EVAL: Utilizing Processors with Variation-Induced Timing Errors*

Smruti Sarangi, **Brian Greskamp**, **Abhishek Tiwari**, and **Josep Torrellas**
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

Parameter variation in integrated circuits causes sections of a chip to be slower than others. If, to prevent any resulting timing errors, we design processors for worst-case parameter values, we may lose substantial performance. An alternate approach explored in this paper is to design for closer to nominal values, and provide some transistor budget to tolerate unavoidable variation-induced errors.

To assess this approach, this paper first presents a novel framework that shows how microarchitecture techniques can trade off variation-induced errors for power and processor frequency. Then, the paper introduces an effective technique to maximize performance and minimize power in the presence of variation-induced errors, namely High-Dimensional *dynamic adaptation. For efficiency, the technique is implemented using a machine-learning algorithm. The results show that our best configuration increases processor frequency by 56% on average, allowing the processor to cycle 21% faster than without variation. Processor performance increases by 40% on average, resulting in a performance that is 14% higher than without variation — at only a 10.6% area cost.*

1 Introduction

As integrated circuit technology continues to scale, the next major challenge faced by high-performance processor designers is parameter variation [30]: the fact that Process, Voltage, and Temperature (PVT) values change from their nominal specifications. Designing processors under variation is harder because they have to work under a wide range of conditions.

One of the most harmful effects of variation is that some sections of the chip are slower than others — either because their transistors are intrinsically slower or because temperature or supply voltage conditions render them so. Logic paths in these sections may take too long to propagate signals and, as a result, induce timing errors. On current trends, designers in upcoming technology generations may have to create overly conservative designs to avoid risking these errors. It has been suggested that parameter variation may wipe out a sizable fraction of the potential gains of future technology generations [2].

An alternative scenario, which this paper explores, is that in a high-variability environment, cost-effective processors may be designed to tolerate errors due to parameter variation. In this case, processors may be designed not for worst-case parameter values but for closer to nominal-case parameter values — and provide some transistor budget to tolerate the resulting variation-induced errors. The result may be a higher-performing processor and/or a cheaper manufacturing process — in short, a more cost-effective design.

To explore this scenario, it is first necessary to consider how parameter variation induces timing errors in high-performance processors. Second, while we could reuse existing fault-tolerant architectures to handle these errors, it is important to understand how the rate of these errors can be traded-off for other quantities, such as processor power or frequency. Finally, we need to identify microarchitecture techniques that minimize such errors, possibly also affecting the power and frequency of the processor. This paper addresses the last two challenges and makes two contributions.

First, it introduces a novel framework called EVAL (Environment for Variation-Afflicted Logic) to understand how processors can tolerate and mitigate variation-induced errors. In EVAL, microarchitecture techniques can trade off error rate for power and processor frequency.

Second, this paper presents *High-Dimensional* dynamic adaptation, an effective microarchitecture technique to maximize processor performance and minimize power in the presence of variation-induced timing errors. The parameters adapted include the processor frequency, multiple voltages, and two processor structures. To efficiently support this technique, we propose an implementation based on machine learning.

Our results show that, under variation-induced timing errors, high-dimensional dynamic adaptation is feasible and effective. With no support for handling variation, a processor can only cycle at 78% of its no-variation frequency. However, by dynamically adapting processor frequency, per-subsystem voltages, issue queue size, and functional-unit structure, the processor increases its frequency by 56% on average — effectively cycling 21% faster than under no variation. Processor performance increases by 40% on average (or 14% over the no-variation scenario), always within error-rate, power, and temperature constraints. The area overhead of this technique is only 10.6% of the processor area.

This paper is organized as follows. Section 2 gives a background; Section 3 presents the EVAL framework; Section 4 describes high-dimensional dynamic adaptation and its implementation; Sections 5–6 evaluate it; and Section 7 discusses related work.

*This work was supported by the National Science Foundation under grant CPA-0702501 and by SRC GRC under grant 2007-HJ-1592. Smruti Sarangi is now with IBM India Software Laboratories, in Bangalore (srsarangi@in.ibm.com). Abhishek Tiwari is now with Goldman Sachs, in New York City (abhishek.tiwari@gs.com).

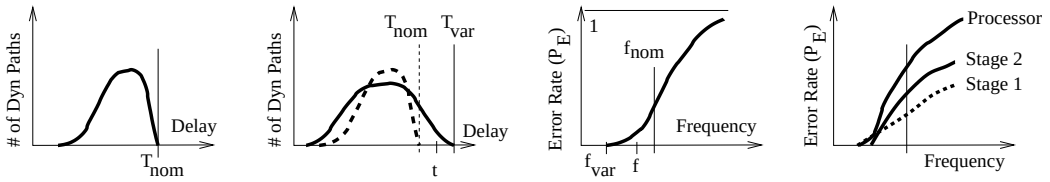


Figure 1: Impact of variation on processor frequency.

2 Background

2.1 Modeling Process Variation

While process variation exists at several levels, we focus on Within-Die (WID) variation, which is caused by both *systematic* effects due to lithographic irregularities and *random* effects, primarily due to varying dopant concentrations [30]. Two important process parameters affected by variation are the threshold voltage (V_t) and the effective channel length (L_{eff}). Variation of these two parameters directly affects a gate's delay (T_g) [25] and a gate's static power from subthreshold leakage (P_{sta}). These two measures plus a gate's dynamic power (P_{dyn}) are given by:

$$T_g \propto \frac{V_{dd} L_{eff}}{\mu (V_{dd} - V_t)^{\alpha}} \quad (1)$$

$$P_{sta} \propto V_{dd} T^2 e^{-qV_t/kT} \quad (2)$$

$$P_{dyn} \propto C V_{dd}^2 f \quad (3)$$

where V_{dd}, T, C, and f are the supply voltage, temperature, capacitance, and frequency, respectively, while μ and α are process parameters and q and k are physical constants.

To model process variation, we use the model in [26]. In this model, the systematic component of V_t's variation is modeled with two parameters: σ_{sys} and ϕ. A chip is divided into a grid. Each grid cell takes on a single value of V_t's systematic component as given by a multivariate normal distribution with parameters μ=0 and σ_{sys}. Along with this, the systematic component of V_t is spatially correlated using a function that only depends on the distance between two points — not on their position in the chip or on the direction of the line that connects them. Such a function decreases to zero for a distance ϕ called *range*. Intuitively, this means that at distance ϕ, there is no correlation between the systematic components of V_t for two transistors. L_{eff} is modeled similarly with a different σ_{sys} but the same ϕ. Overall, with this model, we can generate per-chip personalized maps of the systematic components of V_t and L_{eff}.

Random variation occurs at the much smaller granularity of individual transistors. Random variation is added analytically, as random values from a normal distribution with μ=0 and σ_{ran}.

By combining the systematic and random components of variation, we get the total variation. From here, using Equations 2 and 1, we compute the variation in the static power and delay of gates. Then, integration of the static power over the whole processor provides an estimate of the processor's static power. To estimate the processor's frequency, we take the variation in gate delay and, ideally, would apply it to the design files of a state-of-the-art processor. Since we do not have such files, we apply the gate delay variation to the models of critical path distributions in pipeline stages with logic and with memory structures found in [26]. From the resulting slowest critical paths, we estimate the processor frequency.

2.2 Modeling Variation-Induced Timing Errors

Process variation slows down some critical paths in a processor. As a result, the maximum frequency attainable by the processor decreases. If we do not operate the processor at the resulting low, safe frequency, the processor will suffer timing errors. To estimate the rate of these timing errors as a function of the processor frequency, we use the VATS model [26].

VATS considers the dynamic distribution of the delays of all the exercised paths in a pipeline stage. Without variation, such distribution may look like that in Figure 1(a). All paths take less than the nominal clock period (T_{nom}). Parameter variation changes gate delay (as per Equation 1), making some paths faster while others slower. The result is a more spread-out dynamic path delay distribution as in Figure 1(b). The processor now requires a longer clock period (T_{var}) to operate without timing errors.

If a processor is clocked with a period $t < T_{var}$ (Figure 1(b)), when the paths to the right of *t* are exercised, they may cause timing errors. An alternate way to see this is by plotting the error rate (or probability of error P_E) of the pipeline stage as we increase the pipeline frequency (Figure 1(c)). For $f > f_{var}$, where f_{var}=$1/T_{var}$, there are errors.

VATS generates a $P_E(f)$ curve like the one in Figure 1(c) for a given pipeline stage. VATS then models an n-stage pipeline as a series failure system, where each stage can fail independently. Each stage *i* has an activity factor ρ_i, which is the number of times that the average instruction exercises the stage. Finally, the processor error rate per instruction as a function of the frequency f is given by Equation 4, and is shown in Figure 1(d) for a 2-stage pipeline.

$$P_E(f) = \sum_{i=1}^{n} (\rho_i \times P_{E_i}(f)) \quad (4)$$

2.3 Fine-Grain ABB and ASV Application

Two techniques that modify the properties of gates are Adaptive Body Biasing (ABB) and Adaptive Supply Voltage (ASV). ABB [21, 35, 36] applies a body-bias voltage V_{bb} between the substrate and the source (or drain) of transistors to either decrease V_t (Forward BB or FBB), or increase V_t (Reverse BB or RBB). As per Equations 1 and 2, decreasing V_t reduces T_g but increases P_{sta}; increasing V_t causes the opposite behavior. ABB requires some extra fabrication steps.

