Unobtrusive Software and System Health Management with R2U2 on a parallel MIMD Coprocessor

Johann Schumann[1], Patrick Moosbrugger[2]

[1] *Stinger Ghaffarian Technologies, Inc., NASA Ames Research Center, Moffett Field, CA, 94040, USA*

johann.m.schumann@nasa.gov

[2] *Vienna University of Technology, Treitlstrasse 3, 1040 Vienna, Austria*

moosbrugger@cps.tuwien.ac.at

Abstract

Dynamic monitoring of software and system health of a complex cyber-physical system requires observers that continuously monitor variables of the embedded software in order to detect anomalies and reason about their root causes. There exists a variety of techniques for code instrumentation, but instrumentation might change runtime behavior and could require costly software re-certification.

In this paper, we present R2U2/E, a novel realization of our real-time, Realizable, Responsive, and Unobtrusive Unit (R2U2). The R2U2/E observers are executed in parallel on a dedicated 16 or 64 core EPIPHANY co-processor, thereby avoiding additional computational overhead to the system under observation. A DMA-based shared memory access architecture allows R2U2/E to operate without any code instrumentation or program interference.

1. Introduction

Modern cyber-physical systems, like unmanned aircraft (UAS), autonomous vehicles, or space systems are equipped with numerous sensors that make it possible for the system to perceive its environment and enable accurate guidance, navigation, and control. Measurements of these sensors need to be processed in real-time by a software system of considerable size. Functions for advanced autonomous operations, decision making, and planning substantially add to the complexity of the software. This software, which needs to be executed on board of the aircraft obviously is highly safety-critical: failures can cause not only loss of the vehicle and an unsuccessful end of the mission, but also might harm human life. Therefore, such software must undergo rigorous certification.

Nevertheless, such cyber-physical systems can fail due to a multitude of reasons. Faulty sensors, unreliable communication between sensor subsystem and flight computer, errors in the design or implementation of the control system or control logic can cause problems that could lead to total loss of the mission or destruction of the vehicle.

If, however, off-nominal situations can be detected reliably, on-board and in real-time, then mitigation actions can be triggered, or dangerous actions avoided in the first place.

For example, the European Mars lander, Schiaparelli encountered several errors during descent and landing, which caused the probe to shut down its retro rockets at an altitude of several kilometers, causing it to crash into the planet with a velocity of about 150m/s. A detailed incident report (Tolker-Nielsen, 2017) found out that saturated values of the inertial measurement units were handled wrongly by the guidance and navigation software — it suddenly estimated a *negative* altitude. Based upon those wrong estimates, the weak decision logic assumed that landing had taken place, turned off the rockets, and activated post-landing procedures. However, the probe was still about 3.7km above the ground. Redundant instruments, like a Radar Doppler Altimeter (RDA), had produced correct measurements, but they had been ignored. The incident report therefore recommends, among others (recommendation 05, (Tolker-Nielsen, 2017)):

Robust and reliable sanity checks shall be implemented in the on-board S/W to increase the robustness of the design, which could be, but not limited to:

- *Check on attitude*
- *Check on altitude sign (altitude cannot be negative).*
- *Check on vertical acceleration during terminal descent and landing (cannot be higher than gravity).*
- *Check altitude magnitude change (it cannot change from 3.7 Km to a negative value in one second).*
- *Check wrt pre-flight timeline (altitude or acceleration profile vs time) to check consistency of measurements*

Our system R2U2 (real-time, Realizable, Responsive, and Unobtrusive Unit) has been designed to carry out checks about safety and performance properties, sensor and software consistency, as well as performing diagnostic reasoning and prognostics *in real time* while the system is in operation. R2U2 combines signal processing, Metric Temporal Logic, Bayesian Networks, and model-based prognostics to enable the system designers to develop powerful and expressive models. Checks, like those mentioned above could be easily modeled within R2U2. A check against the pre-flight timeline could look like

$$\text{RDA-switch-on} \rightarrow \Diamond_{[25,35]}\text{Backshell-separation}$$

After the RDA has been switched on, the backshell needs to separate within the next 25 to 35 seconds (cf. (Tolker-Nielsen, 2017), Fig. 2). R2U2 can perform numerous checks and perform diagnostic reasoning for root cause analysis in case of an anomaly.

However, this checking has to come with a prize: usually, run-time monitoring is done using software observers that are brought into the code by instrumentation of the flight software. Although a large number of approaches exist, there are two severe drawbacks: additional burden on the CPU load of the flight computer and software safety/certification issues.

