HealthLog Monitor: A Flexible System-Monitoring Linux Service

Athanasios Chatzidimitriou George Papadimitriou Dimitris Gizopoulos

University of Athens
Department of Informatics & Telecommunications, Computer Architecture Laboratory
Athens, Greece
{ achatz | georgepap | dgizop }@di.uoa.gr

***Abstract*—Error monitoring is a critical procedure for most computing systems, varying from HPC to embedded systems domains. Several generic architectures have been proposed and employed in modern processors, offering the capability of hardware-level error detection. This critical information is required to isolate and/or mitigate failures. However, research has revealed many cases where indications of upcoming failures can be identified early and before the actual fail occurrence, known as *symptoms*. Such cases become more frequent as technology trends try to exploit the conservative worst-case voltage guardbands and push computing systems towards more aggressive and often hazardous regions.**

In this paper we present HealthLog monitor, a flexible system monitoring service that offers a generic abstraction layer to combine both error and symptom monitoring. HealthLog is capable of monitoring hardware measurements (performance, sensor and errors) as well as external health-related data, allowing combined symptom description and reaction features supported by an API. The scope of the monitor is to offer a universal standard for error reporting and system monitoring mechanisms in all system layers. The current version of HealthLog was developed and tested on AppliedMicro's X-Gene 2 micro-server, but it is a cross-platform solution as it does not depend on a specific architecture. This work demonstrates how platform events, software metrics and external peripheral mechanisms can be combined to deliver early warnings of upcoming failures and trigger evading reactions.

Keywords—reliability, error handling, error reporting, protection mechanisms

I. INTRODUCTION

Following the continuous urge of better and more complex computing systems, technology rapidly improves in multiple directions. Fabrication technologies in the nanoscale allow more hardware to be added in a microprocessor chip and using the same, or even relaxed power constraints. However, the increasing transistor density and the requirement of lower supply voltage have increased the level of variation and decreased the robustness of systems [9] [10]. This results in modern microprocessors to be more vulnerable to faults, caused by cosmic radiation, environmental conditions, aging etc. and countermeasures are required to allow safe operation [16] [17].

Trying to avoid failures derived from these phenomena, hardware protection mechanisms are used for protecting large and vulnerable components of the chip, such as the cache memories where Error Correction Codes (ECC) and Error Detection Codes (EDC) protection is typically selected. Other critical parts, such as the register file of a microprocessor can also be protected against faults [18] [19]. In addition, multiple mechanisms have been proposed in the past for detection and correction of timing faults, which can be caused by voltage emergencies [11] [12]. Apart from the hardware techniques that are present for error detection and correction, a wide range of software techniques are also used for ensuring safe and correct operation [21] [22]. The large number of protection schemes and technologies that are being introduced highlight the severity of threats and urge for reliable operation of computing systems.

In fact, these technologies are also exploited by a different aspect of microprocessor improvement, targeting higher energy efficiency through undervolting. Because of the high variability, manufacturers tend to introduce very conservative voltage and frequency guardbands that will ensure the correct operation of all chips under worst-case situations [15]. However, this extreme caution proves to be overprotecting most of the time, as supply voltage can be adapted to the actual run-time requirements of a chip [20]. Aggressive undervolting approaches however are not always safe as worst-case situations that are covered by the guard-bands can expose the chip into failures. Undervolting can also cause malfunctions in the operation of SRAM cells of the microprocessor chip, which can also introduce faults [13]. In summary, the impact of undervolting can harm the overall reliability of a system, and thus, the requirement of protection is necessary. Traditional mechanisms are not always sufficient for such cases and special undervolting-oriented protection mechanisms have been developed to increase the efficiency and tolerance of systems that operate in off-nominal voltage conditions [14] [23].

Recently proposed techniques can not only detect errors but also identify symptoms of upcoming failures by monitoring both hardware and software metrics [2] [3]. By considering all these aspects, and also the fact that many of these techniques are applicable to different platforms,,we have developed *HealthLog monitor*, a generic framework that bridges the gap between different hardware mechanisms and software technologies and allows flexible combination of internal and external protection, detection and monitoring mechanisms for more efficient reactive and proactive error handling. In this paper we present the structure and functionality of HealthLog

monitor and showcase how it can be used for simplifying sophisticated system monitoring.

