Published in Intelligent Environments 2013(IE13), Athens, Greece, 18-19th July 2013

Stable Metrics in Amorphous Computing:

An Application to Validate Operation and Monitor Behavior

Malcolm Lear
School of Computer Science & Electronic Engineering
University of Essex
Colchester, UK
malcolm@essex.ac.uk

***Abstract*—A recurring theme in intelligent environments is the intelligent surface composed of nanoscale processing units (smart dust). Such a surface (iSurface) can be considered an amorphous computer composed of a large array of identical processing units (iCells) each with its own sensor/effectors. Whilst nano-sized particles interconnecting in an ad hoc way may just be a dream, there are more practical approaches that could have short term applications in intelligent environments. One such approach is a structured array of iCells constructed at more modest scales perhaps making use of new printing methods onto paper. An important requirement of such a surface is the need for a fast, reliable method to determine iCell operation, performance and code integrity. This paper describes a method to create long (>=32 bit) stable, robust metrics using a profiling technique that represents the current operational state of an iCell and thus enabling the quick exchange of diagnostics between iCells along with data traffic. This paper looks at how stable diagnostic metrics and in particular a metric of code integrity can be created even when external events affect program flow within the iCell. Key requirements in the development of this system were fast acquisition of diagnostic variables, minimal affect on normal operation and the possibility of a hardware implementation which could be completely non intrusive in operation. The described method can create several types of metrics, allowing quick determination of for example, code validation, abnormal operation and unusual behavior.**

Keywords-Amorphous computing; metrics; diagnostics; profiling; non intrusive; embedded systems

I. Introduction

People's living space is becoming rich in electronic devices most of which have computational capabilities undreamt of a few years ago; indeed even humble devices such as the bedroom clock usually utilize embedded systems technology. More often such technology will communicate and interact autonomously as protocols are implemented and refined. Of particular interest in the near future are printable electronics and nanotechnology that open up many new possibilities in pervasive computing. Such technology will enable entire surfaces, for example walls to become "intelligent" and "aware" of the environment. Assuming Moore's law [1] is maintained so that in future a particular computational power requires less physical space and lower power consumption then amorphous computing [2-3-4] could be utilized to create these intelligent surfaces (iSurface). Typical applications would be full wall audio/visual systems [5] offering such things as immersive education, ambiance and artistic interaction. The iSurface with its potentially enormous processing power, resolution, functionality and inherent pervasive properties could well shape the future of intelligent environments.

A practical implementation of the iSurface would most likely be the construction of identical cells (iCells), each with microcontroller (MCU), memory, sensors, effectors and communication hardware. Practical issues with power distribution and communication along with economic issues such as minimizing costs would favor a structured, predefined pattern of iCells as opposed to the ad hoc approaches investigated by A. King [6], W. Butera [7] and others. The structured approach to creating an iSurface also lends itself to practical methods of construction such as printing [8] and stretchy circuits (flexible silicon) [9]. Such a structured approach may seem to be at odds with an undefined amorphic structure, however faults in use or manufacture would require an adaptive interconnection and data flow method entirely compatible with that needed for an ad hoc arrangement of cells. Unlike most "conventional" amorphous computing networks that are preloaded with all required programs, it is intended that the iSurface should be reprogrammable in situ by propagating programs across the surface utilizing the data network. A typical problem with such an approach may well be an error or failure in the reprogramming of particular iCells. The use of metrics to allow adjacent iCells to detect such problems, could for example initiate local iCell to iCell repair by means of program mirroring.

The requirement to identify faulty iCells and abnormal behavior within such an amorphous surface suggested the need to develop a diagnostic system that could run in the background on an iCell, taking little (software based) or no (hardware based) processing time. The idea of producing metrics as an iSurface diagnostic tool addressed several problems, the first of which was the need to propagate the diagnostic data quickly from iCell to iCell. Secondly it would be useful for any diagnostic information to include both behavior and code integrity. The creation of metrics derived from iCell (MCU) profiling seemed to provide a promising solution. Metrics based on profiling have been used to create encryption keys (ICMetrics) [10]. However creating stable metrics derived from standard profiling methods in an embedded system is a challenge due to external events causing

© Lear 2013

the execution of rarely used code. However these issues could be resolved if an alternative profiling method was employed that was inherently unaffected by program flow.

In this paper the merits of deriving diagnostic metrics from a profiling technique that is based on address location of branch op-codes (program structure) have been explored. Program structure based on branch addresses should remain static and unchanged in memory after programming and therefore be an ideal candidate to provide metrics aimed at determining code integrity. Monitoring activity at these same locations in memory would likewise be ideal in creating behavioral and diagnostic metrics. A stable metric of code integrity also puts real meaning into the behavioral and diagnostic metrics, because as with biometrics, it's important to know the animal you are investigating first. Previous work on program analysis based on program branch structure provided more evidence that stable diagnostic metrics could be created [11].

II. Experimental Setup

The experimental setup required to determine the viability and stability of the diagnostic metrics can be broken down into the following areas: (i) the selection of a suitable hardware target platform; (ii) the selection of a suitable software development environment to run on the host PC; (iii) the selection of suitable communication channels that will allow control and acquire data for subsequent analysis on the host PC. The following subsections describe these areas in detail.