ASV changes the V_{dd} applied to gates [5]. As per Equations 1, 2 and 3, increasing V_{dd} reduces T_g but increases P_{sta} and, especially, P_{dyn}; decreasing V_{dd} causes the opposite behavior. ASV is simpler to implement than ABB.

A chip can have multiple ABB or ASV domains — an environment referred to as fine-grained. Tschanz *et al.* [35] built a chip with 21 ABB domains. Narendra *et al.* [21] built a chip with a single ABB domain, although the chip includes 24 local bias generators with separate local bias networks — just as would be required in an implementation with domains. Both sets of authors estimate that the area overhead of their ABB support is ≈2%.

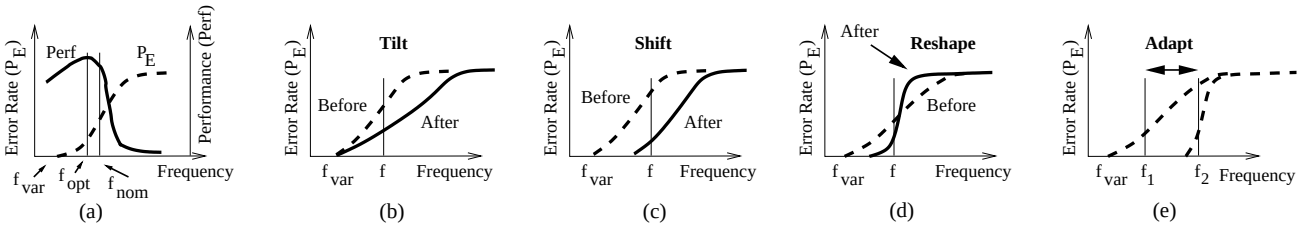


Figure 2: Tolerating (a) and mitigating (b)-(e) variation-induced errors in the EVAL framework.

If we assume that ASV supplies are chip-external, the area overhead of multiple ASV domains is small. Existing power supply pins are repurposed to deliver customized V_{dd} levels to each domain. Then, level converters may need to be added to cross voltage domains.

In this paper, we will initially assume that we can support over 10 ABB and ASV domains in a processor pipeline. While this is a stretch with current technology, current research (e.g., [15]) points to promising directions to make it feasible. We will then see that we do not need as much support.

3 The EVAL Framework

We assume that, in a high-variability environment, cost-effective processors will not be slowed down to operate at worst-case parameter values — their performance at T_{var} is low. Instead, we assume that they will be clocked with a period $t < T_{var}$ (Figure 1(b)) and suffer timing errors during normal execution.

To design cost-effective processors in this challenging environment, we propose the EVAL framework to tolerate and mitigate variation-induced errors. In EVAL, microarchitecture techniques can trade-off error rate for power and processor frequency.

3.1 Tolerating Errors

If we augment the processor with support for error detection and correction, it is possible to cycle the processor at $f > f_{var}$ while still ensuring correct execution. For instance, we can add a checker unit like Diva [40] to verify results from the main pipeline at instruction retirement. To ensure that the checker is error-free, it can be clocked at a safe, lower frequency than the main core, while the speed of its transistors is enhanced with ABB or ASV (Section 2.3) — according to [40], it is feasible to design a wide-issue checker thanks to its architectural simplicity. Alternately, we can add a checker processor like in Paceline [9], or augment the pipeline stages or functional units with error checking hardware like in a variety of schemes (e.g., [8, 37, 38]). With any of these architectures, the performance in instructions per second of the processor cycling at *f* is:

$$\begin{aligned} \text{Perf}(f) &= \frac{f}{CPI} = \frac{f}{CPI_{comp} + CPI_{mem} + CPI_{rec}} \\ &= \frac{f}{CPI_{comp} + mr \times mp(f) + P_E(f) \times rp} \end{aligned} \quad (5)$$

where, for the average instruction, CPI_{comp} are the computation cycles (including L1 misses that hit in the on-chip L2); CPI_{mem} are stall cycles due to L2 misses; and CPI_{rec} are cycles lost to recovery from timing errors. In addition, *mr* is the L2 miss rate in misses per instruction, *mp* is the observed L2 miss penalty in cycles non-overlapped with computation, P_E is the error rate per instruction, and *rp* is the error recovery penalty in cycles.

To a first approximation, CPI_{comp}, *mr*, and *rp* remain constant as we change *f*. If we use a Diva-like scheme, *rp* is equal to the branch misprediction penalty, since recovery involves taking the result from the checker, flushing the pipeline, and restarting it from the instruction that follows the faulty one. On the other hand, both *mp* and P_E increase with *f*.

For small *f*, P_E is small (Figure 1(c)), which makes $P_E \times rp$ small. Consequently, as *f* increases, *Perf* goes up because the numerator grows while the denominator increases only slowly — driven by the second and third terms. Eventually, as *f* keeps increasing, P_E reaches a point of fast growth, as shown in Figure 1(c). At this point, $P_E \times rp$ swells, and *Perf* levels off and quickly dips down. The result is shown in Figure 2(a), which shows the *Perf(f)* curve with a solid line and the $P_E(f)$ curve in dashes. We call f_{opt} the *f* at the peak *Perf*. With this approach, we reach frequencies higher than f_{var} by tolerating errors.

3.2 Mitigating Errors: Taxonomy of Techniques

We can reduce the number of variation-induced errors with microarchitectural techniques. We group such techniques into four classes, depending on how they affect the P_E vs *f* curve. In this section, we describe the four classes — shown in Figures 2(b)-(e) — while in Section 3.3, we present one example of each class.

Tilt: This class of techniques speeds-up many paths that are almost critical in a pipeline stage, but does not speed-up the slowest paths in the stage. As a result, the slope of the P_E vs *f* curve decreases, but the point where the curve meets the *f* axis (f_{var}) remains unchanged (Figure 2(b)). Overall, for an *f* moderately higher than f_{var}, P_E has decreased.

Shift: This class speeds-up all the paths in a stage by a similar amount. As a result, the curve largely shifts to the right (Figure 2(c)), therefore reducing P_E for a given f.

Reshape: There is typically an *energy cost* in tilting or shifting the curve as just described. Consequently, another class of techniques speeds-up the slow paths in the stage (thus consuming energy) and then saves energy by slowing down the fast paths. The first action shifts the bottom of the P_E vs *f* curve to the right and/or reduces the curve's initial slope; the second action shifts the top of the curve to the left and/or increases the curve's final slope (Figure 2(d)). For the *f* considered, the result is a lower P_E with potentially little energy cost. In reality, it may be easier to obtain this behavior at the processor level by speeding-up slow pipeline stages and slowing down the fast stages.

Adapt: As an application executes, it changes the types of operations it performs. As a result, its P_E vs *f* curve also changes, as shown in Figure 2(e). A final class of techniques adapts the *f* of the processor dynamically, to keep it as high as possible while maintaining P_E low at all times (Figure 2(e)).

3.3 Example of Error-Mitigation Techniques

We now list one example of each of the four classes of techniques to mitigate variation-induced errors.

3.3.1 Tilt: FU Replica without Critical-Path Wall

Design tools often design pipeline stages or Functional Units (FUs) with many near-critical paths. This is because, to save area and power, the non-critical paths are not subjected to high optimization — as long as they are shorter than the critical path, they are considered good enough. This creates a critical-path wall. The typical *Tilt* technique consists of taking a design and optimizing the near-critical paths to be shorter, so that the distribution of path delays and, therefore, the P_E vs *f* curve, become less steep. One possible way to do so is by increasing the width (*W*) of the transistors in the non-critical paths. This decreases their delay, which is proportional to K_1 + K_2/W. However, it also increases their power and area, since they are proportional to *W* [22].

Consequently, we propose to have two implementations of an FU side-by-side. Both contain the same logic circuit, but one is the original design (*Normal*) and the other has the transistors in the non-critical paths optimized (*LowSlope*). *LowSlope* is less power-efficient, but it has a lower-sloped P_E vs *f* curve. Consequently, if the pair of FUs falls on a chip area with fast transistors, since the FUs will not limit the processor frequency, we enable *Normal* and disable *LowSlope* — the processor will be more power efficient. If the pair falls on an area with slow transistors and limits the frequency of the processor, we enable *LowSlope* and disable *Normal* — the processor will cycle at a higher *f* for the same P_E.

We implement this technique in the most critical (typically, the hottest) FUs: we replicate the integer ALU unit and, inside the FP unit, replicate the adder and multiplier. Due to the area implications discussed in Section 5, we conservatively add one extra pipeline stage between the register file read and the execute stages.

3.3.2 Shift: Resizable SRAM Structures

A technique that has been proposed for SRAMs such as caches or queues is to dynamically disable sections of them to reduce power, access time or cycle time (e.g., [4]) — since smaller structures are faster. In these schemes, transmission gates separate sections of the structure; disabling transmission gates reduces the structure size. According to [4], SPICE simulations show that the impact of properly-designed transmission gates on the cycle time is negligible.