It is obvious, that any additional code that must be executed within the inner control-loop of a safety-critical system can alter its runtime behavior. That may lead to violation of real-time constraints, time overruns, or missed cycles. In particular, on small and weak flight computers as are often used on drones or small spacecraft, that can lead to severe misbehavior of the software and subsequent system crashes. There are stories that small drones of a student project crashed during flight, because the students simply added some printf statements to log additional variables (pers. comm). So, a successful monitoring must be *unobtrusive* in the sense that it should not change the temporal behavior of the software or change the CPU burden.

Flight software, as mentioned earlier, is highly safety-critical. Therefore, regulations and standards require that this software is certified according prescribed, published standards. For example, DO-178C (RTCA, 2012) defines software development standards for safety-critical software in commercial transport aircraft. Certification is a highly complex, costly and time-consuming process, where the software, after passing is not allowed to be modified again. So, the addition of run-time monitors to the software would require a full software re-certification, something that is, in most cases, out of the question.

In this paper, we present R2U2/E, a realization of R2U2, that combines high performance, low power requirements with an extremely high level of unobtrusiveness. Previous R2U2 versions (Geist, Rozier, & Schumann, 2014; Reinbacher, Rozier, & Schumann, 2014; Schumann, Rozier, et al., 2015) have been developed on a dedicated FPGA chip in order to address the above-mentioned challenges. The new monitoring and reasoning engines of R2U2/E are executed on a modern EPIPHANY chip (Olofsson, Nordström, & Zain-ul-Abdin, 2014), a high-performance, energy-efficient MIMD architecture with an efficient 2D mesh Network-on-Chip and a distributed shared memory model. Designed as a powerful co-processor for numeric computation in real-time embedded systems, this architecture combines low power requirements with a high degree of parallel execution.

We are using the EPIPHANY chip as a co-processor to a Zynq 7000 Series SoC system, that is running a Linux operating system with the flight software. A built-in frame-buffer architecture allows the EPIPHANY chip to access certain memory locations of the main memory using direct memory access (DMA).

R2U2/E is running in parallel on the EPIPHANY chip and fetches the values of variables to be monitored from the main processor's memory. Using this novel architecture, we can monitor sensor variables and the flight software without having to instrument or modify the software.

Our approach addresses the two main challenges for instrumentation, (Watterson & Heffernan, 2007): (1) probes must be capable of observing enough relevant information in order to determine the system's state, and (2) the behavior of the observed system must not be affected. In the application domain of safety-critical systems (e.g., an autopilot), critical data are usually placed in static (dedicated) memory locations. Hence, we have a prior knowledge or can determine the memory location of relevant data for monitoring on startup of the system. Besides, the memory access by means of a dedicated DMA channel, which is executed by a dedicated processing unit operates not only independently, but does not cause additional computational overhead on the observed system or alter its behavior.

The main contributions of this paper are:

- development of an unobtrusive Co-processor based monitoring architecture for R2U2/E. The flight software is running on the main processor and can be monitored in real-time without having to instrument it,

- development of a parallel execution architecture for R2U2/E, which uses the available processors to execute the various R2U2 components in parallel, and

- performance and power consumption analysis of R2U2/E running on a Parallella board (Adapteva, 2017) with a 16 and 64 core EPIPHANY co-processor.

The rest of this paper is structured as follows: In Section 2 we discuss related work. Section 3 will give a short overview of R2U2 and its implementation variants, as well as a short de-

scription of the EPIPHANY multiprocessor and the Parallella board. Section 4 discusses details of the parallel architecture and the unobtrusive access techniques used by R2U2/E. In Section 5, we show results of experiments on measuring the performance of R2U2/E on the Parallella board. Section 6 concludes and discusses future work.

2. RELATED WORK

Tsai, Fang, Chen, and Bi (1990) and Watterson and Heffernan (2007) discuss non-intrusive monitoring. A dedicated monitoring system taps into the communication bus between the processor and the memory. This requires that each memory access (also non-related) is processed immediately, which forces the monitoring system to operate event triggered at a high rate. Reinbacher, Függer, and Brauer (2013), Heffernan, Macnamee, and Fogarty (2014), and BusMOP (Pellizzoni, Meredith, Caccamo, & Rosu, 2008; Meredith, Jin, Griffith, Chen, & Roşu, 2012) follow a similar non-intrusive approach to tap into a communication bus and use an FPGA implementation in order to achieve the required performance. R2U2/E, on the other hand, does not need to monitor the complete bus-traffic but can access an arbitrary memory location at a specific time since it uses a dedicated DMA channel. Therefore, it can operate independently of the system under observation.