The rest of this paper is organized as follows: Section II presents the current available error monitoring tools, Section III describes the layout and features of HealthLog monitor, Section IV shows how HealthLog can be used to deliver sophisticated monitoring and symptom identification while Section V concludes the paper.

II. Related Work

Existing error monitoring tools are tightly bound to hardware-specific mechanisms and architectures, while their functionality is only limited to logging and reporting of errors. This section briefly presents the state-of-the-art of tools that exist today.

A. Machine Check Architecture & Mcelog

Intel systems provide *Machine Check Architecture* (MCA) [8] which is an interface that is responsible for reporting hardware errors like bus, Error Correction Codes (ECC), Parity, cache and Translation Lookaside Buffers (TLB) errors. MCA interface consists of a set of four model-specific 64 bit registers called MSRs. Error-reporting register banks are used for recording any detected correctable or uncorrectable errors. Each error-reporting register bank is associated with a specific hardware unit (or group of hardware units) in the microprocessor. The MCA defines several errors and their location, by providing 16-bit error codes. The errors defined by the MCA are presented in TABLE I and are accompanied by their locality attributes.

TABLE I: MCA error types.

Type	Description
Cache	Errors detected in the generic cache hierarchy.
TLB	Errors detected in instruction or data TLB.
Memory controller	Errors detected in a memory controller.
Bus and interconnects	Errors inside the system buses.

Errors are usually reported to the operating system using the MCA registers. In the case of an uncorrectable error MCA sends an exception called Machine Check Exception (MCE) and a handler is used to collect the details of this error from the registers for further software recovery. Examples of recovery include terminating the affected process or shutting down the system. Corrected errors are polled by a timer or interrupt handler and reported to the mcelog daemon. MCA leverages mcelog [4] that is used to run a daemon for handling and reporting hardware errors. The simplest action in mcelog is to log the error to disk or syslog. The error will be decoded into an ASCII representation containing all information reported by the hardware, like the error address or the type of the error. The default log file is /var/log/mcelog.

B. Linaro RAS & ACPI

Linaro Reliability Availability Serviceability (RAS) daemon is an effort to bring RAS features to the ARM architecture [5]. The existing Linux error reporting modules are mostly architecture specific implementations e.g. mcelog for x86 architectures. With the adoption of the ARM architecture from various vendors, it is necessary to create an architecture independent RAS toolset, which will be available in every Linux environment. Although Linaro is known for developing tools for ARM software ecosystem, the RAS daemon actually runs on x86 as well.

Linaro RAS daemon requires hardware that supports some error detection mechanisms e.g. ECC and is compatible with the Advanced Configuration and Power Interface (ACPI) standards related to error reporting; namely the ACPI Platform error interface (APEI) and Hardware Error Source Table (HEST). APEI [6] and HEST are basically specifications for reporting errors from firmware to OS. APEI and HEST are utilized by Linaro RAS for facilitating error reporting between firmware and kernel.

To deliver error information to the user space, the Linux kernel tracing capabilities are utilized. The ACPI kernel driver collects error information and exports them to trace pipes located in the “Debug Filesystem” (known as debugfs). RAS daemon, which runs in user space, can read an error event from the trace pipes either in poll or probe manner. The daemon can then store the information to a file or a SQLite database and/or forward the error to Automatic Bug Reporting Tool (ABRT) daemon.

C. Linux EDAC Driver

Error Detection and Correction (EDAC) kernel module’s goal is to detect and report hardware errors in a Linux system [7]. EDAC drivers can detect and report memory correctable and uncorrectable errors as well as PCI bus errors. For systems with hardware cache error detectors e.g. ECC; L1, L2 and LLC errors are also reported. For error reporting EDAC drivers use the sysfs interface, typically /sys/devices/system/edac, where file nodes contain types of errors for each category, e.g. edac/mc contains memory controller system errors and edac/pci directory contains PCI errors. Every error report contains all localization information that is available on the hardware platform.

An EDAC driver also reports monotonically increased counters which record the total correctable and uncorrectable errors occurred in a device e.g. a DRAM DIMM. In order to detect errors, EDAC drivers poll the system for error information at regular time intervals. The default value of these time intervals is 1 second. An EDAC driver has some subtle differences compared to other hardware error reporting tools. EDAC drivers require each hardware vendor to provide its own specific implementation, whereas the mcelog or the Linaro RAS attempt to provide a more standardized interface. Moreover, EDAC drivers runs on kernel space while the mcelog and the Linaro RAS run on userspace.