This is a possible *Shift* technique. On chips where the RAM structure falls on a fast region, the whole structure is kept (*Large* design). On chips where it falls on a slow region and limits the chip's *f*, we disable a fraction of the structure (*Small* design). With shorter buses to charge, most of the paths in the structure speed up, shifting the P_E curve to the right. Consequently, at any *f*, *Small*'s P_E is lower than *Large*'s. A shortcoming is that downsizing may decrease IPC. However, we now have room to trade more P_E for higher *f* and still come out ahead in performance.

We implement this technique in the integer and FP issue queues, since they are often critical. We enable them to operate at either full or *3/4* capacity.

3.3.3 Reshape: Fine-Grain ABB or ASV

ABB and ASV have been used to speed up slow sections of a chip and reduce the power of fast sections of the chip (e.g., [31, 36]). Using them in our framework reshapes the P_E curve as in Figure 2(d): speeding-up the slow pipeline stages pushes the lower part of the curve to the right and reduces its initial slope; slowing down and saving power on the fast pipeline stages pushes the upper part of the curve to the left and increases its final slope.

3.3.4 Dynamic Adaptation

Many algorithms have been proposed to dynamically change parameters such as voltage, frequency, or cache size to adapt to application demands (e.g., [6, 7, 10]). In the context of mitigating WID parameter variation, the key difference is that the problem has a very *high dimensionality*. The reason is that, to be effective, we need to sense from and actuate on many chip localities — as many as regions with different parameter values — and then optimize globally. Due to the complexity of the problem, we propose an implementation using machine learning algorithms; they enable rapid adaptation in multiple dimensions with minimal computation cost.

High-dimensional dynamic adaptation is the key technique in EVAL, and the one that agglutinates all the other techniques. We describe it next.

4 High-Dimensional Dynamic Adaptation for Variation Errors

We propose *High-Dimensional* dynamic adaptation as a novel technique to effectively mitigate WID parameter variation in upcoming processor chips. The goal is to boost the processor frequency when there is room in the tolerable P_E, power, and *T*. This technique involves (i) sensing from the several variation localities, (ii) relying on local techniques to *tilt*, *shift*, or *reshape* the P_E vs *f* curve in each locality, and (iii) finally optimizing globally. Given the complexity of the problem, we propose an implementation with a *software-based fuzzy* controller. Every time that a phase change is detected in the application, the processor is interrupted and runs the controller algorithm. The algorithm uses software data structures that contain fuzzy rules built by the manufacturer in a learning phase. In this section, we describe the problem statement, the algorithm, and its implementation. As an example, we adapt using all the example techniques described in Section 3.3, namely different ABB and ASV for each of n processor subsystems, FU replication, and issue-queue resizing.

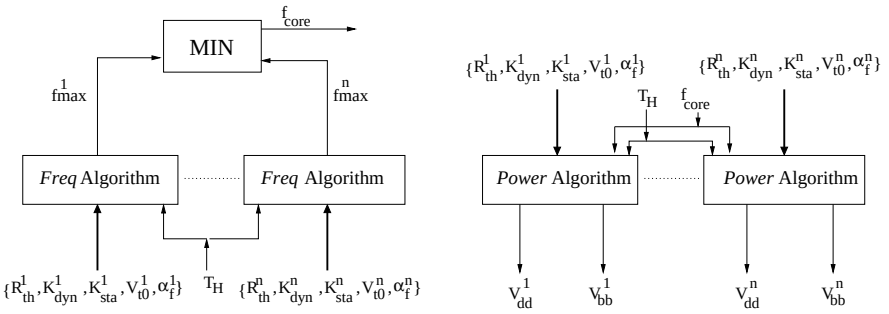
4.1 Optimization Problem Statement

Every optimization problem has a set of outputs subject to constraints, a final goal, and a set of inputs.

Outputs. There are $2n+3$ outputs: (i) the core frequency, (ii) the V_{dd} and V_{bb} for each of the n subsystems in the processor, (iii) the size of the issue queue (full or 3/4), and (iv) which FU to use (normal or low sloped). The last two outputs apply to integer or FP units depending on the type of application running.

Constraints. There are three: (i) no point can be at *T* higher than T_{MAX}, (ii) the processor power cannot be higher than P_{MAX}, and (iii) the total processor P_E cannot be higher than $P_{E_{MAX}}$. The reason for the latter is justified next.

Goal. Our goal is to find the processor *f* that maximizes performance. However, taking Equation 5 and finding the point where its derivative is zero is expensive. Instead, we can find a very similar *f* with little effort if our goal is to maximize *f* subject to the processor's P_E being no higher than $P_{E_{MAX}}$ — assuming we choose an appropriate $P_{E_{MAX}}$. Specifically, the $P_E(f)$ curve in Figure 2(a) is so steep that the range of *f* between $P_E = 10^{-4}$ and $P_E = 10^{-1}$ errors/instruction is minuscule (only 2–3%). Moreover, for typical values of CPI_{comp}, CPI_{mem}, and *rp* in Equation 5, $P_E = 10^{-4}$ makes CPI_{rec} negligible, while $P_E = 10^{-1}$ makes CPI_{rec} so high that *Perf* has already dropped. Conse-



(a): Freq algorithm (b): Power algorithm

Figure 3: Overview of the optimization algorithm.

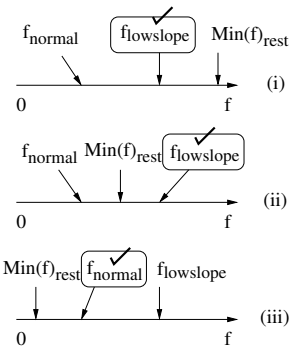


Figure 4: Possible frequency outcomes for the replicated FU.

quently, if we set our goal to maximize *f* subject to P_E being no higher than $P_E = 10^{-4}$ errors/instruction, we will obtain an *f* and a *Perf* that are both very close to the optimal ones.

Inputs. We need inputs that enable us to compute the *T*, P_{sta}, and P_{dyn} of each subsystem. Such values are given by:

$$T = T_H + R_{th} \times (P_{dyn} + P_{sta}) \quad (6)$$

$$P_{dyn} = K_{dyn}\alpha_f C V_{dd}^2 f \quad (7)$$

$$P_{sta} = K_{sta} V_{dd} T^2 e^{-qV_t/kT} \quad (8)$$

$$V_t = V_{t0} + k_1(T - T_0) + k_2 V_{dd} + k_3 V_{bb} \quad (9)$$

Equation 6 gives the steady-state *T* as function of the temperature of the common heat sink (T_H) and the thermal resistance of the subsystem (R_{th}). In Equation 7, K_{dyn} is a constant for the subsystem and α_f is the activity factor of the subsystem in accesses per cycle. In Equation 8, K_{sta} is a constant for the subsystem. Finally, in Equation 9, to compute V_t at *T*, we need to know its value V_{t0} at a reference temperature T_0. The effect of *T*, V_{dd} and V_{bb} on V_t is captured by constants k_1, k_2, and k_3 [19].

These equations form a feedback system and need to be solved iteratively. In these equations, the inputs to our control algorithm are T_H, R_{th}, K_{dyn}, α_f, K_{sta}, V_{t0}, and *f* — the rest are either outputs (V_{dd} and V_{bb}), constants, or intermediate parameters. Among these inputs, R_{th}, K_{dyn}, K_{sta}, and V_{t0} are per-subsystem constants that the manufacturer can measure and store on chip. This is done as follows. First, R_{th}, K_{dyn}, and K_{sta} are unaffected by variation, and are either a function of the subsystem area (R_{th}) or are estimated by the CAD tools based on the number and type of devices in the subsystem (K_{dyn} and K_{sta}). Second, V_{t0} is variation-dependent, and is measured on a tester at a known *T* by suspending the clocks and individually powering on each of the subsystems. The current flowing in is the leakage of that subsystem, from which V_{t0} can be computed according to Equation 8.

On the other hand, T_H and the per-subsystem α_f must be sensed dynamically. T_H can be measured with a single *T* sensor on the heat sink. Since the thermal time constant of the heat sink is of the order of tens of seconds [29], it only needs to be measured every few seconds. Finally, within a program phase, α_f in a given subsystem does not change much [28]. We can measure its average value at the beginning of every phase with performance counters similar to those already available. Adding up all inputs, we get $5n + 2$ inputs, of which only $n + 1$ need to be sensed (the per-subsystem α_f and the T_H).

4.2 Optimization Algorithm

To make the problem tractable, we propose to solve it by selecting a good solution in each of the n subsystems independently, and then modifying the solutions slightly to make them compatible. We proceed in two steps, namely the *Freq* and *Power* algorithms.

In the *Freq* algorithm, each subsystem *i* independently finds the maximum frequency f^i_{max} at which it can cycle using any value of ABB or ASV, and without violating the temperature constraint (T_{MAX}) or error rate constraint (which we conservatively set to $P_{E_{MAX}}/n$). Then, we take the minimum of all f^i_{max}, which becomes the core frequency f_{core}. Next, in the *Power* algorithm, each subsystem *i* takes f_{core} and independently recomputes its V^i_{dd} and V^i_{bb} that minimize the power consumed by the subsystem without violating T_{MAX} or $P_{E_{MAX}}/n$. Finally, a check is made that the overall processor power is lower than P_{MAX}.