Reinbacher, Brauer, Horauer, Steininger, and Kowalewski (2014); Reinbacher, Geist, Moosbrugger, Horauer, and Steininger (2012) use a parallel FPGA design for creating runtime monitors. Similar to R2U2/E, Berkovich, Bonakdarpour, and Fischmeister (2015) use a dedicated multicore architecture for optimizing the performance of runtime monitors. Their monitors run on a dedicated GPU in order to minimize the overhead on the system under observation. They generate an instrumented C program from a specification, whereas R2U2/E follows a non-intrusive approach. Typical GPUs have a high performance, but with a power consumption that can be orders of magnitude higher than the EPIPHANY chip, making that approach less suitable for low power embedded applications.

3. BACKGROUND

3.1. R2U2

R2U2 has been designed as tool for continuous monitoring and system/software health management. Properties can be specified in past-time and future-time Metric Temporal Logic (MTL) as well as "mission time" Linear Temporal Logic. Bayesian Networks can be used to perform probabilistic diagnostic reasoning and root cause analysis. The R2U2 engine receives a vector of sensor signals and values of variables at a certain rate (Figure 1A). The atomic propositional checking (AT) unit performs various forms of signal processing, filtering, and component prognostics (Schumann, Roychoudhury, & Kulkarni, 2015) before the values are discretized.

The temporal logic processing units (TL) are implemented as special-purpose processors and use advanced monitoring algorithms using storage queues (Reinbacher, Rozier, & Schumann, 2014), which requires only a small memory footprint and enables fast processing. Outputs of TL can be fed into the Bayesian Network (BN) execution unit, which calculates posterior probabilities of components and failure modes. BN uses an efficient representation of the Bayesian Network as an Arithmetic Circuit (?, ?; Schumann, Rozier, et al., 2015).
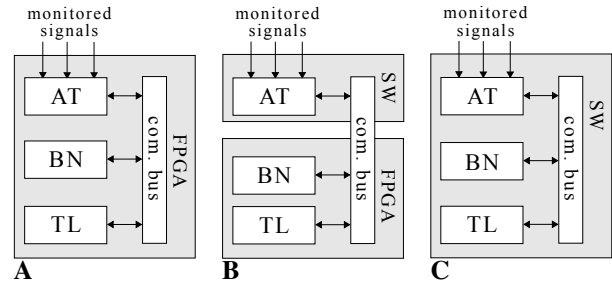

Figure 1. R2U2 Versions

R2U2 has been developed in three main versions: the FPGA standalone version (Reinbacher, Rozier, & Schumann, 2014) (Figure 1A) was developed for high performance applications and resilience against tampering. Its AT, BN, and TL components communicate via an internal communication bus. Each of these building blocks can be instantiated as an arbitrary number of parallel copies, depending on the required performance and available FPGA resources. The R2U2 FPGA version has been used on different case studies, as for example, on the NASA Swift UAS, where it was able to detect a failing altimeter, a wrong configured magnetometer, or pitch oscillation induced by faults originating from the file system (Schumann et al., 2013; Geist et al., 2014).

The hybrid version of R2U2 instantiates the AT and BN building blocks either as software components, or as part of the FPGA design (Figure 1B). To that end, we utilize a Xilinx Zynq SoC-FPGA chip which enables us to execute software on the embedded ARM CPU, instantiate the FPGA building blocks in the programmable logic section, and use chip-internal communication interfaces between these components. This version benefits from the flexibility of software development and allows to steer the tradeoff between performance and resource consumption. The hybrid version of R2U2 has been used on the NASA Dragoneye UAS to monitor and diagnose safety and security threats as, for example, GPS Spoofing, or malicious attacks through attack patterns (Moosbrugger, Rozier, & Schumann, 2017). It also has been used for on-board battery prognostics (Schumann, Roychoudhury, & Kulkarni, 2015).

The R2U2 software-only version (Figure 1C) has been developed to run exclusively as a software component, opening up new application areas. This R2U2 variant has be instantiated

as a software "app" of the NASA Autonomy Operating System (AOS) (Lowry, Rayadurgam, Schumann, Pressburger, & Dalal, 2017) to provide monitoring and diagnostic capabilities for autonomous UAS operations. Our different R2U2 versions use the same modeling tool-chain in order to facilitate model interchange and reuse.