A. *Hardware Target Platform*

Important hardware requirements deemed necessary to analyze program structure and extract metrics included the following features: (i) a flexible interrupt controller with timer, thus allowing periodic sampling of processor states (ii) a counter with sufficient resolution to time processor clock cycles. (iii) Joint Test Action Group (JTAG) interface to offer full control including single step operation. (iv) serial port/s offering a channel of communication for control and data acquisition. Previous work and design experience with ARM processor's suggested that the Atmel AT91SAM7S256 [12] would be an ideal choice to fulfill these requirements. Features of the chosen development target board are:

- AT91SAM7S256 microcontroller.
- 64 Kbytes of SRAM.
- 256 Kbytes of FLASH.
- 48 MHz clock (typically 1 instruction per clock cycle).
- 2 serial ports offering up to 115200 baud.
- JTAG interface.

B. *Software Development Environment*

Important features required from the software development environment were: (i) C and assembler language programming. (ii) debug mode utilizing the processors hardware debug module. (iii) hard and soft break points. Previous experience suggested a particular combination of open source software development tools would be ideal. Components used to build the development system include:

- Eclipse [13] is an open source multi language software development environment including an IDE.
- Open On-Chip Debugger (OOCD) [14] is the software interface to the JTAG hardware debugger module.
- GCC C compiler [15].

C. *Control and Data Acquisition*

The choice of hardware platform and software development environment offers two possible modes of communication. First there are the serial ports and secondly the JTAG interface. Serial port communication could use a bespoke protocol preferably but not limited to ASCII characters for control and acquisition of data from the target processor to a host PC or simply be used with a terminal program such as PuTTY. JTAG offers the possibility to control the processor using OOCD commands via a telnet connection. OOCD has a limited but very useful set of commands allowing access to memory, program counter, status and other registers.

III. Program Structure

The most problematic metric to extract is one indicating the integrity of the program code. This metric should ideally be unique to the loaded program whilst remaining stable and unaffected by program flow. A metric based on program structure would fulfill this requirement. Program structure as defined in "SAS-an experimental tool for dynamic program structure acquisition" [11] will be used. In that paper program structure was visualized using 'structure maps'. An example structure map shown in Fig. 1 represents a calibrated portion of the processors memory as a circle with an arrow indicating the normal sequential incrementation of the program counter. Deviations from the circle caused by branches are depicted as lines with green (dashed) indicating a jump forward in memory and red (solid) a jump backwards. The other important visualization is execution frequency (the frequency of address access) being expressed as variation of intensity of the drawn lines. This visualization of program structure and flow will subsequently be used in this and follow up papers as required.

A. *Useful Characteristics for Metrics*

Looking at the structure map in Fig. 1 both fixed and dynamic features can be seen. Fixed features that may be used to extract metrics such as code integrity are branch point source and destination addresses. Dynamic features of the program structure that may be used for behavioral diagnostics are the frequencies of processor activity at those same source and destination addresses. Whilst looking at the structure map it is clear that both source and destination addresses are important fixed features that could be utilized to extract metrics unaffected by program flow, however frequencies at destination addresses can be derived from the branch source and therefore metrics based on frequency analysis need only be concerned with branch point memory locations (branch opcodes).

© Lear 2013

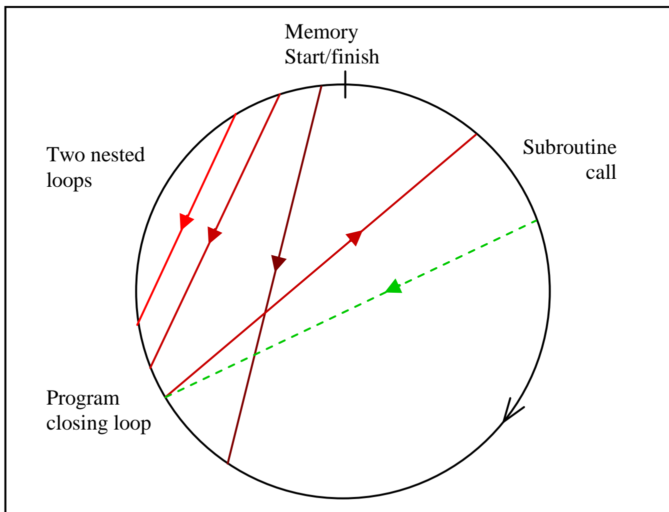

Figure 1. Program structure map

B. Considerations

For experimental purposes, a software based system to determine program structure and acquire test data for further analysis was devised. Various options to accomplish this were considered and evaluated. Simple profiling methods such as periodic sampling could be achieved by using the OOCD commands via the JTAG interface. However the JTAG debug module in most processors, including the AT91SAM7S256 are designed primarily to debug software during the development phase and also programming/verification of the device. The only other dedicated communication channels available were the two serial (UART) ports. Fortunately these ports are reconfigurable, quite fast (115200 baud) and the option of setting interrupts on receive opened up the possibility of an interrupt driven diagnostic and development toolset employing various commands similar to OOCD.