An overview of the overall process is shown in Figure 3. In the two algorithms, each subsystem *i* takes the 6 inputs described in Section 4.1, which we represent with superscript *i*.

FU Replication. This technique is representative of those that provide the choice of one of two configurations in a subsystem. In this case, we need to run the *Freq* algorithm for each of the two configurations, and generate two f^i_{max}, namely f_{normal} for the normal FU and $f_{lowslope}$ for the low-sloped one, where $f_{normal} < f_{lowslope}$. To decide which of the two FUs to enable, we compare f_{normal} and $f_{lowslope}$ to the minimum value of f^i_{max} for all the other subsystems in the processor, which we call $Min(f)_{rest}$. The possible relative values of these three frequencies create three cases, as shown in Figure 4. If $f_{normal} < Min(f)_{rest}$ like cases (i) and (ii), the FU is critical and, therefore, we enable the low-sloped implementation to maximize frequency. Otherwise, like case (iii), we enable the normal one to save power.

Issue Queue Resizing. This technique is representative of those that provide two configurations which, at the same *f*, induce a different processor CPI. This is what occurs with the two different queue sizes: at the same *f*, they result in different processor CPIs. In this case, at the beginning of each phase, we need to take several μs to estimate with counters the CPI_{comp} with either queue size, namely $\text{CPI}_{comp_{1.00}}$ and $\text{CPI}_{comp_{0.75}}$. We then run the *Freq* algorithm for each queue size and, together with the f^i_{max} of all the other subsystems, compute the frequency we would select for the core, namely $f_{core_{1.00}}$ and $f_{core_{0.75}}$. Finally, we compare the estimated performance given in Equation 5 with either $\text{CPI}_{comp_{1.00}}$ and $f_{core_{1.00}}$, or with $\text{CPI}_{comp_{0.75}}$ and $f_{core_{0.75}}$. Finally, we enable the queue size that delivers the higher performance of the two.

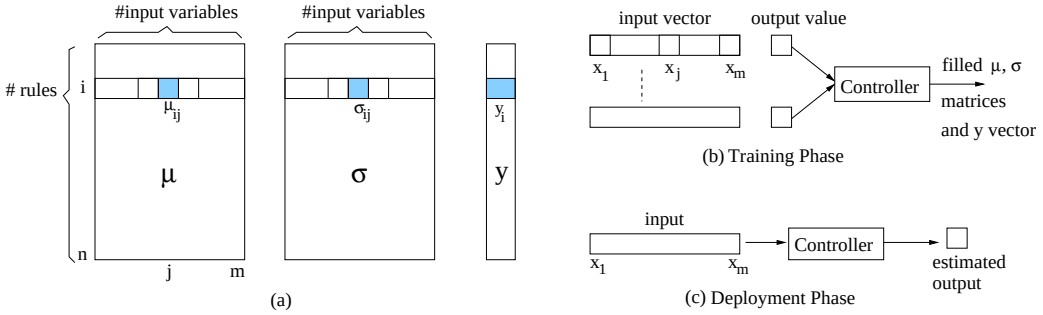


Figure 5: Operation of a Fuzzy Controller (FC).

4.3 Implementation

4.3.1 *Freq* and *Power* Algorithms

The *Freq* and *Power* algorithms are non-linear, multidimensional problems that have no analytic or simple algorithmic solution. In this section, we outline an exhaustive solution and then our proposed solution based on a fuzzy controller.

Exhaustive Algorithm. In this algorithm, we start from a finite set of values for each of *f*, V_{dd}, and V_{bb} in a subsystem, and exhaustively try all possible combinations to select the best one for the subsystem subject to constraints. Specifically, in the *Freq* algorithm for subsystem *i*, we compute, for each *f*, V_{dd}, and V_{bb} value combination, the resulting subsystem *T* and P_E. We select as f^i_{max} the maximum *f* that does not violate the T_{MAX} or $P_{E_{MAX}}/n$ constraints. In the *Power* algorithm, we take f_{core} and compute, for each V_{dd} and V_{bb} value combination, the resulting subsystem *T*, P_E, and power. We select the V_{dd} and V_{bb} combination that, while not violating the T_{MAX} or $P_{E_{MAX}}/n$ constraints, consumes the lowest power. Unfortunately, *Exhaustive* is too expensive to execute on-the-fly.

Fuzzy Controller Algorithm. Fuzzy Controllers (FC) are used when we know that there is a relationship between a multidimensional input and an output but we do not know the form of the function [39] — and therefore non-linear regression cannot be used. An FC learns a set of rules during a training phase, stores them, and uses them later-on in deployment to provide accurate answers to queries. The advantages of using an FC for each subsystem's *Freq* and *Power* algorithms (each box in Figure 3) are FC's good accuracy and very low response time. In our scheme, we use a software FC implementation.

An FC operates on two matrices called μ and σ, and a column vector of the same number of rows (Figure 5(a)). Each row of the matrices is a fuzzy rule, and its output is an element of the column vector. The number of columns in each matrix is equal to the number of variables per input vector. In the training phase (Figure 5(b)), we train the FC with thousands of training examples to fill in the matrices and vector (Appendix A). A training example is an input vector along with its correct output. We generate each training example by running *Exhaustive* offline. In the deployment phase (Figure 5(c)), the FC takes an input vector and produces an estimated output (Appendix A).

In the *Freq* algorithm, the 6 inputs shown in Figure 3 are fed to a per-subsystem FC whose output is f^i_{max}; in the *Power* algorithm, there are two FCs per subsystem — one for output V^i_{dd} and one for output V^i_{bb}. Their inputs are shown in Figure 3.

We do not expect that training the FCs will increase the chip test time excessively. First, of all the constants needed in Section 4.1, only the per-subsystem V_{t0} must be measured by the manufacturer on the chip hardware. Moreover, we discussed in Section 4.1 that V_{t0} can be measured quickly. The second step involves populating the FCs. This is done by running many inputs on a software model of the chip that implements *Exhaustive*.

4.3.2 Controller System Interface

The controller system consists of a set of privileged-mode software routines and data structures that implement FCs for each subsystem. The FCs access a sensor attached to the heat sink that provides T_H, and a set of performance counters that provide α^i_f for each subsystem. In addition, there are per-subsystem thermal sensors [29] to detect overheating, a core-wide power sensor to detect power overruns, and a core-wide P_E counter from the checker hardware (Section 3.1) to detect error-rate overruns. When a hardware-based application phase predictor like the one by Sherwood *et al.* [28] detects a new phase, it interrupts the processor, which then runs the controller routines.

4.3.3 Timeline

Figure 6 shows the timeline of the whole system. The phase detector detects a new phase on average every $\approx$ 120ms, and interrupts the processor. If this phase has been seen before, a saved configuration is reused; otherwise, the controller attempts to find a good configuration. For that, it first lets the application run for $\approx 20\mu$s, while counters estimate each subsystem's new activity factor α_f. During this period, counters also estimate CPI_{comp} with the full-sized issue queue (for the first half of the period) and the 0.75 configuration (for the second half)[1]. As per Section 4.2, these CPIs will be used to resize the integer or FP issue queue — depending on the type of application running.

Subsequently, the fuzzy controller routines take over the CPU and execute. Based on the number of instructions executed in these routines, we estimate that a 4GHz processor takes about 6μs to run them. This is the only time when the application is not running. After the new configuration is chosen, the application runs again and the system transitions to the selected f_{core} and per-subsystem V_{dd} and V_{bb}. While the transition latency depends on the magnitude of the change required, we estimate it is at most 10μs — a figure slightly more aggressive than the one for Intel's XScale technology.

Due to inaccurate estimation when using fuzzy control, the final configuration may not be optimal or may not meet constraints. If it is too aggressive, a sensor may log a constraint violation — a thermal/power violation within a thermal time constant ($\approx$ 2ms) or an error constraint violation sooner. In this case, the system performs a minor readjustment, which involves decreasing *f* ex-

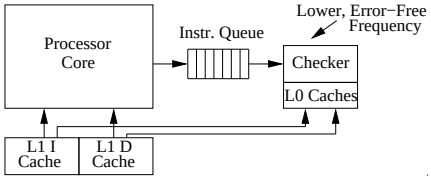
[1] If these two tests end up testing very different code sections, we may make a suboptimal decision. Correctness is never in question.