3.2. EPIPHANY and Parallella

The Parallella board is a credit-card sized high performance computer featuring a dual-core ARM A9 processor and a 16 or 64 core EPIPHANY co-processor. This co-processor consists of a scalable array of simple RISC processors with a fast floating-point arithmetic unit. This MIMD (Multiple Instruction, Multiple Data) mesh of independent cores is connected together with a fast on-chip network within a distributed shared memory architecture (Olofsson et al., 2014). A low-level library provides access mechanisms to the on-chip distributed memory and the memory of the ARM processor, Mutexes and other synchronization mechanisms, as well as utility functions to load and control each of the cores. Each core has two dedicated DMA channels for high-speed data transfer between the different on- and off-chip memory regions.

According to the EPIPHANY datasheet (Adapteva, 2014) the processor is tailored to low power applications and capable to achieve 102 GFLOPS peak performance while consuming less than 2 Watts.

4. PARALLEL ARCHITECTURE OF R2U2/E

The R2U2/E architecture has to meet two important goals: performance gain of the R2U2 monitoring and reasoning components by exploiting the multicore architecture of the EPIPHANY chip, and a highly unobtrusive access to the variables of the system under observation (SuO) running on the main processor of the Parallella board.
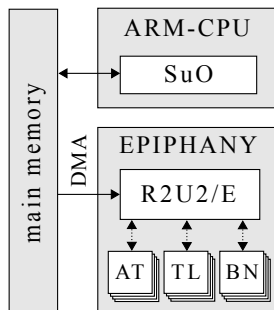
4.1. Architecture

Figure 2. Architecture of R2U2/E on Parallella

Figure 2 shows an overview of the architecture of R2U2/E. The system under observation (SuO), e.g., the autopilot soft-

ware, is executed on the ARM CPU of the Parallella board as a Linux process and can communicate with external subsystems and sensors (not shown here). The SuO is configured in such a way, that global variables of interest are located in a specific region of the memory, which can be accessed by R2U2 running on the EPIPHANY chip. A hardware-based DMA (direct memory access) mechanism allows the EPIPHANY hardware interface to copy data from specific parts of the DRAM into the local memory of the EPIPHANY chip. This low-level access and synchronization is implemented in hardware in such a way that the Linux process is oblivious to this access and no process- or context switches or interrupts need to occur. With this architecture, the SuO need not be instrumented for monitoring, and its execution is not unduly affected by the monitoring.

R2U2/E itself resides on the cores of the EPIPHANY chip. An additional program, running as a Linux process on the ARM CPU is in charge of initializing the cores, coordinating the data transfer, controlling the time stamps, or to start mitigation actions. For clarity, Figure 2 does not show this component. The R2U2/E master on one of the cores is in charge of controlling the data transfer and setting up for the worker cores that contain distributed versions R2U2 components. In the following, we will present the detailed architecture and data flow on the EPIPHANY chip as well as the access mechanisms to the memory of the ARM CPU.

4.2. Parallel Execution

The execution model of R2U2/E attempts to obtain speed-ups with a two-level parallelization scheme: (a) execute the R2U2 components (AT, BN, TL) in parallel to the master M that is handling the data transfer, and (b) execute the AT and TL component in parallel by splitting up the R2U2 model. Each of the R2U2 components can be seen a functional block that take an input vector I_t at time t and produces an output vector O_{t+1}. Signal values S_t comprise the input to AT.

In our model, we run all worker processes in a synchronous loop, as shown in Algorithm 1, synchronized by barriers. By using two sets of input and output vectors, the individual components of R2U2/E TL can be executed in parallel on different cores. Because the workers need to have access to the previous values of inputs and outputs, we use buffers B^I and B^O that point to the respective vectors in order to avoid unnecessary copies. These buffers and the vectors are located in the EPIPHANY RAM that can be accessed by all cores.

For the inner level of parallelism, we employ the fact that an R2U2 model is highly modular: a model consists of a number of filters that work on the input signals, which can be Booleans, integers, floating point numbers, but also strings or complex data structures. Each filter produces, for each time stamp t, a single element of the input vector I. Therefore, we can, without any additional need for synchronization, run

