A toolchain-based approach to handling variability in embedded MPSoCs

Ian Gray*
University of York
ian.gray@york.ac.uk

Gary Plumbridge*
University of York
gary.plumbridge@york.ac.uk

Neil C. Audsley*
University of York
neil.audsley@york.ac.uk

August 27, 2014

Abstract

Manufacturing variability is an increasingly significant problem. Silicon devices that are designed to be identical will display widely ranging characteristics after manufacture. Power use, supported clock frequencies, and lifespan may all vary considerably. This is of particular concern for embedded systems due to their extensive use of complex SoC-based architectures. If this variability is not tolerated by the software, then manufacturing yields are reduced and devices are not used efficiently. This paper discusses a novel approach to the integration of variability-mitigation techniques that uses model-driven engineering to explicitly consider variability as part of the development process. Developers can build systems that are much more resilient to variability effects, allowing systems to have higher yields, lower costs, and greater reliability. The approach uses code generation and code transformation to simplify design space exploration and reduce time-to-market. The approach is illustrated with an example of audio processing on a complex MPSoC with simulated variability, and it is shown to be increasingly effective as system variability becomes more significant.

1 Introduction

As the fabrication of integrated circuit moves to smaller and smaller process nodes, *variability* becomes an increasing problem. Variability causes the same silicon design to display slightly different characteristics every time it is manufactured. This results in systems that are designed to be identical, but in reality

*This work is supported in part by the EU FP7 TouCHMore project (288166)

display quite significant differences in their power usage, maximum clock frequency, or expected lifespan. This effect is observed between functional units of a design (i.e. in a dual core system, the two cores may be designed to be identical but may be manufactured differently at the silicon level) and between multiple copies of the same design.

Mitigation techniques for this issue have tended to focus at the silicon level. Recently, work has started to also consider the middleware or software library level. This work argues that a complete solution to the variability issue must be supported at all levels with a toolchain-based approach that is powered by model-driven development. By capturing variability at a high level we can provide better mitigation of its effects and make more efficient use of hardware.

Section 2 will discuss variability issues and the ways in which existing systems attempt to mitigate them. Section 3 will describe a new approach that integrates model-driven engineering (detailed in section 4) and code transformations (section 5) to create a whole-stack approach to variability mitigation. The system is evaluated in section 6 and section 7 concludes.

2 Background

There is increasing consumer demand for powerful embedded devices, but the steady increase of processor frequencies that used to result from transistor scaling has largely ceased. These two factors combined to motivate the move to multicore and Multiprocessor System on Chip (MPSoC) devices, in which increased computational power is obtained from the integration of a large number of parallel processing units. Such architectures are now widely deployed throughout the multimedia and consumer mobile domains, and are being increasingly deployed in higher-criticality domains, such as automotive. [1]

These domains are driven by the three primary design requirements of low power use, low cost, and high reliability and lifespan. In particular, automotive and telecommunications often have to guarantee silicon lifespans of 20 to 30 years.

Variability is a large issue for all of these requirements. Devices can only be sold based on what they *guarantee* they can do, not what they are designed to do. Variability introduces *uncertainty* into the manufacturing process and therefore lowers the guarantees that can be made. A recent study [2] found that identical DRAM designs taken from the same wafer of the same production run vary in write power consumption by up to 22%. In this example, guaranteeing a particular power window must take this variation into account. As variability increases, wider ranges will be observed and so either *yields* must worsen or guarantees weaken.

Variability has always been measurable in VLSI manufacture, but it becomes increasingly noticeable as node size decreases [3,4]. Scaling to and past the 10nm technology node is requiring more extensive variability mitigation techniques. Section 2.1 will characterise some main sources of variability, and section 2.2 will describe existing mitigation techniques.

2.1 Types of Variability

Broadly, there are two main types of variability that exist, static and dynamic. Static effects are caused due to variation in the manufacturing process and are fixed for the lifespan of the artefact. Dynamic variation is caused by devices wearing out due to use.

There are a wide range of effects that cause variability in the silicon manufacturing process. Two significant ones are:

- Line Edge Roughness (LER) [5,6] refers to the fact that due to lithography limits, fabricated silicon wires have rough edges. This affects the off-state current and threshold voltage of transistors. LER does not largely affect performance at 90nm, but became a significant issue around the 32nm node.
- Random Dopant Fluctuations (RDF) [7] are a consequence of transistors approaching the atomic scale. When reduced to the nanoscale, dopant is discretised by its component atoms leading to fluctuations and lack of uniformity. As with LER, RDF affects the current, threshold voltages, and slew time of transistors.