4−core CMP; Tech: 45nm; Vdd: 1V; f (no variation): 4GHz
Round trip latency in cycles from processor to:
L1: 2; L2: 8; Memory: 208
Parameter changes
f: From 2.4GHz to over 4GHz in 100MHz steps
ABB: From −500mV to 500mV in 50mV steps
ASV: From 800mV to 1200mV in 50mV steps
Area measured from die photo:
IntALU subsystem (3 add/shift + 1 mult): 0.55% proc area
1 FPadd + 1 FPmult: 1.90% proc area
Full−sized issue queues:
Integer: 68 entries
FP: 32 entries
Fuzzy controller system:
Each FC: 25 rules; 10,000 training examples
Phase detector: 32 buckets; 6 bits/bucket
Process parameters:
Vt: μ: 150mV at 100C; σ/μ : 0.09; ϕ : 0.5
Leff: σ/μ: 0.5 x Vt's σ/μ; ϕ:0.5
Number of chips per experiment: 100
P_{MAX}=30W/proc, T_{MAX}=85C, T_{H_MAX}=70C, P_{E_MAX}=10^{-4} err/inst

(a): Some parameter values

Subsystem	Type
DTLB	memory
Dcache	memory
FPUnit	logic
FPQ	mixed
FPReg	memory
FPMap	memory
IntALU	logic
LdStQ	mixed
IntReg	memory
IntQ	mixed
IntMap	memory
ITLB	memory
Icache	memory
BranchPred	mixed
Decode	logic

(b): Subsystems used



(c): Logical organization of the checker

Source	Area (% Proc)
ASV	0.0
Checker	7.0
IntALU Repl	0.7
FPAdd/Mul Repl	2.5
I−Queue Resize	0.0
Phase Detector	0.3
Sensors	0.1
Total	10.6

(d): Additional area

Figure 7: Characteristics of the system modeled.

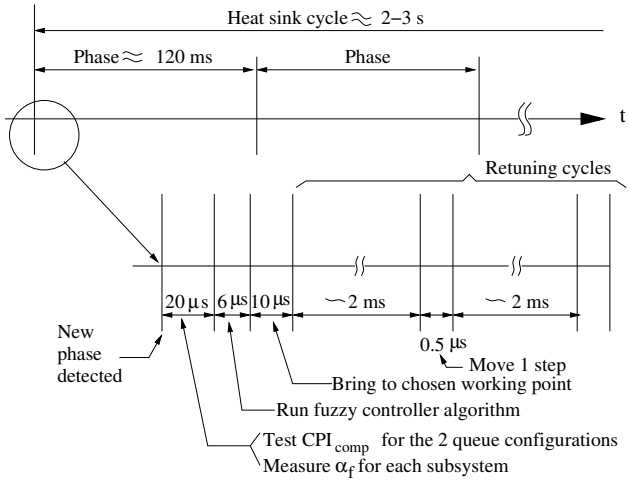


Figure 6: Timeline of the adaptation algorithm.

ponentially — first by 1 100MHz step, then by 2 steps, 4, and 8 without running the controller — until the configuration causes no violation, and then gradually ramping up *f* in constant 100MHz steps up until right below a *f* that causes violations.

These small *f* changes to prevent violations are called *Retuning Cycles* (Figure 6). They do *not* involve re-running the fuzzy controller. Finally, every 2–3s, the T_H sensor is refreshed.

4.3.4 Summary of Complexity

We believe that the complexity of this technique is modest. The key part of the technique, namely the controller, is implemented in software, which reduces design complexity. The hardware aspects include sensors, FU replication, issue-queue resizing, Diva checker, and fine-grain ABB/ASV. Note that this is not an "all-or-nothing" system — different subsets of techniques can be used. Finally, fuzzy control has been shown to be a simple, effective way to handle complicated control systems [39].

5 Evaluation Environment

We model a Chip Multiprocessor (CMP) at 45nm technology with four 3-issue cores similar to the AMD Athlon 64. Each core has two 64KB L1 caches, a private 1MB L2, and hyper-transport links to the other cores. We estimate a nominal (i.e., without variation) frequency of 4GHz with a supply voltage of 1V. Figure 7(a) shows some characteristics of the architecture. In this evaluation, we choose to have 15 subsystems per core, as shown in Figure 7(b).

The frequency changes like in Intel XScale, and all changes can be effected in 10μs. V_{bb} and V_{dd} can be adjusted on a per-subsystem basis, as in [36]. Figure 7(a) shows the ranges and step sizes of the changes. Based on [21, 35], we estimate that the area overhead of ABB is $\approx$ 2%. However, we will not include ABB in our preferred configuration. As indicated in Section 2.3, we optimistically assume that the area overhead of ASV is largely negligible [5], given chip-external ASV supplies. We will include ASV in our preferred configuration.

Each core has a checker like Diva [40] to detect and tolerate core errors (Figure 7(c)). The checker is sped up with ASV so that it runs at 3.5GHz without any errors. It has a 4KB L0 D-cache, a 512B L0 I-cache, and a 32-entry queue to buffer instructions retiring in the processor. To ensure that the checker reads the L1 reliably, the L1 is augmented with the SRAM Razor scheme [14], which adds duplicate sense amplifiers to L1. L1 reads are performed with speculative timing for the core, and then a fraction of a cycle later with safe timing for the checker. Since checker design is not our contribution, we do not elaborate further, although we include its area and power cost in our overall computations.

FU Replication Technique. We replicate the integer ALU unit and, inside the FP unit, replicate the set composed of adder and multiplier. To estimate the area, power, and timing of low-sloped FUs, we use the data in [1]. Although their circuit is a small sequential circuit rather than an ALU, we consider that their measurements are an acceptable estimation of this optimization's impact. On average, the low-sloped unit consumes 30% more area and power, and its path delay distribution changes such that the mean decreases by 25% and the variance doubles [1]. From this, and the FU area shown in Figure 7(a), it follows that integer and FP FU replication adds 0.7% and 2.5% of processor area (Figure 7(d)).

Since replication lengthens the wires that connect the register file to the FUs, we conservatively add one additional pipeline stage between the register file read and the execute stages. While

Environment	Explanation	Environment	Explanation
1: *Baseline*	Plain processor with variation effects	5: *TS+ASV+Q*	TS + ASV + issue-queue resizing (Sec 3.3.2)
2: *TS*	Baseline + Diva checker for timing speculation	6: *TS+ASV+Q+FU*	TS + ASV + Q + FU replication (Sec 3.3.1)
3: *TS+ASV*	TS + adaptive supply voltage (Sec 3.3.3)	7: *ALL*	TS + ASV + Q + FU + ABB
4: *TS+ASV+ABB*	TS + techniques of Sec 3.3.3	8: *NoVar*	Plain processor with no variation effects

Table 1: Key environments considered.

this increases the load-misspeculation and branch-misprediction loops by one cycle, it does not affect the execution of back-to-back ALU instructions. As a result, the overall performance impact is modest and the scheduler complexity is largely unaffected.

SRAM Resizing Technique. We resize the integer and FP issue queues (Figure 7(a)), so that each can operate at either full or *3/4* capacity [4]. This technique adds no extra area (Figure 7(d)).

Dynamic Adaptation Technique. The fuzzy controller routines have ≈120 Kbytes of data footprint. For each Fuzzy Controller (FC), we choose to have 25 rules and train them with 10,000 randomly-selected examples generated with *Exhaustive* — these settings give good results. This data is stored in a reserved memory area. When the controller runs, it pollutes the data cache, but since it runs only once per phase, its performance impact is negligible. The code footprint is tiny because the code is only a few tight loops.

The hardware-based application phase detector uses basic-block execution frequencies to detect phases. It is the one described by Sherwood *et al.* [28]. Its parameters are shown in Figure 7(a). Using CACTI [32], we estimate that it adds ≈0.3% of processor area (Figure 7(d)). The detector is designed to detect *T* and power phases as in [13]. It uses similar stability criteria as [13], and obtains similar phases. The average length of a stable phase in SPEC applications is ≈120ms, so adapting at every phase boundary has minimal overhead. Such stable phases account for 90-95% of the execution time.

Finally, there is a set of sensors as described in Section 4.3.2. We estimate their area overhead to be ≈0.1% (Figure 7(d)). Consequently, the total area overhead of our EVAL system is 10.6%.

Process Variation. We model V_t and L_{eff} variation using the model in [26]. Using the parameters recommended in the model, we set V_t's σ/μ to 0.09 and use equal contributions of the systematic and random components. Consequently, $\sigma_{sys}/\mu = \sigma_{ran}/\mu = \sqrt{\sigma^2/2}/\mu = 0.064$. Moreover, we set ϕ to 0.5, and L_{eff}'s σ/μ to 0.5 of V_t's σ/μ. Consequently, for L_{eff}, we use $\sigma/\mu = 0.045$ and $\sigma_{sys}/\mu = \sigma_{ran}/\mu = 0.032$ (Figure 7(a)). Each individual experiment is *repeated 100 times*, using 100 chips that have different systematic V_t and L_{eff} maps generated with the same σ and ϕ. We find that using more than these 100 samples changes our results insignificantly.

5.1 Performance, Power, and Temperature

We use the SESC [23] cycle-level execution-driven simulator to model the performance of the chip running the SPECint and SPECfp 2000 codes. Each application is run on each of the 4 cores of each of 100 chips. The resulting average performance is combined with that of the other applications.

The simulator is augmented with dynamic power (P_{dyn}) models from Wattch [3] and CACTI [32] to estimate P_{dyn} at a reference technology and frequency. In addition, we use HotLeakage [41] to estimate static power (P_{sta}) at the same reference technology. Then, we obtain ITRS scaling projections for the per-transistor dynamic power-delay product, and for the per-transistor static power [12]. With these two factors, we can estimate P_{dyn} and P_{sta} for the scaled technology and frequency relative to the reference values. For each of the four cores in the CMP, we set P_{MAX} for the core to 30W.