III. HealthLog Monitor Layout

A. Functionality

1) HealthLog Events

The HealthLog monitor includes a set of events (HealthLog events) that describe situations that HealthLog must identify and react with a corresponding action. Events can be either raw-events (discrete) or conditional-events (condition-met). In addition, these events can be generic in their description and are not necessarily bound to the system platform. The details that accompany a HealthLog event are highly related to the features and capabilities of the operating platform, but the event itself is generic. For this reason, the HealthLog is accompanied with platform-specific utility functions that can perform the translation between platform information and HealthLog events. Additionally, to allow adoption of the monitor across platforms, HealthLog allows external utilities to deliver notifications about HealthLog events and furthermore, to define/declare new HealthLog events that are not already described. If, for example, a new peripheral hardware component is capable of delivering information about critical levels of voltage noise or hardware corrected errors, then this information can be passed on to the HealthLog monitor and the event can be integrated to the monitoring procedure. Fig. 1 shows how HealthLog is integrated in a system. At any time, HealthLog keeps a list of its available events which have a unique identifier (id), a human-friendly description (to easily understand what the event is and how it can be used) and a generic data structure which saves all of the information accompanied with the event.

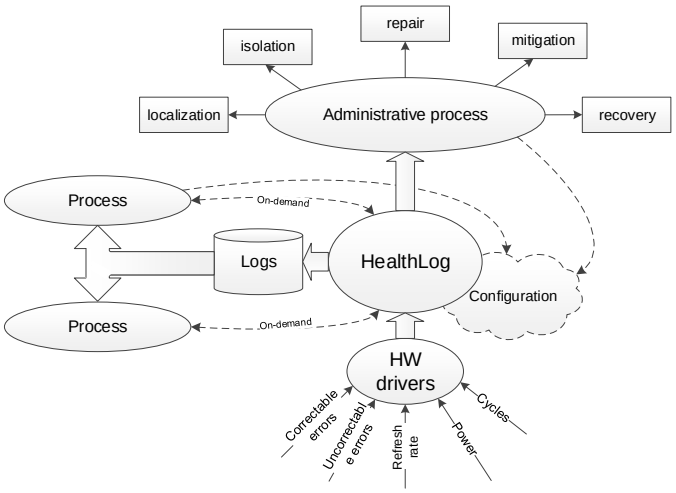

Fig. 1: HealthLog system integration.

Events can be also conditional. This means that HealthLog can trigger (generate) a "condition-met" event, allowing further flexibility and control on the system. A good example of this functionality is the monitoring of critical error rates, such as L2 Cache corrected or uncorrected error rates, which according to a (specified by user or application) threshold, can require an action [1], [2]. In contrast, a bad example of conditional events would be to define conditions that can already be detected by the hardware, for instance, critical temperature. While one can configure the HealthLog to periodically check the SoC temperature and react if critical level is reached, this is also an event that can be delivered as raw by the firmware. Special caution should be given for conditional events, as misuse of the function can lead to redundant and/or excessive overheads.

2) Event-driven & Logging Service

Whenever an event occurs, HealthLog reacts to it by initiating the appropriate actions that are associated to it. These actions are:

- Logging
- Notification
- Data update

Every process that has requested notifications on the event occurrence is notified through the communication socket. The notification is delivered as plain text with the mnemonic "notification" along with the event ID. The ID is known a priori to the process, as it was used to enable the notification and refer to the event. Along with the event, all associated data are updated, including counters and rates. Finally, if the configuration has set the event to be logged, the occurrence is added in the logfile, as presented in Fig. 2. The options and the level of detail is configurable, allowing the user (or the application) to store all of the information of available sensors and metrics that may be considered useful for each situation. In addition, logging can also be encoded to allow easier porting and handling of data by external tools.