4

Algorithm 1 R2U2/E. Initially: $v = \text{true}, t = 0$

$B^I \leftarrow [I_1, \ldots, I_4]; B^O \leftarrow [O_1, \ldots, O_4]; B^S \leftarrow [S_1, S_2]$
while true **do**
    in parallel {
    M :
    if v **then**
        swap$(B_1^I, B_2^I); B_1^S \leftarrow$ DMA;
        R2U2 $\leftarrow B_3^O$; swap(B_2^O, B_3^O);
    else
        swap$(B_3^I, B_4^I); B_2^S \leftarrow$ DMA;
        R2U2 $\leftarrow B_1^O$; swap(B_4^O, B_1^O);
    end if
    AT^i :
    if v **then**
        $B_1^I \leftarrow$ AT(B_2^S);
    else
        $B_3^I \leftarrow$ AT(B_1^S);
    end if
    TL^i :
    if v **then**
        $B_1^O \leftarrow$ TL(B_3^I, B_4^I, B_4^O)
    else
        $B_3^O \leftarrow$ TL(B_1^I, B_2^I, B_2^O)
    end if
    BN^i :
    if v **then**
        $B_1^O \leftarrow$ BN(B_3^I)
    else
        $B_3^O \leftarrow$ BN(B_1^I)
    end if
    } barrier
    $t \leftarrow t + 1; v \leftarrow \bar{v}$
end while

---

different AT filters on different cores. In a similar way, the R2U2 model consists of numerous individual temporal formulas that can be executed independently on different processors. Figure 3 shows the situation, where the TL engines have access to all elements of the current input vector B^I, but only write to disjoint subsets of the output vector B^O.
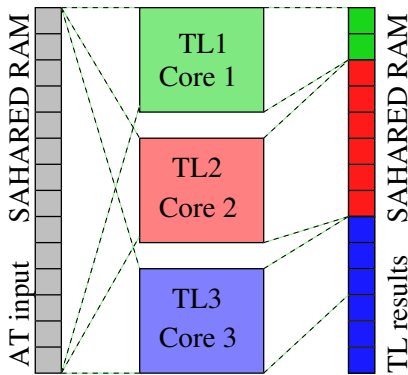

Figure 3. Memory access for R2U2/E cores. Input and output vectors used shared memory on the EPIPHANY chip

Here, our parallel execution model builds upon the memory-mesh architecture of EPIPHANY, which allows multiple

cores to read the same memory cell. Unprotected writes are only possible by one processor. The mapping of the elements of the R2U2 model will be described below. In case, AT must access results of TL or BN, as, for example, to discretize a posterior probability, those results are fed back to the input vector and will be processed at the next time stamp as is done in all R2U2 implementations.

4.3. Load Balancing

The maximum update rate R_{max} of R2U2/E is governed by the maximum of the execution times, the master and each of the workers need for one update step. It is therefore essential to break down the individual components of the R2U2 model into subsets that roughly exhibit the same execution time.

Each AT filter, the execution of a Bayesian network, and the temporal logic operators have bounded, statically determinable execution times (for proofs see (Reinbacher, Rozier, & Schumann, 2014)), We are currently using a simple static allocation method that is based upon averages of measured execution times for the individual formula components. The actual time-stamp rate R_{R2U2} must be set to a value smaller than R_{max} in order to avoid timing skews.
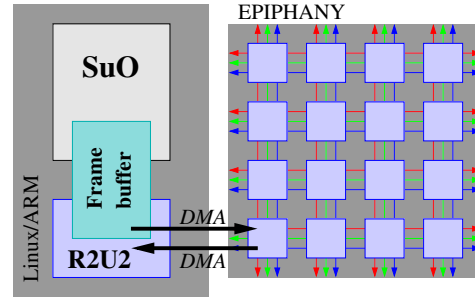

Figure 4. Architecture of R2U2/E and SuO

4.4. Access to Software under Observation

The software to be monitored by R2U2, the software under observation (SuO) is executed as a regular process on the Linux system; global variables that are of interest for R2U2 are located in the system's RAM memory. Ideally, R2U2 only needs to get to know the physical memory addresses of those global variables and then could pull their values using a direct memory access (DMA) channel. This unprotected read-only access is possible on the Parallella board. However, it would require low-level code modifications.

For this paper, we therefore use a different approach (Figure 4): the Parallella board has a built-in frame buffer, that is usually used to drive the video display. Technically, this is nothing more than a region of shared memory that can be accessed by other processes or the EPIPHANY cores. Since R2U2 is only doing read accesses to this memory no addi-

tional access protection is needed. The loader of the SuO is directed to place all global variables into the frame buffer. No instrumentation or other modification of the code is required.

The R2U2 process on the host is in charge of pulling the global variables from the frame buffer into a shared buffer that can be accessed by the EPIPHANY cores in regular intervals. Its main loop then notifies the R2U2 master on the EPIPHANY that new data are available and that processing by the EPIPHANY worker cores should start. After parallel execution, results are read back into the host memory and logged or used to trigger mitigation actions. Figure 4 shows the various components of R2U2 and the interaction between their memory regions.