IV. Determining Program Structure

Any method of determining program structure by way of branch address location would need to be dynamic due to legitimate reprogramming of some or all of program memory. This requirement and that the Code Integrity Metric should be stable, led to an approach inspired by Popper's scientific method of falsification [16]. The practical application of this method in determining program structure involves the creation of a metric derived from branch locations in the entire program memory space and then the application of runtime checks to disprove the metric. Once a change in the structure is determined a new metric is created, thus the metric is stable, reflects the current program structure and updates quickly. However, a problem with this approach is that the ideal properties of a program structure metric would preclude the possibility of verifying program structure on-the-fly. For this reason a table dedicated to holding program structure was devised. This program structure table could then be used both for falsification runtime checks and provide the source for the variable length Code Integrity Metric. It was not deemed important that the scan of program memory to create this table should be particularly non-intrusive (a system on chip (SOC) solution would still require memory access) since this would occur only when a change of program structure had been detected (a significant event).

Runtime methods needed to verify executed branch locations (local program structure) against structure information held in the table had to be of a low intrusive nature (ideally non-intrusive if SOC). These requirements led to the idea of verifying the locations of frequently accessed branch addresses in the first instance. Other techniques running at a lower priority could be used to determine the branch locations of rarely executed or dormant code. A Programming Structure Toolkit (PST) was developed utilizing one of the UART ports with the interrupt controller configured to issue a non-maskable interrupt on reception of characters (commands). The associated interrupt routine performs various operations returning information via the UART if required.

A. Program Structure Table

Ideally the program structure table should have the following properties.

- Can be created quickly in software or SOC.
- Resolution to record all branch locations.
- Format that allows fast comparisons to verify program structure.
- Minimal size to maximize program space in a software implementation and lower costs in a SOC implementation.

The first three properties could be met by storing the presence or absence at a memory location by use of a single bit. This method requires no complex calculations to enable reversible cross checking of table entries and branch locations and allows for high speed operation in software or SOC solutions. Using this method the entire program memory of 64Kb would require a table size of 8Kb (8 bits per byte). Although workable it would severely limit the program space or be an expensive SOC solution. However in the case of the AT91SAM7S256 processor all branch instructions are placed on even addresses as are all instructions due to its particular 32 bit architecture. Therefore only one bit of the program structure table is required for every two bytes of program memory thus reducing the table requirement to 4Kb. The possibility to reduce the table size even further by skipping N bytes of the program memory without loosing too much program structure accuracy was also explored.

When looking at assembler code it is quite apparent that branch instructions like most other instructions seem to have a somewhat random distribution. To better understand the effects of further program structure table reduction, analysis of branch distribution was performed. Four basic low complexity software routines were employed to ensure representative and comparable results. More specifically the test programs were based on algorithms from the automotive package from the MiBench suite of benchmark algorithms [17], namely: Angle Conversion, Bit Count, Cubic Functions and Random Numbers. A possible approach to obtaining the branch addresses needed for analysis would be to perform a simple

© Lear 2013

parse of the compiled test programs binaries, noting the addresses of valid branch instructions. However this method would result in many false positives due to data areas being parsed as well. The solution employed was to direct the compiler to produce comprehensive listings that included branch addresses. A program was then written to extract the branch memory locations from these listings and then perform the analysis. This was done for all four programs and the results can be seen in Fig. 2. It can be seen immediately that no branches are closer than 12 addresses apart, so a single bit in the program structure table could represent the presence or absence of a branch for every 12 bytes of program memory without loosing any accuracy. The rather curious similarity between the 4 test programs distribution is due to common library routines used by all 4 programs. This reduction would bring the program structure table size down to 5462 bits or 683 bytes assuming 64Kb of program memory. The sharp rise in the number of branches 16 bytes apart is quite apparent from the graphs. Calculations show that choosing to further reduce the program structure table and increase granularity by assigning 1 bit to 16 memory locations results in an average loss of 10% program structure detail, in other words 10% of the branches in the program memory space would not be included in the program structure table. Further processing of the data used to produce the branch distribution graph (Fig. 2) allowed a graphical look at the relationship between the program structure table size (granularity) and loss of program structure detail (see Fig. 3). This was achieved by plotting the percentage of entries already plotted against the total entries in the data set whilst proceeding from the shortest to the longest branch distribution entries. In this way the graph can show the percentage of program structure detail (closer branch distributions) not represented in the program structure table. The effects of a less than optimal program structure table size of 683 bytes are investigated further later in this paper, however the optimal size of 683 bytes is perfectly acceptable for use as a means to check code integrity of iCells within the iSurface. It should be noted that locating branches for the building of the program structure table can implement the much more simple approach of parsing and checking the entire program memory space for branch instructions since false positives found in data areas will not be encountered and checked for falsification at runtime.

The merits of a second table like the first but being based on branch destination addresses will be evaluated if more program structure detail is deemed necessary.

V. Code Integrity Metric

The code integrity metric is derived from the program structure table and should ideally have the following properties.

- Can be created quickly in software or SOC.
- Retains the uniqueness of the program structure table.
- Variable length (bits).

