

Application of Accelerated Processing Units in Safety-Critical Systems

Dawid Trawczynski
Advanced Micro Devices

Janusz Zalewski
Florida Gulf Coast Univ.

ABSTRACT

The paper discusses application of Accelerated Processing Units (APU) in safety analysis and design of safety-critical systems. Specifically, a dependability simulation technique called software fault injection is discussed in the context of APU application to respective injection algorithms. We discuss parallel fault seeding techniques that can be used to expand and speed-up the fault space exploration. These techniques are presented in the context of a case-study of an Anti-Lock Braking System.

CITATION: Trawczynski, D. and Zalewski, J., "Application of Accelerated Processing Units in Safety-Critical Systems," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 6(1):2013, doi:10.4271/2013-01-0179.

1. INTRODUCTION

Increasing complexity, functionality and presence of safety-critical systems in today's rapidly evolving embedded applications, requires improved analysis and design techniques to assure that safety is not compromised [2]. To that extent, techniques that enable testing for dependability are of particular interest because they can detect and mitigate early system risks costly to resolve in later design stage or even production. Software fault injection, a stress testing method, is one such technique used in academic as well as industrial settings.

Stress testing is very useful to analyze system behavior in abnormal, high-load conditions that can occur during normal system operation [4], [10], [11]. It allows detection of problematic behavior and thus adjusting a design to correctly handle uncommon operational scenarios. However, the fundamental limitations of testing still apply, primarily, the size of a fault space, which can be very large in complex systems. Thus, there is a need to evaluate large fault space with shorter simulation time and increased structural coverage.

Taking into account limitations of current sequential stress testing, we propose to exploit recent off-the-shelf microprocessors with integrated graphics engine cores (such as AMD APU) to increase fault coverage by using integrated Single Instruction Multiple Data (SIMD) engine cores. Such approach can speed-up parallel workload based simulation,

and thus evaluate greater number of faults in a test case run. The minimum requirement for this is having a data-based parallel workload that can be applied to safety-critical algorithms. This requirement is a necessary condition so that all SIMD cores within an APU are busy processing useful data and thus operating at maximum efficiency.

We make a point that the purpose of our approach is not to use APU explicitly in safety-critical systems (although in [17] such case is already presented) but rather to use an APU to simultaneously inject many test vectors into an algorithm to stress test the algorithm more efficiently by concurrently simulating more faults in a given amount of time than possible with sequential injection.

This paper is structured as follows. First, we discuss hardware and software aspects of an APU that are beneficial in stress test simulation. Second, we focus on essential information related to one particular stress testing approach, namely Software Implemented Fault Injection (SWIFI). Third, we discuss a representative safety-critical case study, car's Anti-Lock Braking System based on the standard Proportional Integral Derivative (PID) linear control algorithm in the context of APU simulation testing. Finally, we summarize our findings and outline future research directions.

2. BACKGROUND

To understand a parallel driven approach to stress testing of safety-critical control applications, one needs to be familiar with the graphical processing of parallel workloads, as well as fault injection framework and methodology that is used to support the testing effort. The following two sections present those basic concepts, and Section 2.3 discusses related work on fault injection and tolerance with Graphical Processing Units (GPU).

2.1. Accelerated Processor Basics

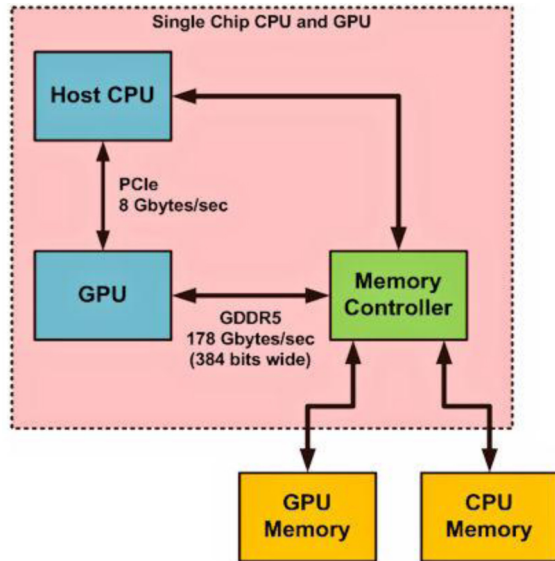


Figure 1. APU Architecture [14]

Accelerated Processing Units are multi-core microprocessors that have been integrated on a silicon die with a graphics engine known as GPU (Fig. 1). Due to shorter interconnect length, advanced power management (i.e., power and clock gating), this new breed of computing processors can function at a much lower power, thus giving new computing capabilities to embedded systems such as tablets, smartphones, gaming consoles, etc.

A GPU, which is a main component of an APU, is formed from an array of SIMD cores (e.g., Compute Units in AMD's latest architectures) that operate on workloads through abstract concepts such as threads and thread groups (wave fronts). Threads that belong to the same thread group execute the same instructions operating on independent data. Figure 2 shows an example GPU core used in the AMD Radeon HD6970 GPU architecture [14].

The main sub-systems in Fig. 2 are Command Processor, SIMD, L1 and L2 cache memory, wave scheduler, and local video memory and controller (e.g., GDDR5). The main job of a command processor is to parse draw-compute calls (in form of packets) and initialize the middle level hardware (SIMD) per context for upcoming wave fronts. SIMD form a compute pipeline, performing floating and integer arithmetic on vector

registers, data and texture memory. A wave scheduler is mainly responsible for sequencing instructions and assigning their workload to correct SIMD for throughput or power efficient computation. Finally, the memory system is responsible for fetching and storing instruction data used by threads in wave fronts.