We use HotSpot [29] to estimate R_{th} for each of the subsystems. Once we have all the parameters in Equations 6–9, we solve the equations for any (V_{dd}, V_{bb}, *f*) operating point. Solution is by iterating until convergence. We set T_{MAX}=85°C, $T_{H_{MAX}}$=70°C, and as per Section 4.1, $P_{E_{MAX}} = 10^{-4}$ err/inst.

With this setup, we model the environments of Table 1, without and with the high-dimensional dynamic adaptation technique.

6 Evaluation

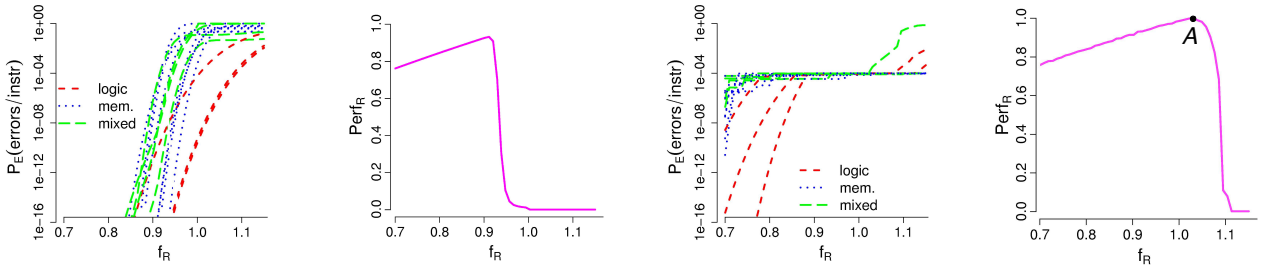
We first show that P_E, power, and *f* (or performance) are tradeable quantities. Next, we evaluate the relative performance and power consumption of the environments in Table 1, without and with dynamic adaptation. Finally, we characterize the dynamic adaptation technique in detail.

6.1 Error Rate, Power, and Frequency (or Performance) Are Tradeable

In our processor, P_E, power, and *f* (or performance) are tradeable quantities. To show it, we experiment with one application (*swim*) running on one sample chip. For the *TS* environment, Figure 8(a) shows the P_E vs frequency curves of all the processor's subsystems. The P_E curves are labeled based on the type of subsystem (*logic*, *memory*, or *mixed*), while the frequency (f_R) is shown relative to the *NoVar* environment (Table 1). In the figure, the slope of the P_E curves depends primarily on the subsystem type. Memory subsystems, with their homogeneous paths, have a rapid error onset. Logic subsystems have a wide variety of paths and, therefore, produce a more gradual error onset. Mixed subsystems fall between the two extremes.

Figure 8(b) shows the processor performance relative to *NoVar* (*Perf*$_R$) as a function of frequency. As indicated in Section 3.1, as frequency increases, performance improves as long as P_E (Figure 8(a)) is not prohibitively high. As P_E increases past a critical point, *Perf*$_R$ falls off sharply. From Figure 8(b), we see that, with *TS*, the processor runs optimally at $f_R \approx 0.91$ (that is, slower than *NoVar*) and delivers a *Perf*$_R \approx 0.92$ (again, lower than *NoVar*). This is much better than under *Baseline*, which cannot tolerate any error. Indeed, *Baseline* can only run at the f_R where the leftmost P_E curve in Figure 8(a) intersects the x-axis ($f_R \approx 0.84$).

To improve *Perf*$_R$, we need to delay the exponential P_E onset. To this end, Figures 8(c) and (d) repeat the P_E and *Perf*$_R$ curves for an environment that has per-subsystem ASV and ABB — set by the *Exhaustive* algorithm of Section 4.3.1. For each f_R, this algorithm finds a configuration where the sum of all P_E curves is $\approx 10^{-4}$ errs/inst, which is $P_{E_{MAX}}$. This is accomplished by speeding up slow subsystems and saving power on fast ones. This is why Figure 8(c) shows a convergence of lines at $P_E \approx 10^{-4}$. However, for high f_R, it becomes increasingly power-costly to keep $P_E \leq P_{E_{MAX}}$. Eventually, the power constraint is reached, and no further ASV/ABB can be applied to speed up subsystems and keep $P_E \leq P_{E_{MAX}}$. Then, as can be seen in Figure 8(c), some P_E curves escape up and, as shown in Figure 8(d), *Perf*$_R$



(a) Subsystem error rates under *TS*

(b) Processor performance under *TS*

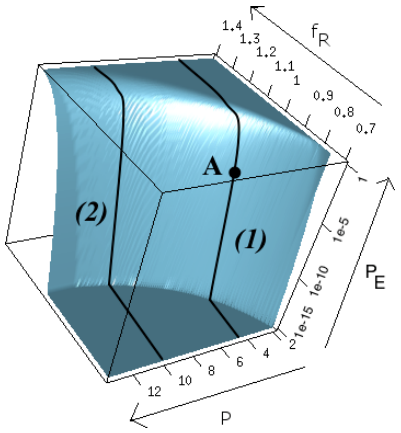
(c) Subsystem error rates under *TS+ASV+ABB*

(d) Performance of processor under *TS+ASV+ABB*

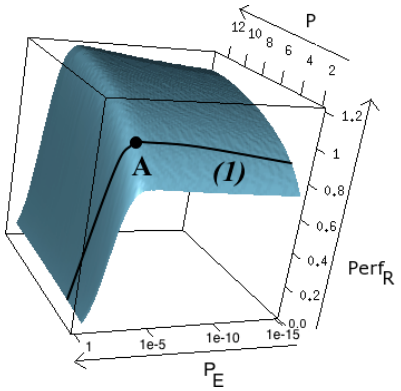
Figure 8: Subsystem error rates *vs* frequency, and processor performance *vs* frequency under *TS* (a and b) and under *TS+ASV+ABB* (c and d).

plunges. However, by keeping P_E under control, as we went from Figure 8(b) to Figure 8(d), we moved the peak of the $Perf_R$ curve to the right and up (Point *A*); the optimal f_R is ≈ 1.03 and the optimal $Perf_R$ is ≈ 1.00 — as high as *NoVar*'s.

We gain further insight by focusing on one subsystem and extending its P_E vs f_R graph with power (*P*) as a third axis. Figure 9(a) shows this for the integer ALU in the presence of per-subsystem ASV/ABB. The surface in the figure is constructed using the *Exhaustive* algorithm of Section 4.3.1 to find the minimum realizable P_E for each *P* and f_R. Point *A* in the figure shows the conditions at the optimal f_R in Figures 8(c) and (d).



(a)



(b)

Figure 9: Three-dimensional views of the power *vs* error rate *vs* frequency surface (a) and of the power *vs* error rate *vs* performance surface (b) for the ALU in Figure 8(c).

In Figure 9(a), if we draw Line *(1)* at constant *P* through *A*, we obtain the familiar P_E vs f_R curve: As f_R grows, P_E first remains at zero and then suddenly increases very steeply to reach one. Moreover, if we are willing to spend more *P* in this subsystem via ASV/ABB (Line *(2)*), the subsystem supports a higher f_R before P_E reaches a given level. We see, therefore, that power and error rate are tradeable quantities. If the goal is to increase f_R, we can either pay with a higher *P* consumption for a given P_E, or with a higher P_E for a constant *P*.

Figure 9(b) replaces f_R with the $Perf_R$ of the processor and shows the same Line *(1)* and point *A*. As P_E increases along Line *(1)*, we obtain a familiar curve: $Perf_R$ first increases slowly and then drops abruptly. At the maximum point, $Perf_R$ can further increase only at the cost of consuming more *P*.

6.2 Frequency, Performance and Power

Figure 10 shows the processor frequency for each of the environments in Table 1 normalized to *NoVar*. The two horizontal lines are the frequencies of *Baseline* and *NoVar*. Due to process variation, *Baseline* only reaches 78% of the frequency of an idealized no-variation processor (*NoVar*). To counter this variation, we add error tolerance and mitigation techniques one at a time. For each set of techniques, we consider three cases: no dynamic adaptation (*Static*), dynamic adaptation using our proposed fuzzy controller (*Fuzzy-Dyn*), and dynamic adaptation using our *Exhaustive* search algorithm (*Exh-Dyn*).

The leftmost set of three bars shows the frequency impact of Timing Speculation (*TS*), which is a prerequisite for all subsequent schemes. *TS* increases frequency by ≈12%. Since there are no techniques to reshape the subsystem error curves, dynamic adaptation does not offer much improvement over *Static*.

The next set of bars (*TS+ASV*) adds per-subsystem ASV. Without dynamic adaptation, the frequency reaches 97% of *NoVar*. However, because the maximum ASV level that a subsystem can tolerate before exceeding constraints is application-dependent, *Static* must be conservative. The dynamic environments are not so restricted and apply ASV more aggressively. As a result, they end up delivering a frequency 5–6% higher than *NoVar*.

Continuing to the right (*TS+ASV+ABB*), adding ABB to *TS+ASV* produces only modest gains under the dynamic schemes, as it provides some additional flexibility in reshaping P_E curves. Overall, however, it does not offer much improvement over *TS+ASV* for its added complexity. Therefore, we initially exclude ABB as we begin to add microarchitecture techniques.