To maintain the uniqueness of the program structure table it was clear that all bits in the program structure table must in some way be used in the creation of the code integrity metric, i.e., any bit change in the table would result in a change of the metric. Whist there are many possible approaches to meet these goals it was decided to investigate the simplest and most obvious which is to XOR the bits in the program structure table in such a way as to create the new reduced length bit pattern of the code integrity metric. A decision on how best to XOR the bits of the program structure table and maintain the structure information of particular programs was needed since quite similar programs may produce the same code integrity metric. Two simple XOR bits reduction patterns were chosen to be evaluated for the uniqueness quality of various lengths of metric created from the 4 test programs. Tables I and II show these 2 patterns of XOR bit reductions from a small 32 bit (4 byte) program structure table to a 4 bit metric. Whilst these examples are of little practical use due to size, this visualization

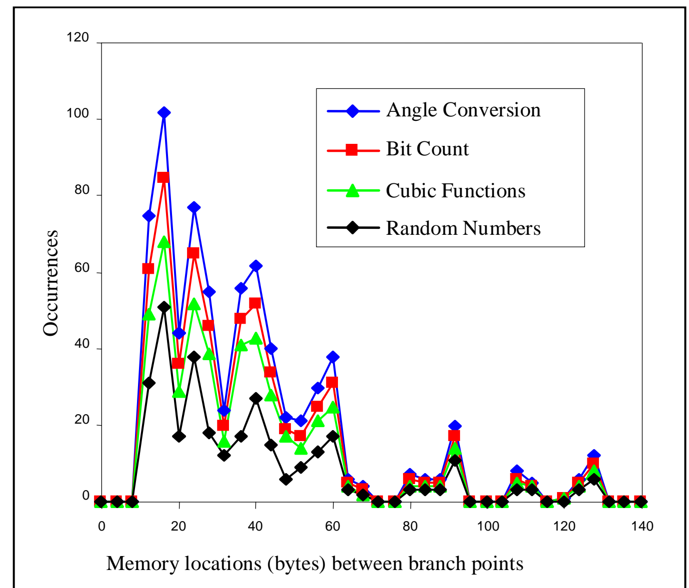

Figure 2. Branch distribution

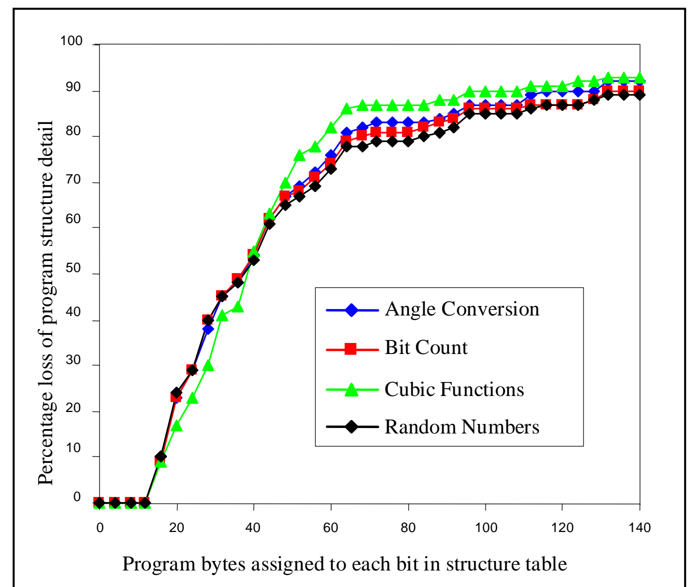

Figure 3. Loss of program structure detail as granularity of program structure table becomes more coarse.

© Lear 2013

may allow us to deduce properties of both before we perform tests.

If we examine the first method shown in Table I, it will be noticed that sequential bits in the program structure table are XORed together to create a single bit of the code integrity metric. This sequence length can be calculated using the following formula:

$$\textit{Sequence Length} = \textit{Table Length} \,/\, \textit{Metric Length}$$

where 'Sequence Length' is the number of sequential bits XORed together, 'Table Length' is the total number of bits in the program structure table, and 'Metric Length' is the bit length of the code integrity metric. This method which we shall call 'Type 1' will reflect the characteristics of the program structure table and hence the position of the branches in the program memory and could therefore be used to identify roughly where in memory the metric doesn't match with the expected one. However a disadvantage is that small local changes that you find with small alterations in code may not show as a change in the code integrity metric. This could be a major issue if the design of the iCell required the independent loading of small programs. The reason being, that a section of program memory could be reprogrammed with legitimate code and that change could be entirely reduced to 1 bit of the code integrity metric. Since that single bit was created by an XOR process, there is a 50:50 chance the variation will not show as a change to the code integrity metric.

The second method shown in Table II, (Type 2) sources the bits required for XORing across the entire program structure table in a stepwise fashion with each step being the length of the code integrity metric in bits. Such a distribution results in the loss of any direct relationship between the program structure and the code integrity metric, however the uniqueness of the code integrity metric with similar programs should be greater than that using the Type 1 XOR pattern. The uniqueness with both XORing patterns will also be related to the length of the code integrity metric, as clearly the fewer bits used and the granularity becomes coarse, there is less opportunity for the code integrity metric to express unique metrics for different programs. With this in mind, tests were performed to determine the uniqueness of the code integrity metric using the 4 test programs with a range of metric lengths with both types of XOR pattern.

