ViRUS: Virtual Function Replacement Under Stress

Lucas Wanner and Mani Srivastava
Electrical Engineering Department — University of California, Los Angeles
Email: {wanner, mbs}@ucla.edu

***Abstract*—In this paper we introduce ViRUS: Virtual function Replacement Under Stress. ViRUS allows the runtime system to switch between blocks of code that perform equivalent functionality at different Quality-of-Service levels when the system is under stress — be it in the form of scarce energy resources, temperature emergencies, or various sources of environmental and process variability — with the ultimate goal of energy efficiency. We demonstrate ViRUS with a framework for transparent function replacement in shared libraries and a polymorphic version of the standard C math library in Linux. Case studies show how ViRUS can tradeoff upwards of 4% degradation in application quality for a band of upwards of 50% savings in energy consumption.**

I. Introduction

Energy efficiency in software is becoming an increasingly dynamic problem, influenced by various sources of stress including Process, Voltage, and Temperature variations, remaining battery capacity at any time, and user behavior patters (e.g. in mobile phone app usage or charging habits). Any given energy management decision may have a significant positive impact on battery lifetime under one context, while being innocuous under different environmental and usage conditions. Energy management actions should therefore be informed by device state and context.

In this paper we introduce the concept of ViRUS: Virtual function Replacement Under Stress. ViRUS is loosely related to polymorphic engines [14] in that it is used to transform sections of a program into different versions with alternate code paths that perform roughly the same functionality. Polymorphic engines are used to intercept and modify code transparently, typically for malicious purposes such as hiding malware functionality from anti-virus software. In ViRUS, the different code paths provide varying quality-of-service for different energy costs. Transformations are triggered by sensing and adapting to various sources of energy stress. A block of code may be activated in ViRUS, for example, when processor temperature reaches a certain threshold. A second block may be activated when remaining battery capacity drops below a specified percentage. The different code blocks may be either standard library functions provided by the runtime system or alternative implementations provided by application programmers. Per-application configuration files determine when and under what circumstances transformations are triggered. The runtime system monitors sensors for energy stress and transparently triggers adaptation at appropriate times.

We present a realization of ViRUS in the form of a framework for transparent function replacement in shared libraries demonstrated with a polymorphic version of the standard C math library in Linux. Microbenchmarks show that the system can monitor hardware and environmental conditions and trigger adaptations with negligible overhead. We show how ViRUS can help users in developing and analyzing tradeoffs between accuracy and energy consumption in different contexts and reduce the energy consumption with user-defined quality-of-service degradation.

II. Related Work

Alternate code paths, or *algorithmic choice* have been explored in the energy-aware software literature. Petabricks [1] and Eon [12], for example, feature language extensions that allow programmers to provide alternate code paths. The runtime system dynamically chooses paths based on energy availability. In Petabricks, multiple versions of object code are created and profiled for execution time and quality using a sample set of input data. Paths for an application may be chosen statically or altered in runtime through accuracy valuations. A similar process is used in Green [2], where a combination of a calibration phase and runtime accuracy sampling are used by the application to define which function to execute from a set of candidates. In Eon, the runtime system dynamically chooses paths based on energy availability. In Levels [7] programmers define multiple versions of a task, and the run-time chooses the highest quality task levels that will meet a required battery lifetime.

Algorithms and libraries with multiple implementations can be matched to underlying hardware configuration to deliver the best performance [6], [8]. Such libraries can be leveraged to choose the algorithm that best tolerates the predicted hardware variation and deliver a performance satisfying the quality-of-service requirement. With support from the OS, the switch to alternative algorithms may be done in an application-transparent fashion by re-linking a different implementation of a standard library function. Similarly, application code may be optimized for performance and energy with input from the programmer indicating code regions [5] or data [3], [11] that may be amenable to approximation.