3) On-demand interface

HealthLog monitor has a built-in interface for accessing health-related information (metrics, sensors, performance counters etc.), as well as configuration and control of the

```
0000000|Tue Jun 23 20:09:39 2015|<DEBUG>|HealthLog daemon started
0000001|Tue Nov 14 17:30:23 2017|<RAW>|L1D error,CPU:1,Error:Corrected,Multiple:No
0000002|Tue Nov 14 17:30:23 2017|<RAW>|L2 error,PMD:1,Error:Uncorrected,Multiple:No,Type:3,
Action:0,Group:0x0,Way:0x0,Syndrome:0x0,Timeout:No
0000003|Tue Nov 14 17:30:23 2017|<RAW>|MCU error,MCU:0,DIMM:1,Error:Corrected,Rank:0,Bank:0x0
,Row:0x0,Column:0x0
0000004|Tue Nov 14 17:30:23 2017|<RAW>|MCU error,MCU:1,DIMM:0,Error:DemUcErr,Rank:0,Bank:0x0,
Row:0xdead,Column:0xbeef
0000005|Tue Nov 14 17:30:23 2017|<RAW>|L3 error,Source:Tag,Error:Retry,Multiple:No,Group:0x4,
Way:0x0,Bank:0x0,Set:0x0,Syndrome:0x0
0000006|Tue Nov 14 17:30:23 2017|<RAW>|HOT error,Source:SoC,Temperature:0,Channel(mask):0x2a
```

Fig. 2: HealthLog monitor log file sample.

HealthLog daemon. This is achieved with the deployment of a Unix Domain Socket in the filesystem which users (or applications) can open and then communicate with the HealthLog daemon. The socket is used for both reading of the desired information and setting/configuring of the HealthLog daemon. For example, a user can define a new HealthLog event through this socket, deliver notifications about it in user space and receive notifications about event occurrences. Fig. 3 shows the data flow on both event driven and on-demand requests. The core of the on-demand interface has a modular design in which keywords are attached to handlers. For instance, a "read" keyword is attached with a handler that reads information of the system. This allows easy expansion of the on-demand functionality for future revisions of the monitor. The on-demand interface socket is placed at /var/run/healthlog inside the filesystem and each user (process) can have its own socket for communication with the HealthLog.

B. Structure

1) HealthLog Daemon

The main functionality of HealthLog is implemented on the HealthLog daemon, which is the component that receives events from both User and Kernel space (where the low-level handlers receive them from the hardware) and is responsible for filtering, reporting, logging and reacting to these. The daemon delivers both event-driven and on-demand functionality, as we described in the previous subsection.

The daemon is deployed as a system service and it is designed to receive HealthLog event notifications by deploying a userspace link (this is actually a Linux pipe) and by connecting to devfs /dev/healthlog for receiving events from the kernel space. If the device node is not available (e.g. no module loaded), the daemon will only receive notifications from userspace. While the HealthLog daemon is running and serves the requests from users or applications, it keeps logs all the necessary information into a file. The daemon deploys the actual log file, the HealthLog at /var/log/healthlog, where all of the configured events are logged. By default, the log contains all of the information regarding event occurrences and debug information in text format. The log levels are configurable to discard or include HealthLog configuration messages and debugging events and the output format can be changed between text and encoded data. The option to log the information using external handlers is also supported (e.g. a database or XML files).

2) HealthLog Kernel Module

The HealthLog Kernel Module is the HealthLog component designed to operate within the Kernel Space context. The purpose of the module is to act as the "listener" of the monitor in the Kernel Space by providing an API for other modules to deliver their events to the HealthLog daemon. The module has built-in implementation and support for some platforms (such as Applied Micro's (APM) X-Gene micro-servers). The HealthLog module uses *sysfs* to deliver information on the HealthLog daemon (by deploying a character device node /dev/healthlog). The device node is designed to have only one open instance (only 1 process can read from it) and can be configured to deliver either binary data structures or human readable text. Binary mode is required for operation by the HealthLog, while text mode is available for alternative usage of the module (user or third-party clients).

