*dyn*) to collect performance results.

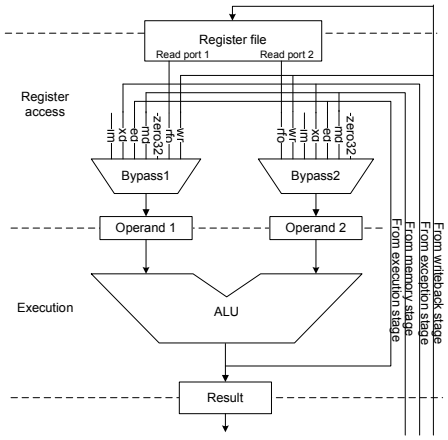


Figure 5. Register access and execution stages of Leon3 processor

We focus our study on the logic surrounding the execution unit, because execution unit is usually the most timing critical part in the whole processor design. Two pipeline stages of the Leon3 – register access and execution – are shown in Figure 5. The ALU performs addition/subtraction and a variety of logic operations. In the register access stage, the most time consuming combinational logic parts are the two lanes of bypassing logic. These two lanes of bypassing logic have identical physical design. Functionally, the bypassing module chooses one signal to output from 7 possible sources: write-back register (wr), immediate data (im), register file output data (rfd), execution data (ed), exception data (xd), memory data (md), and all "0" (zero). We will evaluate two aspects of the experimental data. Firstly, we allow individual modules to run at their individual peak throughput points to study the full effect of DynaTune. As a result, each individual module may report a different operating frequency *F*. Secondly, synchronized design methodology requires that every pipeline stage runs under the same frequency. As a result, the frequency of the most critical module (the minimum frequency value among all the modules) will determine the speed and the throughput of the entire processor.

First of all, we would like to validate the static probability we obtained through the training set using the operand2 bypassing lane as a case study. As shown in Figure 6, the static probability

extracted from the training set (dashed curve) actually matches the static probability of the test set (solid curve) well. Specifically, all applications tend to use im and rfd as its source for operand2. This application-independent behavior bias of a circuit can be viewed as common probabilistic properties, which provide us opportunity to optimize a module based on the dynamic circuit behavior regardless of applications.

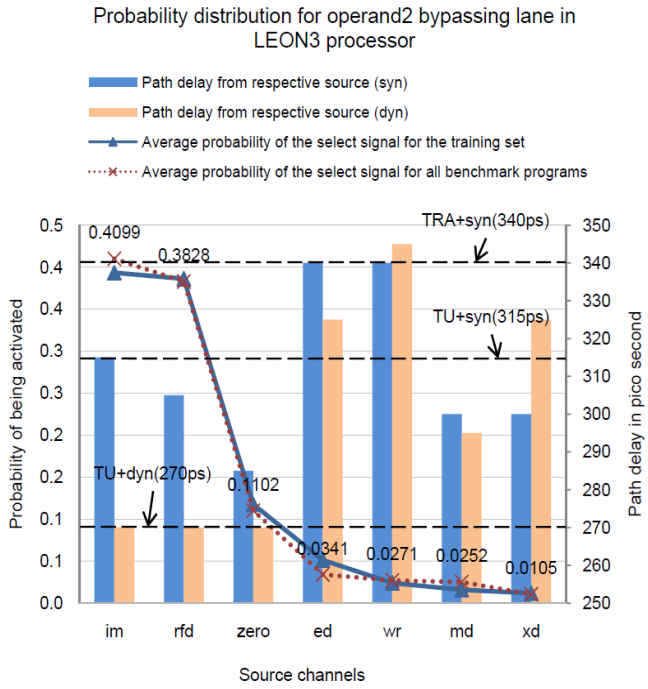


Figure 6. Probabilistic property and optimization results of operand2 bypassing lane in Leon3 CPU

A similar phenomenon is reported in [10] that although the dynamic optimization is performed for a representative subset of applications, a broader range of applications can benefit from this optimization in term of throughput. The reason behind this was once pointed out by the authors of [9]: the interaction between software program and processor can make the hardware logic reveal common probabilistic properties for a range of applications. Considering a 32-bit adder, various types of applications have the tendency to use the lower bits of the adder, thus the carry-out bit tends to propagate along a shorter distance and its activity concentrates on the lower bits. In general, the overlapping properties enjoyed across different application domains provide a great potential for circuit-level optimization towards timing speculation.