When these effects and others vary the critical path of a design, its maximum clock frequency may be varied accordingly. The result is that overall performance differs from the nominal design, and varies across multiple instances of the fabricated chip. In a multicore system, this affects every core of the design.

Smaller technology nodes are also increasingly affected by wear out effects, two common examples of which are:

- Negative Bias Temperature Instability (NBTI) [8] is the gradual dissociation of molecular bonds along the silicon-oxide interface inside the transistor, leading to an increase in threshold voltage. Shown to cause a 10% voltage increase over three years of use [9] at 32nm, and worsens at smaller nodes [10].
- Hot Carrier Injection (HCI) is the phenomenon where when the device is being used, electrons can become disassociated and trapped elsewhere in the device. Over time this increases device instability. Like NBTI this effect worsens at smaller technology nodes.

There are many other effects that are observed at this scale, and new ones that are discovered at lower process nodes. Dealing with these issues is critical to prevent poor yields and therefore high costs. Section 2.2 discusses some common strategies used.

2.2 Variability Mitigation

There are a range of approaches that are used to combat the effects of variability. These occur at both the *manufacturing level* (section 2.2.1) but also at various stages throughout the software stack (section 2.2.2).

2.2.1 Manufacturing-based Mitigation

The aim of silicon manufacturers is provide devices that meet specified guarantees (on power usage, clock frequency, etc.) at yields which are as high as possible. As uncertainty increases due to the variability at lower process nodes, this is becoming difficult. There are a wide range of hardware-based techniques that are commonly deployed and a full discussion is outside of the scope of this paper, but two important approaches are *binning* and *guardbands* [11].

Binning is the general name for selling the same device under different sets of guarantees depending on post-fabrication analysis. This can involve selling devices at different speed grades depending on their performance, or by disabling failed components (e.g. selling a quad-core as a tri-core device).

Guardbands are slack inserted into the design to cope with uncertainty in the manufacturing process and to accommodate wear out effects over the device's expected lifespan. Guardbands will shrink during the device's life, and when they are exhausted timing or functionality violations can occur.

2.2.2 Software-based Mitigation

It is becoming increasingly hard to disguise the effects of variability at the manufacturing level alone. Software-based mitigation approaches accept the fact that *homogeneous embedded devices are not really homogeneous*. To this end, such approaches attempt to weaken the guarantees that are normally demanded of the hardware.

The cores in a nominally homogeneous multicore system will display very different lifespans. To prevent the entire system's lifespan being determined by the worst-performing core, guardband consumption should be as uniform as possible. A way of achieving this is to use Dynamic Frequency and Voltage Scaling (DVFS) to slow an aging core [4, 11–13].

NBTI effects can be mitigated by interleaving core activity time with idle periods where the core is placed into a recovery state. This means that runtime-task allocation can be made NBTI-aware [14, 15] by allocating work periods and recovery periods to trade system performance for system lifespan in a more gradual way than DVFS. Similarly, task allocation may attempt to maximise performance by allocating more work to cores that are performing faster. This will allow a system to maintain its performance guarantees for longer in the presence of ageing [16–18]. These techniques require accurate runtime information on the performance of the core. To do this, ring oscillators or other online monitors can be integrated into the silicon fabric [19, 20]. If this information is not available, it can be estimated online [21].

The task allocation approaches described previously are applied at runtime. Some approaches have also proposed bringing variability-awareness into existing parallel compilers. OpenMP's [22] fork-join model splits computation into work units and computes them in parallel. This gives the potential for extensions to the OpenMP compiler to manage core lifespan with fine-grained, NBTI-aware, idleness insertion [23].

2.3 Problems With Existing Approaches

This section has detailed a range of approaches that are used to handle the variability of modern silicon fabrication. However, the following problems are still observed.

- Existing variability mitigation strategies imply a trade-off. For example, the use of DVFS to reduce core ageing will also reduce performance. However, exploration of this design space is slow; requiring custom runtime support and software to be refactored.
- When such work is done, it is not portable (it cannot be moved to related architectures or applications).
- The cost of this acts as a barrier to the adoption of software-level variability mitigation techniques.
- Current approaches are not directed from the specification and design of the final system. Current systems are designed as if hardware is not subject to variability, and then the techniques applied post-design.