A. *Uniqueness of the Code Integrity Metric*

It was possible to extract the uniqueness data entirely on a PC. The 4 test program were compiled using the Eclipse development environment and the binaries intended for loading into the targets (AT91SAM7S256) SRAM were then used as input files to an analysis program developed and running on a PC.

First the SRAM files were scanned for branch locations using simple binary comparisons at each memory location. Then the program structure table was built using the branch location data. The program structure table was 8192 bits (1024 bytes) in length which works out at 1 bit for every 8 bytes of program memory (64Kb). A conservative size for the program structure table was used to maximize structure detail since the object of these tests is centered on the code integrity metric.

With the program structure tables complete, the various code integrity metrics were created using both Type1 and 2 XOR patterns and bit lengths ranging from 4 to 32. With just 4 test programs it seemed unreasonable to go beyond 32 bits unless the results showed otherwise. Fig. 4 and Fig. 5 show results of program uniqueness utilizing XOR patterns Type 1 and 2 respectively. Note that the resulting code integrity metric value was scaled due to the variable bit length. Maximum shown in the graphs represents the maximum numerical value possible for the various code integrity metric bit lengths. Also note that for clarity matching values have been circled.

When viewing the results of the analysis utilizing the Type 1 XOR pattern (Fig. 4) it will be noticed that as expected the ability of the code integrity metric to differentiate between programs suffers to a greater degree as the granularity becomes more coarse compared to that of the Type 2 XOR pattern (Fig. 5). The ability of the code integrity metric based on the Type 2 XOR pattern to differentiate 4 similar programs with a metric length of only 6 bits (64 possible values) seems a good result. The iCells will operate with metric lengths of 128 to 256 bits, so program integrity checks using this system should produce unique metrics for any program.

TABLE I.

32 Bit Table (4 Bytes)																**4 Bit Metric**	
0	^^	1	^^	2	^^	3	^^	4	^^	5	^^	6	^^	7		=	0
8	^^	9	^^	10	^^	11	^^	12	^^	13	^^	14	^^	15		=	1
16	^^	17	^^	18	^^	19	^^	20	^^	21	^^	22	^^	23		=	2
24	^^	25	^^	26	^^	27	^^	28	^^	29	^^	30	^^	31		=	3

TABLE II.

32 Bit Table (4 Bytes)																**4 Bit Metric**	
0	^^	4	^^	8	^^	12	^^	16	^^	20	^^	24	^^	28		=	0
1	^^	5	^^	9	^^	13	^^	17	^^	21	^^	25	^^	29		=	1
2	^^	6	^^	10	^^	14	^^	18	^^	22	^^	26	^^	30		=	2
3	^^	7	^^	11	^^	15	^^	19	^^	23	^^	27	^^	31		=	3

VI. FALSIFICATION OF THE PROGRAM STRUCTURE TABLE

A method of disproof was required that ideally had the following properties.

- Low or no intrusiveness to normal program operation.
- Achievable in software or SOC.
- Targets current program activity.
- Fast operation.

Whilst initial work on the iCell development is based on the AT91SAM7S256 with a communication layer working in a Field-Programmable Gate Array (FPGA), the final design may well be entirely FPGA. With this in mind, a SOC solution with a zero intrusive nature is an attractive proposition. A practical

© Lear 2013

SOC solution is quite straightforward and would rely on direct access to the program counter the data read from memory during normal operation. Fortunately such access is possible in a totally non intrusive way when using soft cores in an FPGA. Operation would involve comparing the data read from memory checking for presence or absence of branch instructions and cross checking with the program structure table. Once a mismatch is detected, the table is disproved and a new scan of program memory is completed to create a new program structure table and code integrity metric. In conclusion, a SOC solution would meet all ideal requirements.

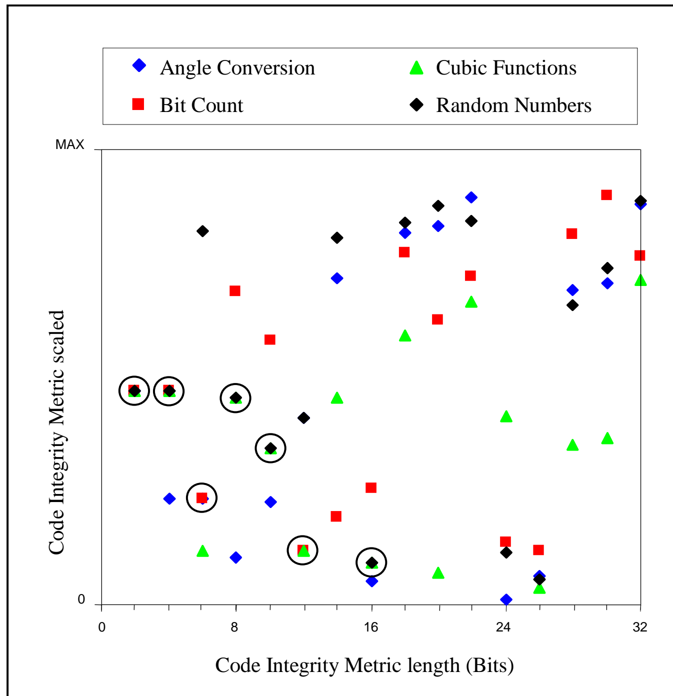

Figure 4. Uniqueness of Code Integrity Metric (Type 1).