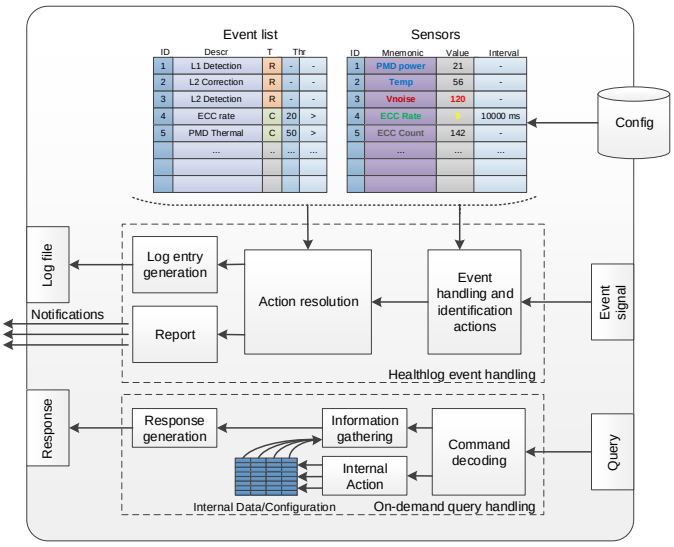

Fig. 3: HealthLog data flow.

IV. Enhanced Monitoring

HealthLog was developed and tested on APM's X-Gene 2 platforms and has built-in support for X-Gene's hardware exposure interface, which is used to deliver messages from hardware protection mechanisms, as well as system sensors and counters. In this section we will present how the hardware supported mechanisms can be combined with externally installed monitors to proactively handle and avoid upcoming unsafe situations. To do so, we use as example the technique presented in [3].

In [3], the authors have developed a novel technique where the electromagnetic emanation of the chip can be used as a proxy of a potential voltage droop magnitude. This information can be considered as an active guardband for the supply voltage. Low electromagnetic emanation can indicate low voltage noise (thus high guard band) while high emanation indicates higher voltage noise thus a narrower guard band. The monitoring setup requires the installation of an external antenna and peripheral infrastructure, which is not part of the platform itself. The driver application of this installation however can be easily configured to comply with the HealthLog API and be integrated in the health monitoring procedure, which can allow adaptive voltage configuration depending on the electromagnetic emanation.

The application can register a new HealthLog event of "critical emanation level", where it will itself deliver the event occurrence or register a new sensor input, which will be polled by the HealthLog. In the latter case, this sensor can be combined with a custom threshold level for describing a conditional event. In such a way, the specific conditional event will be enabled in the HealthLog daemon to start monitoring the desired sensor. When the conditions are met, the HealthLog daemon will log and/or report (depending on what action was requested by the application) to the application the appropriate responses. The notification of the process occurs immediately,

as critical mitigation reactions may be required. This is what we will showcase in this section. To further simplify the procedure, different conditional events can be assigned for each voltage level. The example is presented in Code 1.

Code 1: Registering new sensor.

```
// UNIX-CONNECT:/var/run/healthlog
→ sensor new VNoise "Voltage Noise"
← 34
→ sensor notify 34 150
→ sensor notify 34 120
→ sensor notify 34 123
→ sensor notify 34 150
```

This will register a new sensor with description "Voltage Noise" on the HealthLog which will be assigned the sensor id "34". All sensor values are 32-bit integers. Then the application regularly updates the sensor value. The interval is configured on the application side and at any given time, HealthLog will keep the latest available value. Built-in metrics can be assigned with an interval for updating. With the voltage noise being regularly updated, a new conditional event can be assigned. Following a system characterization process, we demonstrate how an *Adaptive Voltage Driver* that will utilize the findings of the characterization (system Vmin, guardbands and Vmin per voltage noise level) can configure the HealthLog monitor to simplify the system integration.

Code 2: Registering new conditional event.

```
// UNIX-CONNECT:/var/run/healthlog
→ event cond 34 gt 200
← 50
→ event log 50
→ event enable 50
→ listen
                    …
← notification 50
```

In Code 2, a new conditional even was created, assigning sensor ID 34 (Vnoise) when greater than (gt) value 200. The event receives the ID 50 and is then marked to be logged and enabled notification. Then the client falls to "listen" mode, waiting for event notifications. Once the condition is true, the client receives a notification. Notice that, at any time, the client can close the socket and discard its state. However the registered events will remain valid on the system and, in this example, will be logged. At any time, the list of available sensor values and events can be viewed along with their IDs and descriptions. Conditional events can also combine 2 logical conditions using logical AND and OR. The previous example can be enhanced with the inclusion of current voltage setting, as shown in Code 3.