The timing results of both *syn* and *dyn* circuits are reported in Figure 6. Each bar represents the longest path delay from a source (in x axis) to the output. The longest path of the *syn* circuit (optimized with DC) is 340ps (from ed/wr to output), equivalent to a throughput of 2941 MOPS for a *TRA* configuration. If TU is directly applied on this circuit (*TU+ syn*), the peak throughput can be achieved at T=315ps, and computation can finish in one cycle whenever im, rfd, zero, md, and xd are selected with a total P=0.94. The overall peak throughput is 3079 MOPS. This is merely 5% higher than the peak throughput offered by the *TRA* configuration.

In contrast, DynaTune (*TU+ dyn*) can significantly boost the throughput by selectively reducing the path delay from those commonly used sources, i.e., im, rfo and zero, while sacrificing the delay reduction on rarely exercised paths. When clocked at 270ps, the output can become stable whenever they are activated with a total P=0.903. As a result, the overall peak throughput is 3524 MOPS, which is 20% higher than the peak throughput of (*TRA*).

Tables 2 and 3 report the details of the experimental results for Leon3. Table 2 compares frequency, throughput and activation probability among (*TRA*), (*RZ+syn*), and (*RZ+dyn*). On average, (*RZ+dyn*) offers 13% and 9% better throughput over (*TRA*) and (*RZ+syn*), respectively. In Table 3, on average, (*TU+dyn*) offers 20% and 13% better throughput over (*TRA*) and (*TU+syn*), respectively. *T**, *P**, and *F** are the *T*, *P*, and *F* values under the peak throughput condition. In general, using *TU* can produce higher peak throughput than *RZ* because of its smaller penalty factor. Given the great interest Razor Logic receives from the research community, we believe Telescopic Unit can provide further improvement of the overall processor performance along the similar design methodology provided by Razor: targeting the commonly exercised portion in the circuit. DynaTune goes one more step by actively optimizing these commonly exercised portions and is also general enough to work for both RZ and TU type of architectures.

The secondary effect of DynaTune is to enable the critical module to be over-clocked and run faster. This can make certain critical modules no longer the performance limiter of the whole design. As shown in Tables 2 and 3, the most time consuming module in Leon3 execution stage is the adder. Designing under the traditional way, the delay of the adder determines the cycle time and puts an upper limit of 1942 MHz for the whole design. After applying DynaTune, this upper limit is raised by 36% to 2632 MHz. And the throughput gains for *TU+dyn* and *RZ+dyn* are 25% and 19% for the adder, respectively. In other words, we manage to transform 70% (53%) of the frequency increase into real performance gain – throughput. Note that because the processor is a synchronous design, with timing speculation the final frequency of the processor can break the 1942 MHz barrier and be clocked up to 2632 MHz. Also, if we just use the adder's results, the effect of DynaTune is even larger than the average values reported in the previous paragraph.

It should be pointed out that for cases where no obvious bias behavior exists, DynaTune's improvement is little. For example, the 32-bit adder in Table 4 assumes a static probability of 0.5 for all PIs, and the activity of the propagation chain no longer concentrates on the lower bits (Figure 4). As a result, DynaTune does not report much improvement for this adder.

5.2 DynaTune on MCNC benchmarks

To demonstrate that DynaTune optimization is generally applicable to other types of circuits, we apply it to MCNC benchmarks. Since there are no input vectors available for these benchmarks, we set a default static probability of 0.5 on every PI. For *TU+syn* (*RZ+syn*), these MCNC circuits can speculatively operate at a frequency that is 24% (21%) higher than the frequency determined by *TRA* on average. After taking the timing speculation penalty into account, 46% (33%) of the increased frequency can eventually be transformed into a real 11% (7%) throughput gain. In contrast, DynaTune can transform the speculative operating frequency into throughput more effectively: for *TU+dyn (RZ+dyn)*, the average operating frequency is increased by 32% (25%) over that of *TRA*, resulting an average throughput gain of 20% (15%). In other words, 60% of the frequency increase can be effectively transformed into throughput gain. The actual numbers of $LowV_t$ cells used in each circuit are also shown in Table 4. All the benchmarks, including the modules from Leon3 finish within 15 minutes on a desktop PC, Core2 Duo, 2.0 GHz with 2G memory.