Section 3 details the approach described in this paper, which positions variability as a first-class component of the design chain. Through the use of model-driven engineering (described in section 3.1), hardware designs are characterised not just by their architecture or capabilities, but also by the variability they exhibit. Software can be mapped automatically over the architecture based on both design-time and run-time variability metrics. The approach is supported by code generation and code transformation (section 5) to aid automation and reduce errors.

3 Approach

The work described in this paper was completed as part of the ToucHMore project, an EU FP7 project. ToucHMore attempts to overcome the problems identified in Section 2.3 through the observation that because variability affects all levels of embedded system development, it cannot only be considered at a single abstraction level (that of source code). Effectively targeting modern MPSoCs requires the use of a tool flow in which variability is a key element of the development process. The approach combines existing variability mitigation techniques with Model-Driven Engineering (MDE), code generation and customisable compilers.

An important contribution of the proposed toolflow is the ability to use MDE to control implementation choices with regards to variability. The model-driven flow contains a description of the input hardware in terms of the variability it expresses, as well as its topology and capabilities. The model also describes the variability mitigation options that are available, such as voltage scaling or power gating capabilities. For example, a network link in the hardware model

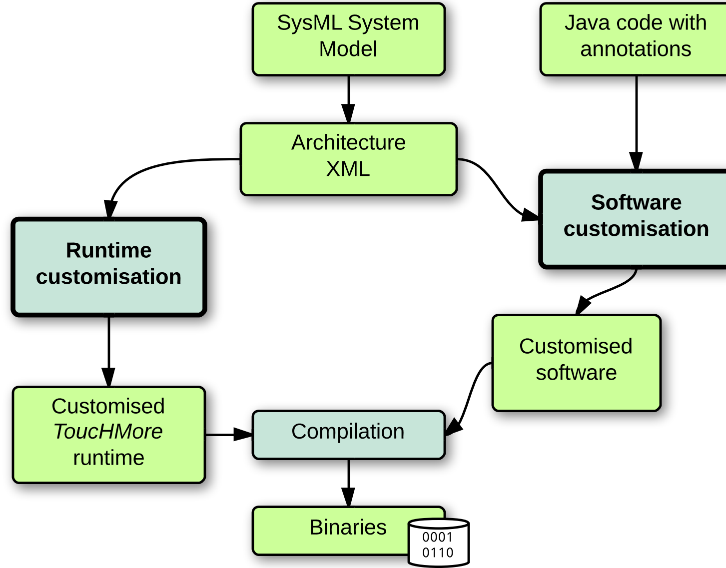

Figure 1: The toolchain of the ToucHMore approach.

may be described as having a minimum latency of 7ms, but the model will also describe whether this value is subject to variability.

Equally, the input software is modelled according to a programming model (section 3.3) that allows for the code to be transparently mapped over this architecture in terms of the variability that it contains. For example, the model can describe that a given software operation should be mapped to 'the fastest CPU in this processor group' and this will be performed dynamically, reacting to both process variability and wear-out effects (section 2.1).

Finally, software-based mitigation techniques (detailed in section 2.2.2) can be easily enabled and disabled from the modelling level over specific parts of the software and hardware. Low-power modes, work levelling, NBTI mitigation etc. can be enabled by the designer, and the behaviour of the compiler and toolchain are automatically customised to reflect these choices, accelerating design-space exploration and reducing the possibility for errors. This is effected by an automatically-customised, variability aware runtime, that is specifically targeted towards mitigating variability on the target platform for the modelled input application.

An overview of the toolchain of the described approach is shown in Figure 1.

3.1 Model-Driven Engineering

A critical requirement of this work was that it would be amenable to industrial use. MDE was used because it allows existing development methodologies to be

evolved rather than replaced. Given the target domain of embedded and safety critical systems, it is not feasible to require a new programming language or to replace large amounts of legacy code. MDE already enjoys wide adoption in the target domain [24,25], and the selected modelling language, SysML [26], has also been shown to be very effective when used to integrate academic contributions into existing industrial flows [27, 28].