While these systems provide valuable design references for approximation and algorithmic choice, many of their assumptions do not hold true in the presence of dynamic vectors of energy stress. In Petabricks, execution time is the primary resource usage metric. Similarly, in Green power is assumed to be a function of execution time. With variability in active mode power, the energy cost of a fixed number of CPU cycles varies across instances and ambient conditions. Both systems rely on a calibration phase to reduce runtime overhead of evaluating quality-of-service and cost of different code paths. With variability, runtime cost of a given code

path will also be variable across nominally identical devices and across the lifetime of a device, due both to aging and changes in operating conditions. Levels triggers chances in run levels based on a history of power consumption and remaining lifetime of the system. As with the calibration phase in Petabricks and Green, a projection of future power consumption based on past history may lead to overly conservative or optimistic adaptation decisions due to variations in power consumption across time and due to ambient conditions.

III. System Design

We propose ViRUS as an application runtime support system where the operating system adjusts service quality according to energy-aware policies. To accomplish this, we (i) leverage techniques for algorithmic choice, building this capability into the runtime system; (ii) allow for adaptation with minimal application intervention (iii) expose service quality information to the applications, so that developers can configure adaptation according to application requirements; (iv) build energy-aware adaptation policies that expose sensor information to the software stack and drive system adaptation.

A. System-Driven Algorithmic Choice

Several services provided by the operating and runtime support are amenable to adaptation and may be extended to support elastic quality levels. These services include numeric and signal processing services (e.g. with variable numeric precision), multimedia services (e.g. with variable video encoding and decoding quality), sensing stack (e.g. the aforementioned location service), and communication stack (e.g. diversity of communication channels). Different versions of each of these services can be provided to applications through a common interface, or through a wrapper that adapts versions with different interfaces.

In ViRUS we build a thin wrapper around shared libraries to provide context-aware algorithmic choice. This wrapper takes adaptation configuration parameters from applications (e.g. what functions are sensitive to sources of stress and amenable to adaptation, bounds for quality requirements), adapts multiple libraries with common functionality but different function signatures into a common interface, and handles messages from the OS that trigger function replacement.

B. Library Generation

Given multiple implementations $f_0(\cdots)$, $f_1(\cdots)$, $\cdots$, $f_n(\cdots)$ of a function featuring the same signature, ViRUS exposes a single function $f(\cdots)$ to applications. In our C language implementation, this is accomplished by declaring $f(\cdots)$ as a function pointer, and dynamically assigning the pointer to the address one of the $f_n(\cdots)$ implementations. Because calls to functions and function pointers are identical in C, there is no indirection between a call to the function entry point f and the function call $f_n(\cdots)$. In other words, the function pointer $f(\cdots)$ is aliased to one of the implementations; for example, if the function pointer $f(\cdots)$ is assigned to $f_1(\cdots)$, an application call to $f(\cdots)$ translates to the same sequence of operations as a call to $f_1(\cdots)$.

function	priority	sensor	range	quality
f	0	temperature	$[0, 40)$	$\{0, 1\}$
f	0	temperature	$[40, 100)$	$\{2, 3\}$
g	0	battery	$[20, 100)$	$\{0, 1, 2\}$
g	0	battery	$[0, 20)$	$\{2\}$
g	1	temperature	$[0, 60)$	$\{0\}$
g	1	temperature	$[60, 100)$	$\{1, 2\}$

TABLE I: ViRUS configuration rules example.

Taking a design reference from [2] and [7], we expose the multiple implementations of a function as a function array ordered by quality. For each f, a four-function API is exposed: f itself, two mutator methods for quality level (getter/setter), and a method that returns the number of implementations available. In general, multiple implementations of a function f may have different type signatures. For every f_n function implementation we therefore must create a wrapper function that matches the signature of pointer f. Function wrappers and mutators are created automatically in ViRUS using a series of C pre-processor macros. For each function, a constructor method is automatically generated and executed to set the default function pointer and quality level when the library is first loaded This method also registers the name of the function along with its mutator methods with the ViRUS controller that handles messages from the operating system and triggers function adaptation according to application and system configuration.

C. ViRUS controller

The ViRUS controller monitors hardware and operating environment for sources of stress and dynamically triggers function adaptation according to system and application configuration. Each application using a ViRUS library has its own controller featuring all control knobs (function mutators), function replacement rules, and sensor monitors. Each application is associated with a configuration file listing its function replacement rules. Each rule in the configuration links a function with a stress sensitivity vector and a set of acceptable quality levels. A sensitivity vector is defined as a range of values for a sensor, for example, temperature between 0 and 40°C or instant active power between 100 and 500mW. The set of acceptable quality levels is defined as an integer range from the highest to lowest quality acceptable for each function while operating under each sensitivity vector.