6. Conclusions and Future Work

We proposed DynaTune, a new performance optimization technique based on the dynamic behavior of the circuit. DynaTune uses the behavior curve derived from timed characteristic function to decide which part of the circuit is the best candidate for timing optimization. DynaTune spends more low V_t cells on the most commonly exercised critical paths and allows the least commonly exercised critical paths to spend more cycles to finish. Experimental results showed that DynaTune can provide significant throughput gains over previous timing speculative techniques. Although DynaTune can practically work on modules of a real processor design, due to the inherent complexity of ROBDD, we believe DynaTune would run into scalability issues for large-scale circuits. Future work will study how to perform this optimization in a scalable way.

7. Acknowledgement

We would like to thank Dr. Josep Torrellas and Dr. Brian Greskamp of Computer Science Department of UIUC for helpful discussions. This work is partially supported by SRC 2007-1592, NSF CCF 07-02501, and Sun Microsystems, Inc.

8. References

[1] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Zeisler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation", *Annual Intl. Symp. on Microarchitecture,* Dec 2003.

[2] B. Greskamp and J. Torrellas. "Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking", *Intl. Conf. on Parallel Architecture and Compilation Techniques*, 2007.

[3] S. Nowick, "Design of a low-latency asynchronous adder using speculative completion", *IEE Proc. Comput. Digital Techniques*, vol. 143, pp. 301–307, Sept. 1996.

[4] L. Benini, E. Macii, M. Poncino, and G. De Micheli, "Telescopic Units: A New Paradigm for Performance Optimization of VLSI Designs", *IEEE Trans. on CAD*, Vol. 17, No. 3, March 1998.

[5] S. Ghosh, S. Bhunia, and K. Roy, "A New Paradigm for Low-power, Variation-Tolerant Circuit Synthesis Using Critical Path Isolation", *ICCAD*, Nov., 2006.

[6] Y. S. Su, D. C. Wang, S. C. Chang, and M. Marek-Sadowska, "An Efficient Mechanism for Performance Optimization of Variable-Latency Designs", *Design Automation Conference*, 2007.

[7] Y. M. Kuo, Y. L. Chang, and S. C. Chang, "Efficient Boolean Characteristic Function for Fast Timed ATPG", *ICCAD*, 2006.

[8] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, C-35-8, pp. 677-691, August, 1986.

[9] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, "Opportunities and challenges for better than worst-case design," In *Proc. of the 2005 Asia South Pacific Design Automation* 2005.

[10] B. Greskamp, et al. "Blueshift: Designing processors for timing speculation from the ground up," *High Performance Computer Architecture, 2009.* vol., no., pp.213-224, 14-18 Feb. 2009

[11] S. Devadas, et al. "Computation of floating mode delay in combinational circuits: practice and implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Dec 1993

[12] F. N. Najm, "Transition Density, A Stochastic Measure of Activity in Digital Circuits", *Design Automation Conf.*, 1991.

[13] Aeroflex Gaisler, "Leon3 processor" http://www.gaisler.com/cms/

[14] SIS unofficial release 1.3: http://embedded.eecs.berkeley.edu/Alumni/pchong/sis.html