Code 3: Registering new conditional event with 2 conditions.

```
// UNIX-CONNECT:/var/run/healthlog
→ sensor read_list
← 1 – PMD Power sensor
← 2 – SoC Power sensor
                    …
← 25 – PMD Voltage
                    …
→ event cond 34 gt 180 and 25 eq 940
← 51
→ event cond 34 gt 190 and 25 eq 960
← 52
→ event enable 51
→ event enable 52
→ listen
```

These queries will create specific Vnoise thresholds for each voltage level configuration. Note that the condition checks are always performed on the up-to-date values of each sensor. This means that in this case, the voltage needs to be updated by either a "read" command upon change, or by a polling interval. With this configuration, the client of HealthLog (which in this case can be an adaptive voltage driver) can configure the monitor to notify upon critical situations and accordingly change the operating voltage. In addition, low noise events can be added to trigger lowering of the supply voltage.

Every event of HealthLog (raw and conditional) is implicitly associated with a counter and a rate, and these are also listed in the "sensor" list. For easier usage, the counters and rates are associated with large sensor IDs (starting with 65,535 and downwards). For each rate sensor, the interval is configurable in 100 ms granularity. To further expand the previous example, we will use the methodology of [1] and [2] and integrate it in our example adaptive voltage driver. The authors of [1] [2] show how cache memories introduce faulty SRAM cells as they operate in low supply voltage. The rate of these faults (and corresponding rate of ECC events) can be used to drive the level of undervolting.

Since the ECC errors are already provided by the hardware, HealthLog provides counters for their occurrence rates. To integrate this mechanism in our configuration, it is required to configure the interval of the ECC rate and to define a conditional event with the critical threshold for the ECC rate. Each time the rate is above the threshold, the driver will receive a notification of critical rate and can react by increasing the voltage level. This is presented in Code 4.

Code 4: Setting interval and monitoring ECC rate.

```
// UNIX-CONNECT:/var/run/healthlog
→ sensor read_list
                    …
← 65509 – L1 Data ECC rate
← 65510 – L1 Instruction ECC count
← 65511 – L1 Instruction ECC rate
                    …
→ sensor interval 65509 10000
→ event cond 65509 gt 10
← 53
→ event enable 51
→ event enable 52
→ event enable 53
→ listen
```

In this example, the sensor is assigned an interval of 10,000 ms and a new conditional event is defined with a threshold of 10. If more than 10 L1D ECC events per 10 seconds occur, then the new event 53 will be triggered and the driver will receive a notification of the critical error rate, to react accordingly.

The Kernel space API provides calls for registering raw events, sensors and provide notifications in similar practice. However, the introduction of new conditions and interval configurations for the sensors are only available in userspace. With the use of HealthLog API, external support can be added to any platform using compatible hardware drivers. In addition, failures in higher system layers can also be added and considered by the HealthLog monitor. For instance, if the Kernel modules perform TCP packet repairs, or filesystem drivers perform repairs, this information can be passed to the HealthLog and participate on mitigation process of different levels, consisting HealthLog monitor as a flexible generic health-monitoring interface.

V. CONCLUSIONS

Advanced methodologies on failure prediction and detection tend to require monitoring and capturing of multiple on-system and off-system measurements. Existing error monitoring tools operate primitively and limit their function on exposing hardware error reports. We have developed the HealthLog monitor aiming to deliver enhanced functionality on system-health monitoring. HealthLog enables users to include custom dedicated health-related information on the system and combine them with existing hardware events and metrics. Events raised by peripheral devices, drivers and the whole software stack can be included under the same framework allowing increased flexibility on system monitoring.

HealthLog provides an easy-to-use and low-overhead API for both user and kernel space. We have demonstrated how HealthLog could be used to simplify the implementation of two novel approaches of proactive failure prediction for aggressive undervolting ([2] and [3]) and easily combined under a common driver. Apart from the complex functionality, HealthLog also provides fundamental services on error monitoring and several essential metrics reporting. The HealthLog consists of a kernel module and a daemon to provide API functionality. The daemon is the component that receives platform events and is responsible to filter, report, log and react to these. It also provides the following types of services: (a) event-driven services, (b) on-demand services. The Kernel module allows easy adoption of existing drivers and direct hardware access. The tool is intended to provide a common standard for advanced error reporting functionality.

ACKNOWLEDGEMENT

This work is funded by the H2020 Framework Program of the European Union through the UniServer Project (Grant Agreement 688540 – http://www.uniserver2020.eu) and SARG of the National and Kapodistrian University of Athens.