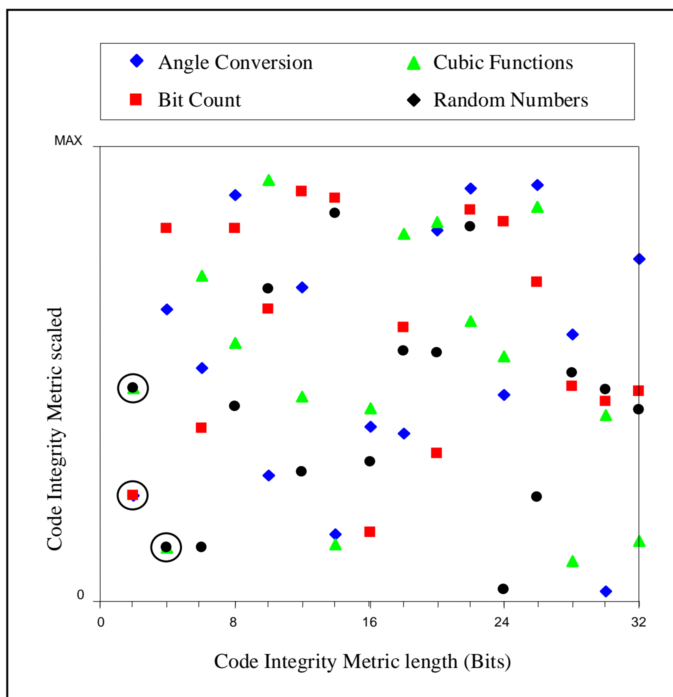

Figure 5. Uniqueness of the Code Integrity Metric (Type 2).

A. *Locating Branch Points in Software*

Meeting the requirements with a software solution is difficult, indeed the very nature of a software solution, i.e., it has to run on the processor, ensures it will in some way be intrusive. However, a variation of the industry standard profiling technique called 'statistical sampling' [18-19] offered a way to approach these ideal requirements without too much compromise. A typical implementation of statistical sampling would periodically halt the processor then return register contents, program counter and stack pointer for further analysis. The technique used for branch location is an extension of this and has the following sequence of operations:

a) Halt the processor.

b) Note the program counter.

c) Search for the next branch in memory following the program counter address.

d) Note the address of located branch.

e) Use the program counter and branch address to cross check the program structure table.

A practical implementation of this technique utilized the MCU's periodic timer to issue system interrupts at a period of 20 milliseconds. The interrupt halts current program execution and retrieves the program counter by way of a modification to the low level interrupt library routine. The program counter is then used as the start point in the search for the next branch in program memory. The typical number of memory locations needed to be read before branch location is 50 to 70 for the 4 test programs resulting in a typical search time of less than 5 microseconds. On branch discovery, a simple routine to disprove the program structure table is executed.

B. *Speed of Falsification*

Initial use of the code integrity metric within the iSurface will be a determination of correctly loaded program code in each iCell. Therefore an experiment to determine average time taken for the system to respond to a change of program in SRAM would provide useful information and help system optimization and future development. Speed of falsification of the program structure table would likely be related to the granularity of the table itself, so this experiment offered the chance to try all permutations of reloading the SRAM with the test programs and varying the table size (bits per program memory bytes). The results of these tests can be seen in Fig. 6 to 9.

Each graph shows the results of the 4 test programs replacing the others in program memory. For example, Fig. 6 shows results of the 'Angle Conversion' program replacing 'Bit Count', 'Cubic Functions', and 'Random Numbers' in program memory. The granularity of the program structure table is shown along the x-axis as program bytes assigned to each bit. The x-axis shows the average (mean) attempts required to determine that the program is not current and has

© Lear 2013

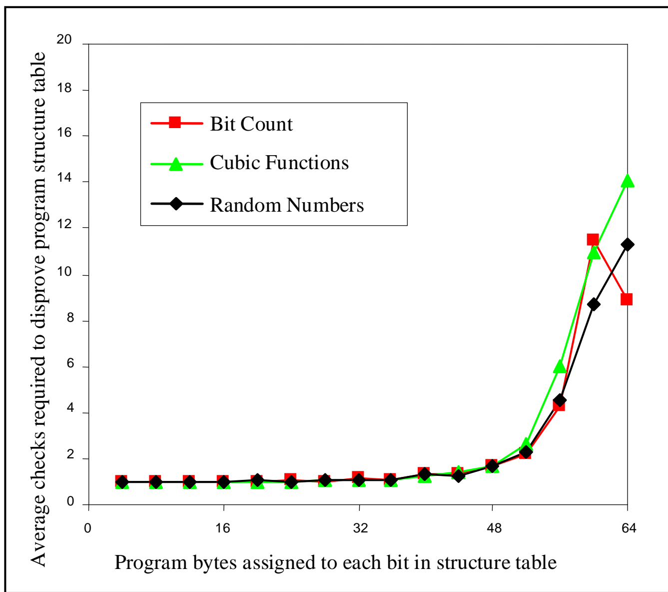

Figure 6. Average code integrity checks required to determine that 'Angle Conversion' has replaced the other 3 test programs in SRAM.