The next two sets of bars show the effect of adding microarchitecture techniques for error mitigation: issue queue resizing

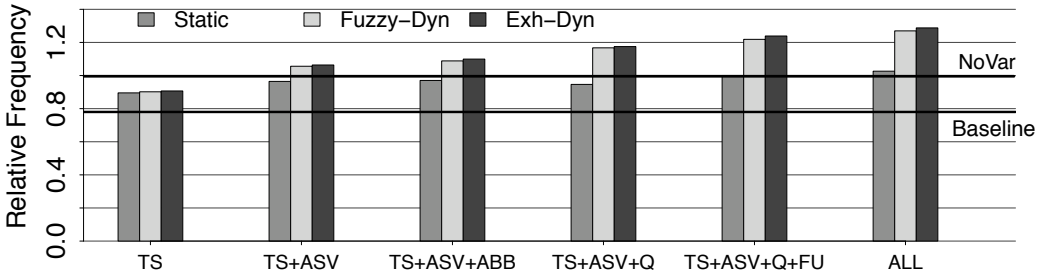


Figure 10: Processor frequency for each environment normalized to *NoVar*.

(*TS+ASV+Q*) and FU replication (*TS+ASV+Q+FU*). Although not shown in the figure, if we add them without any ABB or ASV capability, they deliver a disappointing 2% frequency increase. This is because temperatures stay low enough that optimizing the two subsystems does not deliver, on average, a substantial frequency boost. However, applying ASV to aggressively re-shape the P_E curves pushes the subsystem temperatures and powers higher. Under these conditions, the FUs and issue queues routinely form hotspots and become frequency-limiters. By speeding up these critical subsystems, the microarchitectural techniques deliver good frequency gains.

The *TS+ASV+Q* and *TS+ASV+Q+FU* environments show that, to take full advantage of the microarchitecture techniques, dynamic adaptation is required. Not all applications or phases exercise the issue queue or the FUs enough to make them critical. In these cases, statically enabling these techniques costs performance or power. Consequently, it makes sense to enable these optimizations dynamically. With both microarchitecture techniques and *Fuzzy-Dyn*, we reach a frequency that is 21% higher than *NoVar* (or 56% higher than *Baseline*).

The difference between the two rightmost environments *TS+ASV+Q+FU* and *ALL* is small. This suggests that it is reasonable to forgo the added complexity of ABB in an EVAL system. Similarly, the difference between using a fuzzy adaptation scheme (*Fuzzy-Dyn*) instead of exhaustive search (*Exh-Dyn*) is practically negligible in all environments.

Figure 11 shows the performance of all these environments normalized to *NoVar*. The figure is organized as Figure 10. We see that performance follows the same trends as frequency, except that the magnitude of the changes is smaller. The preferred scheme (*TS+ASV+Q+FU* with *Fuzzy-Dyn*) realizes a performance gain of 14% over *NoVar* or, equivalently, 40% over the *Baseline* processor.

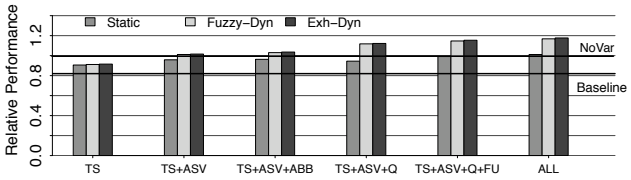


Figure 11: Performance of different environments.

Figure 12 shows the average power consumed, including both P_{dyn} and P_{sta}, in a core and its L1 and L2 caches. Recall that, for each processor, we set P_{MAX} to 30W. We see that the average power for *NoVar* is 25W — although some processors reach 30W at certain points of high activity and temperature — while *Baseline* consumes 17W. The latter runs at lower frequency. As we add mitigation techniques, the power tends to increase, although its actual value depends on many variables. With *TS+ASV+Q+FU* and *Fuzzy-Dyn*, the average processor power is almost exactly 30W. This shows that the *Fuzzy-Dyn* controller is making use of the available power to maximize performance without violating constraints. We also see that *Exh-Dyn* consumes about the same power as *Fuzzy-Dyn*.

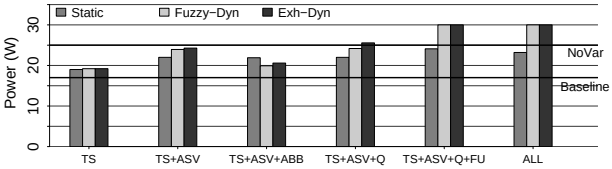


Figure 12: Power per processor (core+L1+L2) for different environments.

6.3 Characterizing Dynamic Adaptation

Finally, we compare the output of the fuzzy controller to *Exhaustive* to show why fuzzy control provides nearly optimal power and performance results. Table 2 shows the mean error in the frequency, V_{dd}, and V_{bb} values generated by the fuzzy controller compared to the output of *Exhaustive*. We show results for memory, mixed, and logic subsystems separately. The errors are shown in absolute units and as a percentage of the nominal value (except for V_{bb}, where the nominal value is zero). The table shows that, in practically all cases, the fuzzy controller predicts frequency, V_{dd} and, to a lesser extent, V_{bb}, quite accurately.

Param.	Environment	Fuzzy Controller - Exhaustive (% of Nom.)		
		Memory	Mixed	Logic
Freq. (MHz)	*TS*	168 (4.1%)	146 (3.6%)	170 (4.2%)
	TS+ABB	170 (4.2%)	135 (3.3%)	149 (3.6%)
	TS+ASV	450 (11.0%)	410 (10.0%)	160 (3.9%)
	TS+ABB+ASV	176 (4.3%)	162 (4.0%)	146 (3.6%)
V_{dd} (mV)	*TS+ASV*	17 (1.7%)	24 (2.4%)	14 (1.4%)
	TS+ABB+ASV	16 (1.6%)	22 (2.2%)	22 (2.2%)
V_{bb} (mV)	*TS+ABB*	72 (–)	69 (–)	76 (–)
	TS+ABB+ASV	115 (–)	129 (–)	124 (–)

Table 2: Difference between the selections of the fuzzy controller and *Exhaustive* in absolute and relative terms.

An important reason why the performance and power of *Fuzzy-Dyn* and *Exh-Dyn* are largely the same despite any inaccuracies shown in Table 2 is the presence of *Retuning Cycles* (Section 4.3.3). Specifically, when the fuzzy controller algorithm selects a configuration, any of the five outcomes shown in Figure 13 is possible. *NoChange* is the best case. Here, no constraint is violated and the first attempt at increasing f fails, signifying that the fuzzy controller's output was near optimal. In *LowFreq*, no constraint is violated, but retuning cycles are able to further increase f. The third case (*Error*) occurs when the configuration violates

$P_{E_{MAX}}$, and as a result, retuning cycles must reduce f. Similarly, the *Temp* and *Power* cases occur when T_{MAX} or P_{MAX} is exceeded, again causing a reduction in f.

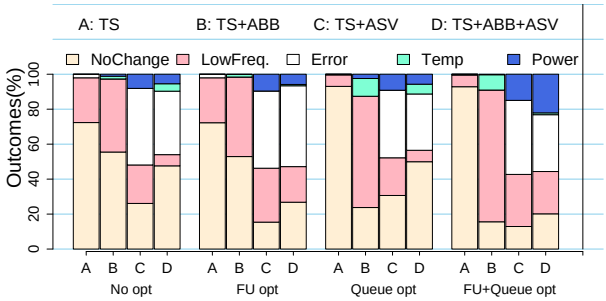


Figure 13: Outcomes of the fuzzy controller system.

Figure 13 shows the fraction of times when each outcome occurs in an environment with no microarchitecture technique (*No opt*), a single technique (*FU opt* and *Queue opt*), or both techniques (*FU+Queue opt*). We see that *NoChange* dominates for *TS* and, together with *LowFreq*, accounts for about 50% or more in all the bars. *Temp* cases are infrequent. *Fuzzy-Dyn* does so well thanks to the correcting retuning cycles in four of the five cases.

7 Related Work

Process Variation Modeling and Mitigation. There are several models of WID process variation that can be used for microarchitecture research (e.g., [11, 16, 18, 24, 26]). The EVAL framework can be used with any of them.

A related approach to mitigate process variation is dynamic retiming of pipelines [16, 33, 34]. The EVAL framework is a more powerful way to handle process (and parameter) variation for three reasons. First, EVAL is designed for a challenging environment with timing errors, where it trades-off error-rate for performance and power; Dynamic retiming always clocks the processor at a safe frequency. Second, EVAL controls the delay and power of each pipeline stage through fine-grain ASV/ABB; Dynamic retiming mostly redistributes slack among pipeline stages. Finally, EVAL manages multiple techniques (ABB/ASV, FU replication, and issue-queue resizing) to get a better working point. As a result, the performance gains from EVAL (40%) are larger than from dynamic retiming (10–20%).

Error Tolerance and Mitigation. There are many architectures for timing speculation (e.g., [8, 9, 17, 20, 27, 37, 38, 40]). Some of these works [9, 38] have been suggested for an environment with parameter variation. Our EVAL framework is more powerful than these architectures. Timing speculation is just one technique that EVAL uses; it manages multiple techniques. Therefore, EVAL can deliver a better power/performance point.