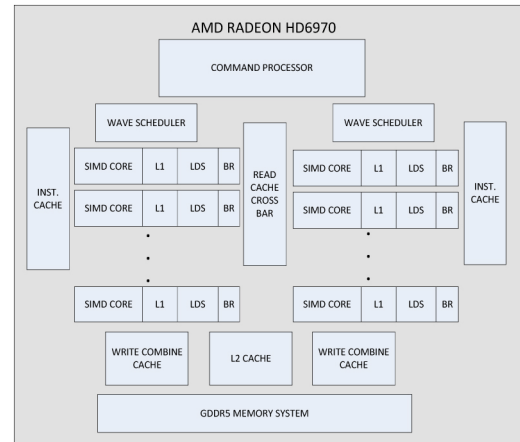


Figure 2. AMD Radeon HD6970 Graphics Core

In order, to take advantage of related GPU hardware, appropriate software interfaces (API) were developed by AMD and NVidia [14]. OpenCL and CUDA are two main languages supporting development of vector compute applications running on a GPU. Microsoft Corp. also recently introduced GPU processing into its Visual Studio Development Environment (MSVC) that uses graphics DirectX interface to communicate with a GPU (C++ AMP) [14]. All of those APIs are based on the concept of a kernel, which essentially is a function operating on vectors. Figure 3 shows an example OpenCL program (left) together with a standard C-language implementation (right) for comparison.

The difference between a sequential (non-GPU) and parallel (GPU) implementation is the need to specify the workload as an OpenCL kernel, create a context, and define implicit parallel vector computing operations with reduced *for* loop count. Such transformation can simply map to SIMD-type algorithms.

Regardless of chosen compute API, to convert a given program P into a kernel, the following steps need to be taken: $P \rightarrow P'$ and $P' \rightarrow P^*$, where P' is the original program instrumented for stress testing (via fault injection) and P^* is the GPU-based kernel program containing original application code as well as related stress testing (fault injection) code. In our approach, the fault injection code is responsible for mutation of the original operators or input data in the process of stress testing.

In the next section, we discuss some basics of this testing methodology.

```

//Integrate rows of Matrix A
for(int i=0;i<hA;i++)
{
//Integrate columns of Matrix B
for(int j=0;j<wB;j++)
{
C[i][j]=0;
// multiply and accumulate
// values in the current row
// of A and column of B
for(int k=0;k<wA;k++)
{
C[i][j]+= A[i][k]*B[k][j];
}
}
}

// wA==hB for valid
// matrix multiplication
__kernel void simpleMultiply(
__global float* output,
int wA,
int hA,
int hB,
int wB,
__global float* inputA,
__global float* inputB)
{
// get global position
// in Y direction
int row=get_global_id(1);

// get global position
// in X direction
int col=get_global_id(0);

float sum=0.0f;

// calculate result of
// one element of matrix C
for (int i=0; i<wA; i++)
{
sum += inputA[row*wA+i]
        * inputB[i*wB+col];
}
outputC[row*wB+col]=sum;
}

```

Figure 3. Matrix multiplication in a regular and OpenCL kernel program

2.2. Software Implemented Fault Injection

The main objective of SWIFI approach is to artificially accelerate the occurrence of faults and analyze system's behavior in their presence. Such analysis gives insight into the behavior of a system when a disturbance affects its resources. The main benefit of SWIFI is the ability to introduce a large number of fault conditions during system operation that can affect such resources as registers, ALU, FPU, cache memory, data bus, I/O interfaces, etc. Moreover, SWIFI can emulate many real faults occurring due to environment disturbances (electro-magnetic, radiation interference, etc.) that affect contemporary integrated circuits. The emulation can be done in a highly and repeatable controlled manner, often not possible in other approaches. Finally, SWIFI testing can provide data for analytical reliability analysis (e.g., Markov modeling) by accelerating the occurrence of faults and recording the effective error rates.

This section presents our dependability simulation methodology that integrates functional and structural models to achieve more realistic representation of system behavior than current approaches. The integration relates to the controller, environment, network and fault models used to simulate the behavior of a system in various test scenarios. In this approach, we disturb (via SWIFI) the controller and network model, to assess the behavior of an embedded

system under faulty and fault-free conditions. The general model of the system used in SWIFI simulation is shown in Fig. 4.

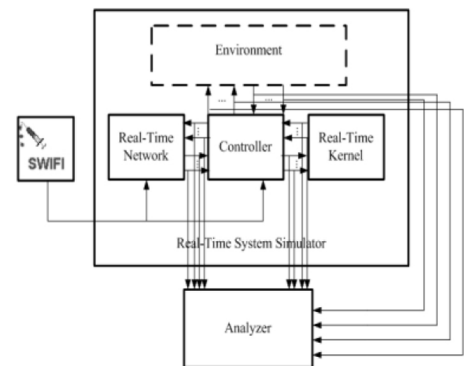


Figure 4. SWIFI system model

In this figure, six main components (SWIFI, environment, controller, network, kernel, and analyzer) are shown that constitute the experimental test bed. Note that this figure shows a mix of hardware (i.e., the controller) and software components (e.g., the real-time operating system kernel), because embedded systems are actually composed of both hardware and software. In reality, the operating system kernel actually manages the controller by executing its task schedule, and thus the two can also be considered to be in the