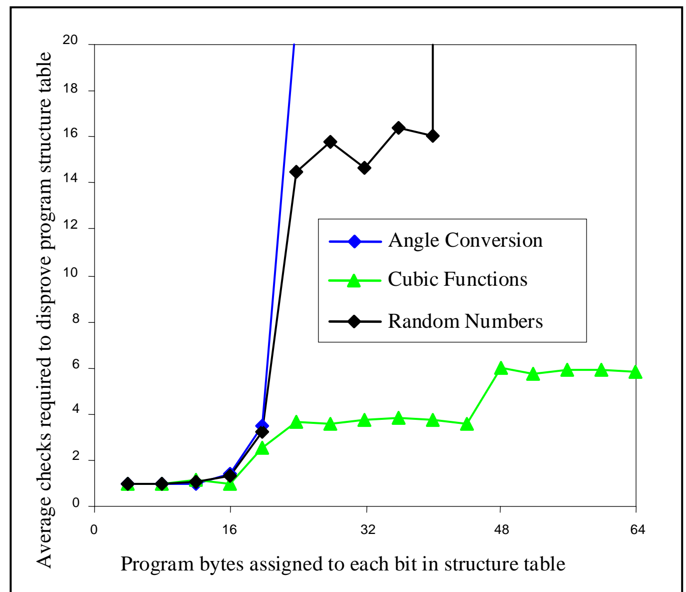

Figure 7. Average code integrity checks required to determine that 'Bit Count' has replaced the other 3 test programs in SRAM.

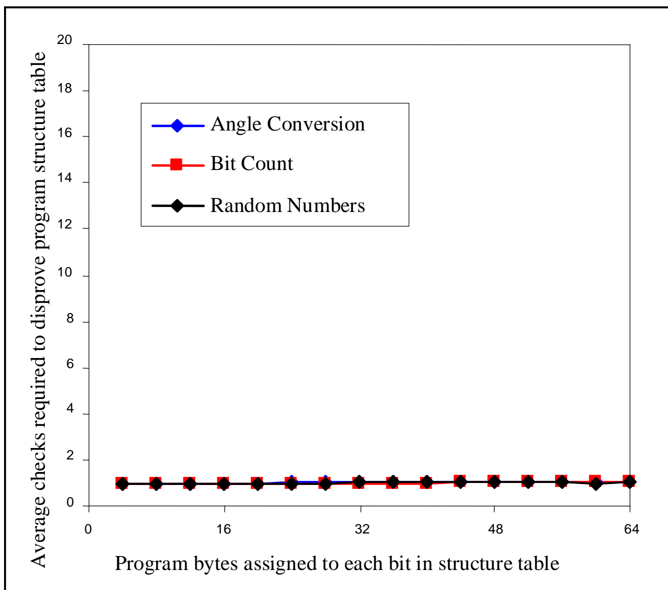

Figure 8. Average code integrity checks required to determine that 'Cubic Functions' has replaced the other 3 test programs in SRAM.

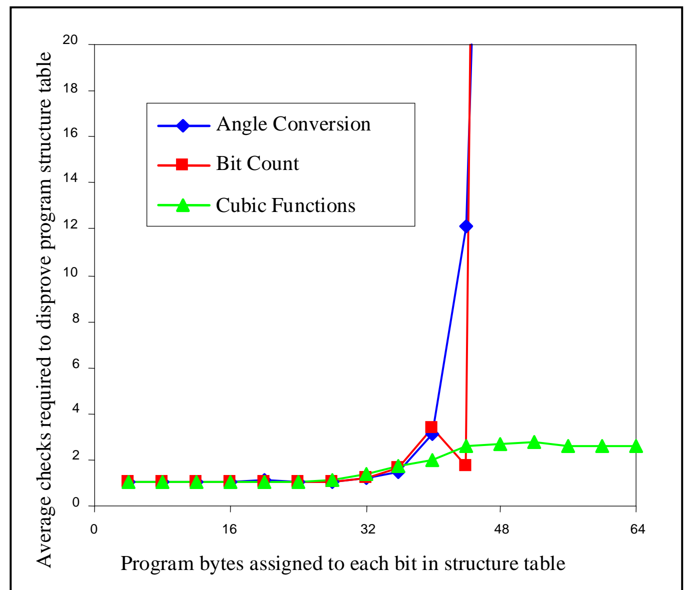

Figure 9. Average code integrity checks required to determine that Random Numbers has replaced the other 3 test programs in SRAM.

been replaced by another in program memory. Average speed of falsification 'time' taken to falsify the program structure table can be calculated by multiplying the average number of attempts by the interrupt timer period. For example, an average number of attempts of 1.5 would take an average time of 30 milliseconds, assuming an interrupt period of 20 milliseconds.