Dynamic Optimization. Many schemes for dynamic adaptation of parameters such as voltage, frequency, cache size, etc. have been proposed (e.g., [4, 6, 7, 10]). This includes the application of whole-chip ABB and DVFS [19], as well as fine-grain ABB in conjunction with whole-chip DVFS [31]. However, compared to EVAL, these proposals did not try to optimize globally so many variables concurrently.

8 Conclusions

This paper explored the environment where processors are not designed for worst-case parameter values and, therefore, need to tolerate variation-induced errors during normal operation. In this environment, the paper made two main contributions. First, it introduced a framework called EVAL that gives insight into how microarchitecture techniques can mitigate variation-induced errors and trade-off error rate for power and processor frequency. It showed how they can tilt, shift, and reshape the error rate *vs* frequency curve. Second, it presented *High-Dimensional* dynamic adaptation, an effective microarchitecture technique to maximize processor performance and minimize power in the presence of variation-induced timing errors. It also showed an efficient implementation of this technique based on a machine-learning algorithm.

Our results showed that, under variation-induced timing errors, high-dimensional dynamic adaptation is a feasible and effective technique. With no support for handling variation, a processor could only cycle at 78% of its no-variation frequency. However, by dynamically adapting processor frequency, per-subsystem ASV, and two modest-cost microarchitecture schemes (issue-queue resizing and FU replication), the processor increased its frequency by 56% on average — effectively cycling 21% faster than under no variation. Processor performance increased by 40% on average (or 14% over the no-variation scenario), always within error-rate, power, and temperature constraints. The area overhead of this technique was estimated to be only 10.6% of the processor area.

We believe that our high-dimensional dynamic adaptation scheme using fuzzy control techniques has wide applicability. It is applicable to situations requiring dynamic adaptation of a large number of inter-related variables. This includes many problems beyond WID variation or timing speculation.

References

[1] S. Augsburger and B. Nikolic. Combining dual-supply, dual-threshold and transistor sizing for power reduction. In *International Conference on Computer Design*, September 2002.

[2] K. Bowman, S. Duvall, and J. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid State Circuits*, 37(2):183–190, February 2002.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture*, June 2000.

[4] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. *Lecture Notes in Computer Science: Power Aware Computer Systems*, 2008:25–37, May 2001.

[5] T. Chen and S. Naffziger. Comparison of adaptive body bias (ABB) and adaptive supply voltage (ASV) for improving delay and leakage under the presence of process variation. *IEEE Transactions on VLSI Systems*, 11(5):888–899, October 2003.

[6] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *International Symposium on Computer Architecture*, May 2002.

[7] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[8] 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. In *International Symposium on Microarchitecture*, December 2003.

[9] B. Greskamp and J. Torrellas. Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2007.

[10] M. Huang, J. Renau, and J. Torrellas. Positional Adaptation of Processors: Application to Energy Reduction. In *International Symposium on Computer Architecture*, June 2003.

[11] E. Humenay, D. Tarjan, and K. Skadron. Impact of process variations on multicore performance symmetry. In *Conference on Design, Automation and Test in Europe*, April 2007.

[12] International Technology Roadmap for Semiconductors (ITRS). Process integration, devices, and structures. 2007.

[13] C. Isci, A. Buyuktosunoglu, and M. Martonosi. Long-term workload phases: Duration predictions and applications to DVFS. *IEEE Micro*, 25(5):39–51, September 2005.

[14] E. Karl, D. Sylvester, and D. Blaauw. Timing error correction techniques for voltage-scalable on-chip memories. In *International Symposium on Circuits and Systems*, May 2005.

[15] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *International Symposium on High-Performance Computer Architecture*, February 2008.

[16] X. Liang and D. Brooks. Mitigating the impact of process variations on CPU register file and execution units. In *International Symposium on Microarchitecture*, December 2006.

[17] T. Liu and S. Lu. Performance improvement with circuit-level speculation. In *International Symposium on Computer Architecture*, December 2000.

[18] D. Marculescu and E. Talpes. Variability and energy awareness: A microarchitecture-level perspective. In *Design Automation Conference*, June 2005.

[19] S. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *International Conference of Computer Aided Design*, November 2002.

[20] F. Mesa-Martinez and J. Renau. Effective optimistic-checker tandem core design through architectural pruning. In *International Symposium on Microarchitecture*, December 2007.

[21] S. Narendra et al. 1.1V 1GHz communications router with on-chip body bias in 150 nm CMOS. In *International Solid-State Circuits Conference*, February 2002.

[22] J. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 1996.

[23] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, K. Strauss, S. R. Sarangi, P. Sack, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[24] B. F. Romanescu, S. Ozev, and D. J. Sorin. Quantifying the impact of process variability on microprocessor behavior. In *Workshop on Architectural Reliability (WAR-2)*, 2006.

[25] T. Sakurai and R. Newton. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. *IEEE Journal of Solid State Circuits*, 25(2):584–594, April 1990.

[26] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. VARIUS: A model of process variation and resulting timing errors for microarchitects. In *IEEE Transactions on Semiconductor Manufacturing*, February 2008.

[27] T. Sato and I. Arita. Constructive timing violation for improving energy efficiency. In L. Benini and M. Kandemir, editors, *Compilers and Operating Systems for Low Power*, 2003.

[28] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *International Symposium on Computer Architecture*, June 2003.

[29] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *International Symposium on Computer Architecture*, June 2003.

[30] A. Srivastava, D. Sylvester, and D. Blaauw. *Statistical Analysis and Optimization for VLSI: Timing and Power*. Springer, 2005.

[31] R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Mitigating parameter variation with dynamic fine-grain body biasing. In *International Symposium on Microarchitecture*, December 2007.

[32] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, Hewlett Packard Labs, April 2008.

[33] A. Tiwari, S. R. Sarangi, and J. Torrellas. ReCycle: Pipeline adaptation to tolerate process variation. In *International Symposium on Computer Architecture*, June 2007.

[34] A. Tiwari and J. Torrellas. An updated evaluation of ReCycle. In *Workshop on Duplicating, Deconstructing, and Debunking*, June 2008.

[35] J. Tschanz, J. Kao, S. Narendra, R. Nair, D. Antoniadis, A. Chandrakasan, and V. De. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *IEEE Journal of Solid State Circuits*, 37(11):1396–1402, February 2002.

[36] J. Tschanz, S. Narendra, A. Keshavarzi, and V. De. Adaptive circuit techniques to minimize variation impact on microprocessor performance and power. In *International Symposium on Circuits and Systems*, May 2005.

[37] A. Uht. Achieving typical delays in synchronous systems via timing error toleration. Technical Report 032000-0100, University of Rhode Island Department of Electrical and Computer Engineering, March 2000.

[38] X. Vera, J. Abella, O. Unsal, A. Gonzalez, and O. Ergin. Checker backend for soft and timing error recovery. In *Workshop on Silicon Errors in Logic — System Effects*, April 2006.

[39] L. Wang. *Adaptive Fuzzy Systems and Control Design and Stability Analysis*. Prentice Hall, 1994.

[40] C. Weaver and T. M. Austin. A fault tolerant approach to microprocessor design. In *International Conference on Dependable Systems and Networks*, July 2001.

[41] Y. Zhang, D. Parikh, K. Sankaranarayanan, and K. Skadron. HotLeakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, University of Virginia, March 2003.

A Fuzzy Controller Basics

Fuzzy controllers have advantages over other machine-learning techniques such as decision trees, perceptrons, and neural networks. First, a fuzzy rule has a physical interpretation, which can be manually extended with expert information. Moreover, unlike perceptrons, they support outputs that are not a linear function of the inputs. Finally, they typically need fewer states and memory than decision trees, and fewer training inputs and memory than neural networks.

Deployment Phase. Given an input vector *X*, the estimated output *z* is generated using the structures of Figure 5 in three steps: (i) for a given input x_j in the vector and rule F_i, find the membership function (Equation 10); (ii) compute the output of the whole rule F_i (Equation 11); and (iii) combine the outputs of all the rules to get the final output (Equation 12).

$$W_{ij} = exp\left[-\left(\frac{x_j - \mu_{ij}}{\sigma_{ij}}\right)^2\right] \quad (10)$$

$$W_i = \prod_{j=1}^{m} W_{ij} \quad (11)$$

$$z = \sum_{i=1}^{n} (W_i \times y_i) \Big/ \sum_{i=1}^{n} W_i \quad (12)$$

Training Phase. The manufacturer-site training uses a large number of input vectors to generate the *n* rules in an FC. With the first *n* vectors, we set μ_{ij} to x_{ij}, which is the value of the j^{th} input in the i^{th} vector. We set σ_{ij} to random values smaller than 0.1 and y_i to the output of the i^{th} vector. Each of the remaining input vectors is used to train all the rules as follows. In the k^{th} step, an input vector will update rule *i*'s μ_{ij}, σ_{ij}, and y_i. Let $\eta(k)$ represent the value of any of these parameters before the update. Let d^k be the output estimated by the fuzzy controller for this input vector using the deployment algorithm, and e the error calculated as $e = (y_i - d^k)^2$. Let α be a small constant representing the learning rate (0.04 in our experiments). As shown in [39], the update rule is:

$$\eta(k+1) = \eta(k) - \alpha \times \left.\frac{\partial e}{\partial \eta}\right|_k \quad (13)$$