REFERENCES

[1] A. Bacha, and R. Teodorescu, “Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors”, in Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA), 2013.

[2] A. Bacha, and R. Teodorescu, “Using ECC Feedback to Guide Voltage Speculation in Low-Voltage Processors”, in Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO), 2014.

[3] Z. Hadjilambrou, S. Das, M. A. Antoniades, Y. Sazeides, “Sensing CPU Voltage Noise Through Electromagnetic Emanations”, Computer Architecture Letters, Vol.: 17, No.: 1, pp: 68-71, 2018.

[4] A. Kleen, “Mcelog: memory error handling in user space”, Available: http://www.halobates.de/lk10-mcelog.pdf - Retrieved on Feb 20, 2018.

[5] “Linaro Kernel,” [Online]. Available: https://wiki.linaro.org/LEG/ServerArchitecture/RAS/RAS_daemon - Retrieved on Feb 20, 2018.

[6] ACPI Platform,” [Online]. Available: https://firmware.intel.com/sites/default/files/resources/A_Tour_beyond_BIOS_Implementing_APEI_with_UEFI_White_Paper.pdf - Retrieved on Feb 20, 2018.

[7] Linux EDAC Documentation, Available: https://github.com/tinganho/linux-kernel/blob/master/Documentation/edac.txt - Retrieved on Feb 20, 2018.

[8] Intel Corporation, “Intel 64 and IA-32 Architectures Software Developers Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C”, September 2014.

[9] R.C.Baumann, “Soft errors in advanced computer systems”, In IEEE Design & Test of Comp., vol. 22, no. 3, pp. 258-266, May/June 2005.

[10] F. Salehuddin, I. Ahmad, F.A. Hamid, A. Zaharim, A. Maheran, A. Hamid, P. S. Menon, H. A. Elgomati, B. Y. Majlis. “Optimization of process parameter variation in 45nm p-channel MOSFET using L18 Orthogonal Array”, IEEE International Conference on Semiconductor Electronic (ICSE), 2012.

[11] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, T. Mudge, “Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2003.

[12] V. J. Reddi, M. S. Gupta, G. H. Holloway, G.-Y. Wei, M. D. Smith, and D. M. Brooks, “Voltage emergency prediction: Using signatures to reduce operating margins”, in International Conference on High-Performance Computer Architecture (HPCA), 2009.

[13] S. Ganapathy, J. Kalamatianos, K. Kasprak, S. Raasch. “On Characterizing Near-Threshold SRAM Failures in FinFET Technology”, In Proceedings of the 54th Annual Design Automation Conference (DAC), 2017.

[14] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, “Trading off cache capacity for reliability to enable low voltage operation”, in International Symposium on Computer Architecture (ISCA), 2008.

[15] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers, S. Das, “Harnessing Voltage Margins for Energy Efficiency in Multicore CPUs”, In Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO), 2017.

[16] C.Constantinescu, “Trends and challenges in VLSI circuit reliability”, IEEE Micro, vol. 23, pp. 14-19, July 2003.

[17] S.Nassif, N.Mehta, Y.Cao, “A resilience roadmap”, DATE 2010.

[18] Pablo Montesinos, Wei Liu and Josep Torrellas, “Using Register Lifetime Predictions to Protect Register Files Against Soft Errors”, DSN 2007.

[19] M. Fazeli, S.N. Ahmadian, S.G. Miremadi, “A Low Energy Soft Error-Tolerant Register File Architecture for Embedded Processors”, HASE 2008.

[20] Y. Zu, C. R. Lefurgy, J. Leng, M. Halpern, M. S. Floyd, V. J. Reddi, "Adaptive guardband scheduling to improve system-level efficiency of the POWER7+", In Proceedings of the 48th International Symposium on Microarchitecture, (MICRO), 2015.

[21] N. Oh, P . P. Shirvani, and E. J. McCluskey, “Error Detection by Duplicated Instructions in Super-Scalar Processors”, IEEE Transactions on Reliability, 51(1):63–74, Mar. 2002.

[22] G. A. Reis et al., “SWIFT: Software Implemented Fault Tolerance”, In International Symposium on Code Generation and Optimization, 2005.

[23] J. Kim, et all., “Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding”, 40tth Intternattiionall Symposiium on Miicroarchiittectture (MICRO), 2007.