5. EXPERIMENTS AND RESULTS

In this section, we present results on experiments to evaluate the performance of the R2U2/E architecture. We have instantiated this architecture for a 16 core EPIPHANY chip mounted on a Parallella board. The SuO and the host R2U2 components are running as individual Linux processes. For this paper, we restrict ourselves to only analyze the run-time behavior of the temporal processing unit (TL).

In a first experiment, we evaluated the overhead the SuO has to face, when it is writing variables into the framebuffer. For an inner-loop access, the time to write a variable was about 7.5ns and did not change noticeable from the time needed to write to the non-shared global memory. The time also did not change measurably, when R2U2 was accessing the shared frame buffer. This indicates that monitoring via R2U2/E can be done in a very unobtrusive manner.

We then determined the execution times for a single update of the R2U2 master running on the EPIPHANY chip. Table 1 shows that it takes considerable time on the host component to package the data, start the master, which in turn fetches the data and sends back results. As expected, most of the time is spent copying the input and result data between the processors. A single update cycle of the R2U2 master takes about $13\mu s$ without data transfer. We also observed that a high rate, with which the SuO writes into the frame buffer can slow down the R2U2/E cycle rate. Note that here, we are measuring means of minimal execution times for R2U2 operations. In a regular application, R2U2 would be executed with a fixed update rate R, which must be larger than our experimentally obtained maximum update rate.

In order to obtain execution times for each TL operator, we use typical formulas from prior case studies, (Moosbrugger et al., 2017; Schumann, Moosbrugger, & Rozier, 2016; Geist et al., 2014; Schumann, Roychoudhury, & Kulkarni, 2015) or artificial formulas running on regular input traces.

Table 1 shows mean execution times for Boolean operators $(\vee, \wedge, \rightarrow, \neg)$, the $\circ$ (previously) operator, as well as MTL

operators with time points (e.g., $\square_{[10]}p$) and intervals (e.g., $\square_{[5,10]}p$). As a comparison, we measured the run-times of the software version of R2U2 as called from the octave[1] system. Here again, we ignore the interface overhead to octave. In a sequential mode, the single operators take about twice the time on the EPIPHANY chip. However, as soon as more than 2 workers are operating at full speed, R2U2/E will be faster than R2U2 running on the ARM CPU. Theoretically, a maximum speed-up of about 7 over the ARM execution should be achievable with a 16 core EPIPHANY chip. On a 64 core chip, a considerably higher speed-up of about 30 should be possible.

Table 1. Basic performance for R2U2/E ([†]) and R2U2 running on ARM A9 ([‡])

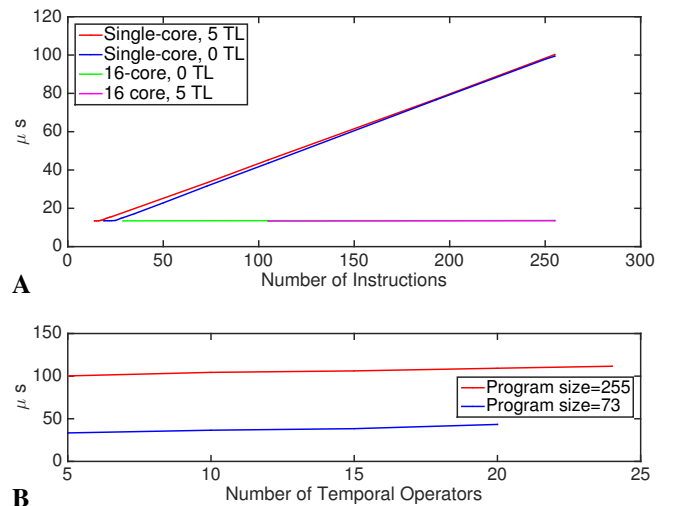| Operation	t[us][†]	t[us][‡]
Master (M)	13.1	NA
M with transfer	28.2	NA
M with high bandwidth transfer	38.4	NA
Boolean	0.23	0.12
Temporal (time point)	0.4	0.2
Temporal (interval)	0.8	0.45

Figure 5. Performance of R2U2/E

The overall calculation time for our models depends on the number as well as the types of TL operators in the formula. Figure 5A shows the mean execution time of two different formulas being executed. The number of operators in the formula corresponds to the number of instructions of the TL processor. Each of them have been executed on a single core, as well as on our 16 core implementation. Times have been measured without data transfer. The red and the magenta traces

---

[1]http://octave.org

6

are formulas that contain 5 temporal logic operators with timing constraints, whereas the blue and the green trace do only contain only temporal logic operators without intervals.