Table I shows a ViRUS configuration table for a hypothetical application. Each rule describes a function, priority, sensor, range for the sensor, and set of acceptable quality levels. The application is sensitive to battery level and temperature. Function f may operate in quality levels 0 or 1 when temperature is between 0 and 40°C, and quality levels 2 or 3 when temperature is between 40 and 100°C. Lower numbers represent higher quality levels. When multiple sensors may trigger mutations for the same function, rules are resolved in order of priority, with lower numbers representing higher priority.

```
Data: Rules: set of (knob, priority, sensor, range, quality) tuples
RulesToProcess ← Rules;
KnobName ← ⋆ ;
while RulesToProcess ≠ ∅ do
    RulesForKnob
    ← {r ∈ RulesToProcess : r.knob.name = KnobName};
    MatchedRules ← ∅;
    forall the rule ∈ RulesForKnob do
        sensor ← rule.sensor.value();
        rmin ← rule.range.min;
        rmax ← rule.range.max;
        if rmin ≤ sensor < rmax then
            MatchedRules ← MatchedRules ∪ rule;
        end
    end
    QualitySet ← {r.quality ∀r ∈ MatchedRules } ;
    forall the rule ∈ MatchedRules ordered by rule.priority do
        if (QualitySet ∩ rule.quality) ≠ ∅ then
            QualitySet ← QualitySet ∩ rule.quality;
        end
    end
    Set quality of functions matching KnobName to
    max(QualitySet) ;
    RulesToProcess ← RulesToProcess \ RulesForKnob;
    KnobName ← rule.knob.name for some rule ∈
    RulesToProcess ;
end
```

Algorithm 1: Function replacement algorithm

Function g may operate in quality levels 0, 1, or 2 when remaining battery is between 20 and 100%, but only in quality level 2 when the battery level is below 20%. A lower priority rule for temperature further refines the choices to level 0 when temperature is between 0 and 60°C, and levels 1 and 2 when temperature is between 60 and 100°C. Due to different priorities, function g will operate at quality level 2 when battery is below 20% even if temperature is below 60°C. When rules are defined for a function with different sensors and the same priority level, the most energy conservative rule (i.e., the one with the lowest quality set of alternatives) is interpreted as having the highest priority. Likewise, when multiple quality choices are available for a function after resolving all the rules, the lowest quality version is chosen. A special function name, $\star$, is used for rules that apply to all functions. If an application configuration mixes function specific and wildcard rules, function-specific rules override wildcard rules.

Stress sensors are exported to the ViRUS controller by a sensor monitor (described in Section III-D) as a table of sensor names and a floating point variable to access current sensor values. The ViRUS controller is implemented as a library that is linked with each application. A constructor method parses the rules file for the application. Each line in the file reflects a line in Table I. If no configuration file is found for the application, a default global configuration file is used.

Adaptations are triggered by a message from the sensor monitor. These can be of a periodic nature or range-based alarms for sensors, according to application configuration. The controller library constructor configures alarms and installs a message handler. When a message is received, we iterate through all the rules to find replacements for each function using Algorithm 1. For every unique knob name, we find the corresponding set of rules, starting with the wildcard ($\star$) rules. For each of the rules for a knob name, we check current sensor values against the range for the rule. If the sensor value is within range, we mark the rule as matched. Starting with a set of all possible quality levels, we iterate through matched rules in order of priority to refine the set of allowable quality levels. Finally we set the quality of the function associated with the rule (or all functions in the case of a wildcard rule) to the maximum of the set of allowable qualities after processing the rules. If this quality is a higher number than the number of alternate implementations provided the function will be set to the lowest quality (highest number) available.

D. Stress Monitor

ViRUS monitors stress vectors for energy usage and notifies applications to trigger function replacements. The monitor is built with two components: a system daemon that monitors sensors and triggers alarms upon certain conditions, and a small library linked with applications that registers processes to receive adaptation triggers and handles notifications from the daemon.