This work does not argue that MDE is itself a guaranteed path to greater productivity. Instead, we show that as MDE is already being used in industry anyway, we can leverage the abstractions and procedures that it exposes to create a novel approach to variability mitigation.

3.2 Chosen Technologies

The chosen modelling language is SysML. This language is selected based on its simplicity and that it is hardware/software agnostic. The approach, however, is general and can be applied to other modelling languages.

The modelling language is used to describe applications written in JSR302-compliant Java, known as Safety Critical Java [29] (SCJ). The SCJ profile is a form of Java designed to be used in embedded, safety-critical software environments. It is based on JSR-1, the Real-Time Specification for Java [30], and is aimed at providing services for applications with demanding certification requirements, such as that of DO-178C [31]. SCJ defines models for concurrency, memory use, I/O and other features, resulting in software that is analysable for both functional and temporal correctness. This approach does not require the use of SCJ, and can be fully used in standard Java if required. We have used SCJ here because it is more appropriate for the target domain, and as a more restricted subset it results in smaller and more predictable runtimes. SCJ is a very limiting subset, however, and so some domains may wish to avoid these. The approach described in this paper is tested with SCJ, but not tied to it.

The described MDE approach implements code generation, but it is the choice of the developer whether this takes the form of full code generation from executable system models, partial stub and class structure generation, or no automatic generation.

Given the target domain, the Java bytecode is compiled to C before compilation to final binaries. JIT compilation is not considered.

3.3 System Model

The described approach uses a programming model based on the concept of *operations*. Operations are class methods in the input Java software application that are explicitly described in SysML for the purpose of allowing the developer to affect the operation's implementation. For example, the developer may deploy operations (methods) throughout the target architecture, may mark them as "low power" or apply other forms of variability-mitigation. This model is designed to reflect embedded and potentially safety-critical applications so it is largely static. Dynamism is limited only to flexibility of variability mitigation.

Operations therefore represent a point in the source application at which variability mitigation may be performed (for example, by applying power states or by moving the computation to a different core, or a different set of cores in parallel).

The implementation of operation offloading is handled by the generated runtime, detailed later in section 3.6.

The input application is a set of (JSR 302) Java classes. Each class may potentially contain a number of methods that are modelled as operations. The target hardware is a set of *processing elements*. A processing element is defined as hardware which is capable of executing (at least a part) of the input application. CPUs, GPUs, and DSPs (including soft-implementations on reconfigurable hardware such as FPGAs) are all processing elements. One of these processing elements is the *master* processing element which will host the Java classes. Other processing elements are target processing elements which can optionally host a set of *offloaded* operations. One target may contain multiple operations, and each operation can be mapped to a set of targets.

This model only explicitly considers a single application on the target architecture. Whilst it does not prevent multiple applications being hosted on the same architecture, this is outside of the scope of the model and existing analysis techniques and isolation mechanisms must be employed to ensure correct system operation.

3.4 Operation Annotations

As described in section 3.1, operations are mapped to processing elements in the architecture according to variability metrics exposed by the platform. They may also be marked for special attention by the system model, which will add some of three annotations listed below. These annotations affect the way in which the toolchain implements them.

It is not necessary to add these annotations manually. The toolflow can add them automatically to allow the integration of legacy code, as long as the code is sufficiently modelled in SysML. In this case, the mitigation instructions are carried in the model rather than the source-level annotations. In systems that use full or partial (stub) code generation from the system models these annotations are added automatically by the toolflow. Finally, the programmer can of course simply add them manually. These annotations are introduced now, but their behaviour and implementation will be discussed later:

- `@Offload` (Section 5.1) - Marks the operation as suitable for offloading from the master processing element to a target element, such as a DSP.
- `@Parallel` (Section 5.2) - Marks the method for parallel, variability-aware offloading.
- `@Energy` (Section 5.3) - Allows the developer to control the energy usage characteristics of the operation.

3.5 Operation Restrictions

Operations are implemented as Java methods with the following restrictions:

- Static, non-variadic, methods only.
- No recursion.
- May only reference static fields from their own class (which otherwise retain normal Java semantics)
- No dynamic memory allocation.
- No synchronization.
- Cannot throw exceptions.
- Arguments must be primitive types, or arrays thereof.
- May not call other offloaded methods. May call other methods if those methods also obey these restrictions.