As Figure 5A shows, the execution time grows linearly with the size of formula, as long as the type of the TL operators does not change. Figure 5B indicates how an increasing number of temporal operators lead to slightly increased (sequential) execution times. The relatively low number of temporal operators compared to Boolean operators is indicative of most of our R2U2 models.

Figure 5A also shows how the non-trivial time consumed for synchronization and data transfer influences the overall system behavior. Even with a formula length of 255 operators (currently the maximum for this configuration) and optimal load balancing, the execution time on 16 cores is still dominated by the synchronization overhead of $13.1\mu s$.
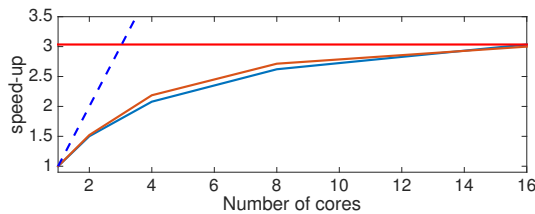

Figure 6. Speed-ups for R2U2 model with 198 Boolean and 36 temporal operators

The substantial data transfer and synchronization overhead reduces the achievable speed-up for smaller R2U2 models. Figure 6 shows the speed-up curves for an existing R2U2 model. Brown and blue lines show the measured speed-up values against running R2U2/E with one TL core for a random mapping of formulas to the cores (blue) and for a near-optimal mapping (brown). The execution times of the host and master component of R2U2/E limit the maximal speed up to approximately 3 for this formula (red line in Figure 6), which is achieved when a larger number of cores are active. The dashed blue line corresponds to the maximal speed-up ignoring communication and synchronization overhead. It shows that for this R2U2 formula only 3 cores could be kept busy in a meaningful manner. This in an indication that the R2U2/E architecture is most beneficial for large and complex sets of temporal formulas.

For the R2U2/E implementation, we made initial measurements on the overall power consumption. Execution of the TL engine on the ARM CPU (ignoring all data transfer) uses approximately 0.2W. 15 workers on the EPIPHANY chip take 0.35W of power (again ignoring all data transfer), yielding a reduction in power usage by a factor of approximately 8. The overall R2U2/E architecture on the host side and on the EPIPHANY chip uses approximately 1.5W. Since we could not measure the power consumption of the individual compo-

nents of the Parallella board separately, these numbers should be regarded with caution.

6. CONCLUSIONS

In this paper, we presented R2U2/E, a parallel and unobtrusive architecture of the R2U2 runtime monitoring system, which is running on a parallel EPIPHANY chip with 16 or 64 cores. The execution of R2U2 on a co-processor relieves the flight computer from computational burdens for monitoring and makes R2U2 substantially more unobtrusive. With a DMA-based architecture for the access of variables of interest, the flight software does not need to be instrumented or otherwise modified and can access these data without changing the behavior of the flight software in a noticeable manner.

On the EPIPHANY chip, a master processor handles synchronization and data transport and is in charge of controlling all workers that work in lock-step on the given input data. Independent subformulas can be executed in parallel on different processors, providing a good potential for high speed ups for most R2U2 models.

Although R2U2/E can achieve timestamp rates for the temporal logic engine of about 20kHz, there is much room for improvement of this architecture. In particular, the transfer of the data from SuO via the host-side R2U2 currently is the main bottleneck prohibiting higher update rates. Direct memory access of the R2U2 master on the shared buffer should solve this issue. Furthermore, R2U2/E on the EPIPHANY will be extended to handle control and should only notify the host-side of R2U2 in case properties are violated, which is expected to happen only at a low rate.

The current implementation of R2U2/E replicates the formulas over all cores, thus wasting precious local memory. Small changes to the R2U2 modeling and compilation tool chain will allow us to monitor more and larger properties, taking advantage of the inherent speedup. Finally, we are planning to investigate how to specify R2U2 models, which contain components with bounded, but high computational requirements, e.g., large Bayesian networks or model-based particle filters and how to efficiently map them to our R2U2/E architecture.

ACKNOWLEDGMENTS

Work supported in part by NASA Autonomy Operating System (AOS) project NASA NNX-14AN61A.

REFERENCES

Adapteva. (2014). E64G401 Epiphany 64-core Microprocessor Datasheet Retrieved from http://www.adapteva.com/docs/e64g401_datasheet.pdf

Adapteva. (2017). *The Parallella board.* Retrieved from

https://www.parallella.org/board

Berkovich, S., Bonakdarpour, B., & Fischmeister, S. (2015). Runtime verification with minimal intrusion through parallelism. *Formal Methods in System Design*, *46*(3), 317–348.