The application library interacts with the monitor through a socket (to register the application), files (to discover available sensors and to read sensor values), and signals. When the process receives a signal from the monitor, the application must read current sensor values to determine if function mutations should be triggered. Because the signal itself does not convey information other than signal number, sensor values are communicated from the monitor daemon to the application through a shared file. This file contains the last sample for each sensor in the same order as they appear in the sensor descriptor file. Values are parsed and the sensor table for the application is updated.

In the sensor monitor daemon hardware and software sensors are abstracted through drivers that expose the sensor name and current value through a floating point function. Available sensors depend on the underlying hardware platform, and are linked with the monitor system service at compile time. In the future this could be extended to allow for dynamic discovery of sensors through a plug-and-play driver architecture. The version of ViRUS used in our evaluation includes sensors for temperature, frequency, voltage, and average active and sleep power.

For each sensor, a user-defined configuration file determines sampling rate in Hz and alarm rules. Four types of alarm are defined: value greater than, value equal to, value smaller than, and change in magnitude. The change in magnitude rule will raise an alarm if the sensor has changed by the expressed percentage since the last sample. The four basic alarm types can be dynamically changed without requiring re-compilation, but more complex alarm rules must be attached to sensors through a software module in compile time.

After acquiring a sample, the monitor iterates though

sensor rules to search for a match. If a match is found, a signal is sent to all processes that registered their process identifiers (PIDs) with the monitor. If a signal can't be sent to a process, that process is assumed to have finished, and the PID is removed from the list of registered processes. While alarms are global, function mutation rules are defined on a per-application basis and therefore a signal may not lead to a mutation.

IV. Evaluation

For the results in this section, we built our ViRUS libraries and applications for the ARM9 architecture and ran it on a prototype Linux system using the VarEMU virtual machine emulator [13]. VarEMU provides users with the means to emulate variations in power consumption and to sense and adapt to these variations in software. Energy results are normalized across the runs under comparison, and obtained from a nominal (typical) instance under nominal voltage and temperature. We use the `-O3` compiler optimization flag for all tests.

A. Memory and Runtime Overheads

The ViRUS controller is implemented as a shared library that handles: 1) knob registration for multi-quality functions; 2) parsing of application-dependent mutation rules; 3) registration and handling of signals from the sensor monitor and reading of sensor data; and 4) function replacement. Combined, these functions use a total of approximately 2.8KB of code memory. When standard library functions needed to perform operations in the ViRUS controller (e.g., `strcmp`, `connect`) are statically linked with the library in an application, code memory usage totals approximately 8KB. Internal variables that keep adaptation knobs (i.e., available functions) and replacement rules add approximately 7KB of data memory for each application. Code memory used by the multiple versions of a function depends on the nature of the function and the implementations, as shown in Section IV-B.

The runtime overhead of ViRUS can be divided into one-time and periodic operations. One-time operations include process and knob registration, sensor discovery, and rule parsing. Assuming an app with ten knobs and ten replacement rules, and a processor frequency of 1 GHz, ViRUS adds approximately 0.7 ms to application initialization. Periodic operations happen every time there is a signal from the sensor monitor to the ViRUS handler in the application process. This triggers sensor reading and rule matching (function replacement) in the application, and corresponds to approximately 70 μs under the assumptions above.

B. Variable Quality Math Library

We implemented a ViRUS library for the mathematical operations of the C standard library declared in the `math.h` header file. For this implementation, we use the standard double and single precision `libc` functions as quality levels 0 and 1 respectively. For subsequent quality levels, we use implementations from the `fastapprox` [9] library. This library provides approximate versions of functions commonly used in machine learning, including exponential, logarithm, power, cos, sin, tan, and others.