SCJ already places restrictions on dynamic memory allocation, recursion, exception use (through its lack of a garbage collector) and synchronization so the main additional restriction is that operations should only operate on primitive data (rather than class instances).

Arguments to operations can be annotated with Java annotations to assist with optimisation of data movement:

- `@Input` argument must be passed in to the operation. Its value may not be read back out.
- `@Output` arguments should be read out after completion. Its initial value may not be sent to the operation.
- `@InOut` is the default state of an argument, and implies both input and output.

3.6 Runtime and OS support

The purpose of the ToucHMore runtime is to act as a transparent interface to the operating system and hardware for the user's application code. For example, at the modelling level the programmer can denote that a given part of the application should be executed in a low-power state. The ToucHMore code generation (discussed in section 4.3) automatically inserts calls into the user's code to the ToucHMore runtime which will indicate this desired behaviour to the runtime. It is then the job of the runtime to actually implement it through low-level hardware calls and OS system calls.

The runtime also implements the offloading and parallelisation of operations (method calls). Sections 5.2 and 5.1 describe how this happens in detail, but the general approach is:

- The user generates a SysML model which describes how the operations of the system are deployed (example in section 4).
- Model driven code generation is used to refactor the source code of the input application.
- The offloaded method is replaced with a local stub which calls into the runtime to send argument data to the remote offload target and wait for the return value.
- The implementation of the communication stubs is inside the runtime and automatically generated.

The runtime itself is partially automatically generated through the use of model-driven code generators. Embedded hardware is often quite parametrizable (based on the number of cores etc.). Rather than require the runtime to be ported to each individual configuration, the hardware models that describe a given target also contain code generators that build the runtime. The generated runtime layer reflects the variability features that are being used. For example, if the user's deployment model does not make use of any low-power states, then the runtime support to handle this does not need to be included and so is cut out. This approach allows the automatic generation of a communications layer to handle offloading of operations.

The ToucHMore runtime is built on top of a target operating system, meaning that the approach must be ported to each OS-hardware pair. Currently supported OSes are Xilinx's Xilkernel [32], FreeRTOS [33] and a 'bare metal' implementation that does not implement a full runtime. Targets such as uClinux [34] would also be very suitable although have not yet been developed. The current approach does not attempt to make the code of the operating system variability-aware, only that of the user application. In an embedded context, the majority of the system work will be in executing the user application, but this is still a potential area for expansion of the technique.

The runtime is responsible for implementing variability mitigation techniques. The aim of the approach is that the end user does not have to worry about the specific technique which is implemented. They instead define what their system should focus on (high performance, low-power etc.) and the runtime will implement this as well as is possible. The techniques implemented include:

- Use of DVFS to slow cores and save power.
- Clock gating to disable cores which are not currently being used.
- Differential work scheduling, to give more work to cores that have more desirable properties. (Higher speed cores if high performance is desired, low-power if operating in low power mode, etc.)
- Rest-period scheduling to reduce NBTI effects.

These techniques are not novel; they are simply used as described in the literature detailed in section 2.2.2.

3.7 Running Example

The rest of this paper will use a small illustrative example to show how variability-awareness is modelled and used. The application is an audio processing system which must process a large stream of data to apply a set of filters. This will be targeted at a system based on a Xilinx Zynq 7000 SoC (the XC7Z045) [35]. This SoC is a dual-core ARM Cortex A9 connected to an area of FPGA reconfigurable logic. For this example, we have placed ten Xilinx Microblaze softcore processors [36] inside the reconfigurable logic, accessible via shared memory. A diagram of this architecture is shown in the following section as Figure 2. This system is chosen because the use of the reconfigurable logic allows us to create custom architectures that are suitable for the simulation of many kinds of manufacturing variability.

4 Modelling The Running Example

This section will detail the ways that MDE is used to model the running example detailed in section 3.7.

4.1 Modelling the Example Hardware

The goal of target platform modelling is to describe:

- The processing cores, DSPs, and other computational elements in the platform (and their capabilities).
- The communications between these cores (whether they are shared memory, message passing, buses, on-chip networks etc.).
- The variability present in the modelled hardware and the features available for variability mitigation.