[15] "TSMC reference flow 7.0", *http://www.tsmc.com*

Table 2. RZ Comparison Results for Leon3 modules

Modules	TRA			RZ+ syn							RZ+ dyn						
	T (ps)	F (MHz)	TP (MOPS)	L (ps)	T* (ps)	F* (MHz)	F* gain	P*	TP (MOPS)	TP gain	L (ps)	T* (ps)	F* (MHz)	F* gain	P*	TP (MOPS)	TP gain
adder32x32	515	1942	1942	515	465	2151	11%	0.999	2149	11%	555	380	2632	36%	0.847	2309	19%
op1 bypassing	340	2941	2941	340	340	2941	0%	1.000	2941	0%	355	260	3846	31%	0.754	3089	5%
op2 bypassing	340	2941	2941	340	315	3175	8%	0.940	3022	3%	345	270	3704	26%	0.903	3416	16%
average							6%			4%				31%			13%

Table 3. TU Comparison Results for Leon3 modules

Modules	TRA			TU+ syn							TU+ dyn						
	T (ps)	F (MHz)	TP (MOPS)	L (ps)	T* (ps)	F* (MHz)	F* gain	P*	TP (MOPS)	TP gain	L (ps)	T* (ps)	F* (MHz)	F* gain	P*	TP (MOPS)	TP gain
adder32x32	515	1942	1942	515	414	2415	24%	0.804	2179	12%	555	380	2632	36%	0.847	2430	25%
op1 bypassing	340	2941	2941	340	305	3279	11%	0.844	3023	3%	355	260	3846	31%	0.754	3373	15%
op2 bypassing	340	2941	2941	340	315	3175	8%	0.940	3079	5%	345	270	3704	26%	0.903	3524	20%
average							15%			7%				31%			20%

Table 4. Comparison Results for MCNC Benchmarks (PI Static Prob. = 0.5)

	LowVt Cell Usage			Non Timing Speculation		Timing Speculation (TS) Throughput (Frequency in MHz Throughput in GOPS)													
	syn	dyn	Δ			syn								dyn					
				TRA(baseline)		TU				RZ				TU				RZ			
Circuits	NL	NL	NLΔ	F*	TP	F*	F Δ	TP	TP Gain	F*	F Δ	TP	TP Gain	F*	F Δ	TP	TP Gain	F*	F Δ	TP	TP Gain
adder32	121	122	0.8%	1920	1.92	2105	10%	2.00	4%	2041	6%	1.95	1%	2222	16%	2.07	8%	2128	11%	2.03	6%
b9.	25	28	12.0%	4350	4.35	4762	9%	4.50	4%	4348	0%	4.35	0%	5128	18%	4.84	11%	5128	18%	4.67	7%
example2.	44	43	-2.3%	2820	2.82	3175	13%	3.15	12%	3175	13%	3.14	12%	4000	42%	3.61	28%	3704	31%	3.43	22%
vda.	198	194	-2.0%	2330	2.33	2410	3%	2.39	3%	2410	3%	2.38	2%	2564	10%	2.48	7%	2439	5%	2.46	6%
dalu.	179	175	-2.2%	1600	1.60	2299	44%	1.80	12%	1802	13%	1.73	8%	2500	56%	1.96	23%	2083	30%	1.89	18%
t481.	18	19	5.6%	2900	2.90	2899	0%	2.90	0%	2899	0%	2.90	0%	3125	8%	3.09	7%	3125	8%	3.08	6%
x3.	73	72	-1.4%	2820	2.82	3125	11%	2.94	4%	2941	4%	2.88	2%	3390	20%	3.07	9%	3125	11%	2.94	4%
x1.	58	56	-3.4%	2820	2.82	4082	45%	3.40	21%	3922	39%	3.06	9%	4167	48%	3.71	32%	4082	45%	3.47	23%
apex7.	55	56	1.8%	2940	2.94	3077	5%	3.03	3%	3077	5%	3.00	2%	3279	12%	3.09	5%	3175	8%	2.99	2%
cordic	34	32	-5.9%	2670	2.67	4545	70%	3.44	29%	5263	97%	3.22	21%	4444	66%	3.91	47%	4167	56%	3.6	35%
comp.	54	36	-33.3%	2200	2.20	3333	52%	2.91	33%	3333	52%	2.66	21%	3509	59%	3.15	43%	3448	57%	2.97	35%
Average:							**24%**		**11%**		**21%**		**7%**		**32%**		**20%**		**25%**		**15%**