Function	Memory Usage (KBytes)					Combined
Quality	Qual: 0	1	2	3	Comb.	ovhd. (%)
exp	31.2	16.2	3.4	3.1	34.7	11.2
log	40.9	15.8	2.8	2.6	44.0	7.6
pow	43.6	18.8	3.7	3.2	50.1	14.9
sin	70.7	9.8	2.7	2.6	79.9	13.0
cos	70.7	9.8	3.0	2.6	79.9	13.0
tan	79.3	9.7	3.1	3.1	88.5	11.7
asin	67.1	16.0	-	-	70.2	4.6
acos	67.1	16.2	-	-	70.3	4.8
atan	37.3	3.6	-	-	39.8	6.9
sinh	33.4	17.7	3.5	3.2	38.6	15.5
cosh	33.4	17.7	3.5	3.2	38.5	15.4
tanh	4.4	4.3	3.5	3.2	8.1	86.7
lgamma	103.5	19.8	3.1	2.8	110.8	7.1
comb.	227.8	64.8	4.5	4.1	251.8	10.5

TABLE II: ViRUS math library memory usage

Table II shows total code memory usage for significant methods in the ViRUS math library. To measure code size of each function, we compare the size of the `.text` segment of a statically linked application binary where the application calls only one of the multiple versions and prints the result with an application that prints a constant number. Because the multiple quality methods may share portions of code, the combined memory usage of the multiple versions of a function is typically smaller than the summation of memory usage for each of the individual versions. Likewise, the combined total for all library functions is less than the summation of the individual functions.

The rightmost column in Table II shows the overhead of the combined multiple quality methods compared to the single highest quality (double precision) method. The overhead of providing multiple versions of a function over providing only its double precision version averaged 16% and ranged from 5% to 86%. The combined memory usage of all methods on the table was approximately 252 KB, compared to 228 KB for all the methods in double precision only—an increase of 10%.

C. Application case studies

In this section we show how ViRUS can use polymorphic libraries to adjust application quality and energy consumption to counteract system stress from process and environmental variability. The potential energy benefits of ViRUS are limited by the fraction of time and energy that an application spends using the polymorphic library functions provided by the system—standard math library functions in our case studies. In cases where applications do not use significant runtime support system functions, ViRUS may still be useful in managing application-level adaptation. Developers can register multiple versions of a function as knobs by using and linking with the ViRUS controller applications. This modality of application-driven algorithmic choice has been explored in the literature, e.g. in [2], [7], [1]. In our evaluation, we show examples of benchmark applications where library functions represent a significant fraction

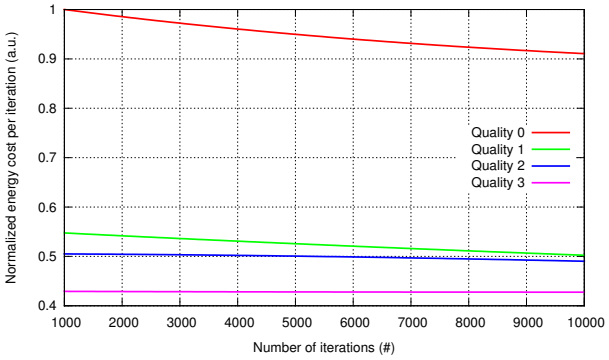

Fig. 1: Energy for `whestone`

of application energy and time cost and library implementation impacts application quality: the `whetstone` benchmark, and the `blackscholes` and `swaptions` applications from the Parsec suite [4] .

1) whetstone : We modified the Netlib implementation [10] of Whestone to use the ViRUS math library, and profiled energy consumption when each of the different quality levels was used. Energy numbers were obtained by running a nominal instance of VarEMU [13] under nominal operating conditions (voltage, frequency, and temperature). For each run, a global destructor in the application is called upon termination and prints out total accumulated energy as determined by the VarEMU virtual machine monitor. ViRUS configuration files provide rules that lead to immediate switching to a desired quality level upon application initialization. No variation alarms are generated for the run. Energy results are normalized to highest energy cost per iteration (highest quality, smallest number of iterations).

Figure 1 shows normalized energy consumption for `whetstone` under different quality levels and across a variable number of iterations for each run. The greatest benefit in energy consumption results from switching from the double precision (highest quality) versions to the single precision versions, which leads to an average 45% reduction in energy consumption. There is a further relative benefit of 8% when going from the single precision version to the first approximate version, and 15% when going from the first approximate version to the lowest quality approximate version. Going from the highest to the lowest quality version results in a 57% reduction in energy consumption.