It will be noticed that speed of falsification is more dependent on the replacement programs structure rather than what it replaced which is to be expected since falsification by branch determination will depend on the current structure detail. In particular 'Bit Count' requires more detail (finer granularity) in the program structure table to determine a change of program. This can be explained by the more simple nature of this program. The binary count program is very short (16 branches) and therefore there is less opportunity to locate detail variations from the program structure stored in the program structure table. These results provide more evidence that the program structure table needs the detail afforded by the fine granularity of a program structure table of at least 1 bit for every 16 bytes of program memory. It should also be noted that even large programs contain small routines that could be executed for long periods of time, particularly so in small embedded systems such as the iCell.

This optimal table size of 512 bytes (1 bit for 16 bytes of program locations) for a 64Kb system, works out at 1.28% of the total SRAM space, making a SOC implementation for other

© Lear 2013

uses such as secure communications commercially viable. Another possibility to reduce SOC costs is to use the JTAG interface pins to connect to an external 512 byte SRAM whilst in run mode, thus a simple redesign of the internal JTAG interface logic could provide a code integrity metric without significant cost penalties to the MCU.

VII. Summary

The need in a large amorphous computing array such as the iSurface for quick inter-iCell diagnostics is clear and I argue that this can be fulfilled by the use of stable metrics based on program structure. Further, behavioral metrics can only have any real meaning when a metric based on program code is available. Traditional methods to produce such metrics rely on the idea of accumulating data during run time using various profiling techniques. However, creating something stable derived from the inherently dynamic process of program activity is a serious problem. The idea of turning this process 'on its head' and using similar profiling techniques to disprove a metric created by program structure has been shown to possess the ideal properties of responsiveness to code change, stability and the possibility of a SOC implementation. Further work on behavioral metrics is already underway and will be addressed in a follow-up paper.

Acknowledgment

I would like to acknowledge Dr. Graham Clarke for his uncanny ability to help visualize a problem thus allowing rather elegant solutions to emerge in a relatively short time. In addition I'd like to acknowledge Prof. Vic Callaghan for his support, great enthusiasm and technical advice particularly relating to program structure analysis.

References

[1] D. A. Grier, "the Inovation Curve [Moore's law in semiconductor Industry]," Computer, vol. 39, pp. 8-10, 2006

[2] D. Coore, "Introduction to Amorphous Computing," Lecture Notes in Computer Science, Springer Verlag: Berlin, pp. 99-109, 2005

[3] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, J. Knight, T. F. R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss, "Amorphous Computing," Common. ACM, vol. 43, no. 5, pp. 74-82, 2000

[4] R. Nagpal and M. Mamei, "Engineering amorphous computing systems," In F. Bergenti, M. P. Gleizes, and F. Zambonelli, editors, Methodologies and Software Engineering for Agent Systems, The Agent-Oriented Software Engineering Handbook, Kluwer Academic Publishing, pp. 303-320, 2004

[5] A. King, V. Callaghan, G. Clarke, "Using An Amorphous Computer For Visual Display Applications In Intelligent Environments," IET 4th International Conference on Intelligent Environments, 2008.

[6] A. King, "Deus Ex Machina: Engineering Emergence in an Amorphous Computer," International Workshop on Intelligent Environments, 2005.

[7] W. Butera, "Programming a Paintable Computer," PhD Thesis, MIT Media Lab, 2001

[8] Chang, J.; Tong Ge; Sanchez-Sinencio, E., "Challenges of printed electronics on flexible substrates," Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on , vol., no., pp.582,585, 5-8 Aug. 2012

[9] D. H. Kim, et al, "Epidermal Electronics," Science, vol 333, pp. 838-843, Aug 2012

[10] Y. Kovalchuk, G. Howells, and K. D. McDonald-Maier, "Overview of ICMetric Technology – Security Infrastructure for Autonomous and Intelligent Healthcare System," International Journal of u- and e-Service, Science and Technology, vol. 4, no. 3, pp. 49-60, Sep 2011

[11] V. Callaghan and K. Barker, "SAS-an experimental tool for dynamic program structure acquisition," Journal of Microcomputer Applications, vol. 5, pp. 209–223, 1982

[12] Atmel's AT91SAM7Sxxx Datasheet: http://www.atmel.com/Images/doc6175.pdf

[13] Eclipse Official Website: http://www.eclipse.org

[14] Online OpenOCD User's Guide: http://openocd.sourceforge.net/doc/html/index.html#

[15] GCC Official Website: http://gcc.gnu.org/

[16] K. R. Popper, "Science as Falsification," Conjectures and Refutations, Routledge and Keagan Paul: London, pp. 30-39, 1963

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," Proceedings of the International Workshop on Workload Characterization, pp. 3-14, 2001

[18] M. A. Jennifer, et al, "Continuous Profiling: Where Have All the Cycles Gone?," ACM, pp. 357-390, 1997

[19] J. Whaley, "A portable sampling-based profiler for Java virtual machines.," In *ACM 2000 Java Grande Conference*, June 2000

© Lear 2013