Geist, J., Rozier, K. Y., & Schumann, J. (2014). Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems. In *RV14* (pp. 215–230).

Heffernan, D., Macnamee, C., & Fogarty, P. (2014). Runtime verification monitoring for automotive embedded systems using the ISO 26262 functional safety standard as a guide for the definition of the monitored properties. *IET Software*, *8*(5), 193-203.

Lowry, M., Rayadurgam, S., Schumann, J., Pressburger, T., & Dalal, M. (2017). Integrating run-time and design-time assurance for AOS. In *Safe and Secure Systems and Software Symposium (S5)*.

Meredith, P. O., Jin, D., Griffith, D., Chen, F., & Roşu, G. (2012). An overview of the MOP runtime verification framework. *International Journal on Software Tools for Technology Transfer*, *14*(3), 249–289.

Moosbrugger, P., Rozier, K. Y., & Schumann, J. (2017). R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. *Formal Methods in System Design*, *51*(1), 31–61.

Olofsson, A., Nordström, T., & Zain-ul-Abdin. (2014). Kick-starting high-performance energy-efficient manycore architectures with Epiphany. *CoRR*, *abs/1412.5538*.

Pellizzoni, R., Meredith, P., Caccamo, M., & Rosu, G. (2008). Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In *RTSS08* (pp. 481–491).

Reinbacher, T., Brauer, J., Horauer, M., Steininger, A., & Kowalewski, S. (2014). Runtime verification of microcontroller binary code. *Science of Computer Programming*, *80*, 109–129.

Reinbacher, T., Függer, M., & Brauer, J. (2013). Real-time runtime verification on chip. In *RV13* (pp. 110–125).

Reinbacher, T., Geist, J., Moosbrugger, P., Horauer, M., & Steininger, A. (2012). Parallel runtime verification of temporal properties for embedded software. In *MESA12* (pp. 224–231).

Reinbacher, T., Rozier, K. Y., & Schumann, J. (2014). Temporal-logic based runtime observer pairs for system health management of real-time systems. In *TACAS14* (pp. 357–372).

RTCA. (2012). *DO-178C/ED-12C: Software considerations in airborne systems and equipment certification*. Retrieved from http://www.rtca.org

Schumann, J., Moosbrugger, P., & Rozier, K. Y. (2016). Runtime Analysis with R2U2: A Tool Exhibition Report. In *RV16* (pp. 504–509).

Schumann, J., Roychoudhury, I., & Kulkarni, C. (2015). Diagnostic reasoning using prognostic information for unmanned aerial systems. In *PHM15*.

Schumann, J., Rozier, K. Y., Reinbacher, T., Mengshoel, O. J., Mbaya, T., & Ippolito, C. (2013). Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. In *PHM13* (pp. 381–401).

Schumann, J., Rozier, K. Y., Reinbacher, T., Mengshoel, O. J., Mbaya, T., & Ippolito, C. (2015). Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. *IJPHM*, *6*(1), 1–27.

Tolker-Nielsen, T. (2017). *Exomars 2016 - Schiaparelli anomaly inquiry* (Tech. Rep.). European Space Agency. Retrieved from http://exploration.esa.int/jump.cfm?oid=59176

Tsai, J. J. P., Fang, K. Y., Chen, H. Y., & Bi, Y. D. (1990). A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Trans SW Eng*, *16*(8), 897–916.

Watterson, C., & Heffernan, D. (2007). Runtime verification and monitoring of embedded systems. *IET Software*, *1*(5), 172-179.

BIOGRAPHIES

Dr. Johann Schumann is Chief Scientist for Computational Sciences with SGT, Inc. and working at the NASA Ames Research Center. He received his PhD (1991) and German habilitation degree (2000) in Computer Science from the Technische University Munich in Germany. He is engaged in research on sensor and software health management, autonomy for UAS, V&V of advanced air traffic control systems, and the automatic generation of reliable code. He is author of a book on theorem proving in software engineering and has published numerous articles on automated deduction and its applications, automatic program generation, V&V of safety-critical systems, and neural network oriented topics.

Patrick Moosbrugger, MSc. is a PhD student at the Cyber Physical Systems group at Vienna University of Technology. He received his MSc (2015) in Electronic Engineering from the UAS Technikum Wien for which he was awarded an honorary prize by the Austrian Federal Ministry of Science Research and Economy. Patrick Moosbrugger is engaged in research on verification and diagnostics of Cyber Physical Systems.