2) blackscholes: We modified the `blackscholes` application [4] to use the ViRUS math library and profiled its energy consumption as described for `whestone`. Table III shows normalized energy consumption and output quality for `blackscholes` under different quality levels. We use two metrics to analyze output quality: normalized root mean square error (NRMSE) and mean absolute percentage error (MAPE). Going across approximate levels in `blackscholes` increases errors by 2–4 orders of magnitude for each step. Going from quality level 0 (double precision) to 1 (single precision) led to a 25% reduction in energy consumption. Going from level 1 to 2 (first approximate version) resulted in a further 18% reduction in energy. From the first to the second approximate version (quality level 2 to 3), there is a 20% energy benefit. Across the board, from the highest to lowest quality version, there is a 52% energy consumption band.

Quality	Normalized Energy	NRMSE (%)	MAPE (%)
0	1	—	—
1	0.74	0.000005	0.00002
2	0.6	0.003	0.1
3	0.48	4.3	39.7

TABLE III: Energy and quality for `blackscholes`

Quality	Normalized Energy	NRMSE (%)	MAPE (%)
0	1	—	—
1	0.85	0.0000004	0.000002
2	0.76	0.002	0.02
3	0.69	0.57	3.8

TABLE IV: Energy and quality for `swaptions`

3) swaptions: Table III shows normalized energy consumption and output quality for `swaptions` [4] under different quality levels. Going across approximate levels in `swaptions` increases errors by 2–5 orders of magnitude for each step. Acceptable results are produced for all quality levels. In the lowest quality level, MAPE is approximately 4%. Going from quality level 0 (double precision) to 1 (single precision) led to a 15% reduction in energy consumption. Going from level 1 to 2 (first approximate version) resulted in a further 11% reduction in energy. From the first to the second approximate version (quality level 2 to 3), there is a 9% energy benefit. Across the board, from the highest to lowest quality version, there is a 30% energy consumption band.

V. Conclusion

We introduced ViRUS (Virtual function Replacement Under Stress), an application runtime support system where the operating system adjusts service quality according to energy-aware policies. We demonstrated ViRUS with a framework for transparent function replacement in shared libraries and a polymorphic version of the standard C math library in Linux. The ViRUS control framework uses less than 3KB of RAM, and the polymorphic math library adds 10% memory overhead to its comparable single choice, high precision version. Application case studies using the polymorphic math library showed how ViRUS can tradeoff upwards of 4% degradation in application quality for a band of upwards of 50% savings in energy.

One implicit assumption in ViRUS is that alternate implementations of a function may be ordered by quality and energy cost. Higher cost functions are assumed to produce higher quality results. For certain applications and functions, quality and cost may be input dependent, and hence this ordering may change dynamically. In future work we intend to explore enhanced cost/quality profilers that could assign or suggest rules for application adaptation.

REFERENCES

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. *SIGPLAN Not.*, 44:38–49, 2009.

[2] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Not.*, 45:198–209, June 2010.

[3] Kenneth C. Barr and Krste Asanović. Energy-aware lossless data compression. *ACM Trans. Comput. Syst.*, 24(3):250–291, August 2006.

[4] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] H. Esmaeilzadeh, A Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 449–460, Dec 2012.

[6] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005.

[7] Andreas Lachenmann, Pedro José Marrón, Daniel Minder, and Kurt Rothermel. Meeting lifetime goals with energy levels. In *SenSys*, 2007.

[8] X. Li, M.J. Garzaran, and D. Padua. Optimizing sorting with machine learning algorithms. In *IPDPS*, 2007.

[9] Paul Mineiro. fastapprox software library. [Online] Available: https://code.google.com/p/fastapprox/, 2014.

[10] "Netlib Repository". Benchmark programs and reports. [Online] Available: http://www.netlib.org/benchmark/, 2014.

[11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164–174, June 2011.

[12] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys*, 2007.

[13] Lucas Wanner, Salma Elmalaki, Liangzhen Lai, Puneet Gupta, and Mani Srivastava. VarEMU: An emulation testbed for variability-aware software. In *CODES+ISSS*, 2013.

[14] T. Yetiser. Polymorphic viruses: Implementation, detection, and protection. Technical report, VDS Advanced Research Group, 1993.