The platform is current using SysML *blocks*. Block inheritance is used to identify subtypes of a SysML block. This allows the modelling of future hardware properties so that new forms of hardware can be modelled without the need for additional profiles. A Block Definition Diagram (BDD) defines the existence of various hardware types and some simple value properties representing hardware capabilities. It does not define how the more complex hardware types are constructed from the SysML blocks. For this, SysML's Internal Block Diagram (IBD) is used. IBDs show the internals of a SysML block in terms of parts typed by other SysML blocks. Figure 2 shows the structure of the Zynq example architecture.

Finally, the model contains hardware variability and variability mitigation capabilities. Capabilities currently modelled are:

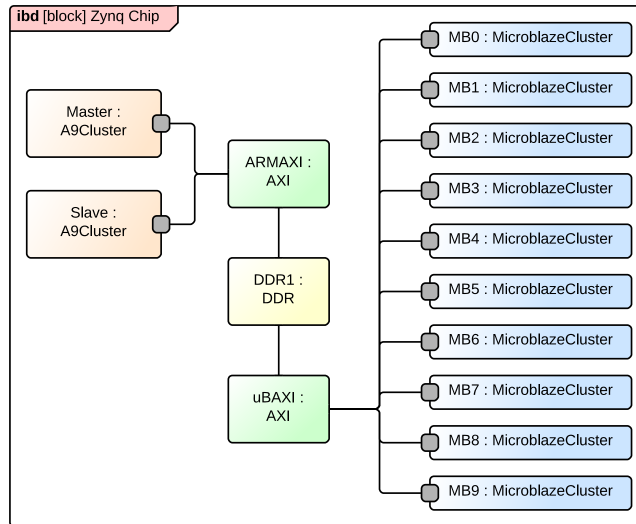

Figure 2: IBD of the Zynq architecture. The blocks are defined in BDD and may themselves have further IBDs if they have internal structure.

- Power saving capabilities of a component
  - Clock gating
  - Voltage gating
  - Voltage or frequency scaling (DVFS)
- Sensing abilities to measure:
  - Temperature
  - Supply voltage
  - Power consumption
  - Memory latencies (core to memory)
  - Communication latencies (core to core)
  - Current maximum clock frequency (using wear sensing)
  - Current battery levels

Capabilities are detailed in SysML BDDs. This is shown for the example architecture in Figure 3.

4.2 Deployment Modelling

The approach does not mandate a specific modelling style for the input software. It instead uses deployment modelling, so the only requirement is that sufficient software elements are present in the model to facilitate this.

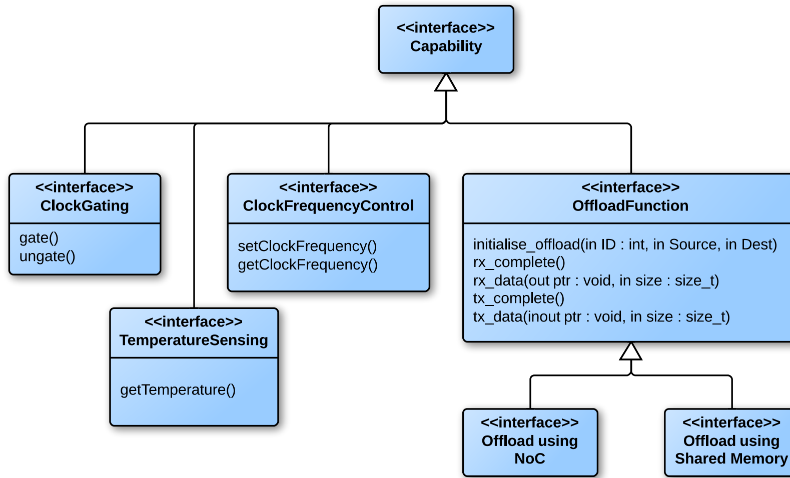

Figure 3: BDDs can also describe variability features. This shows definitions of variability mitigation capabilities: Clock gating, DVFS, and function offloading.

Deployment modelling describes the placement of software operations throughout the target hardware in terms of *maps*. A deployment map identifies which processor core types a given operation is built for and, for offloadable methods, to which processing elements it should be offloaded. Each map is represented by a stereotyped package with dependencies on exactly one platform model and exactly one application model. A map connects operations, classes or whole packages of the application, to processor core instances of the target platform. This indicates that those elements of the application will be built for each of the processor cores of the target platform. In the case where an operation is mapped to a single processor, that processor will always host that software. In the case where multiple processors are specified, the variability aware runtime is given the capability to move operations to any of those processing elements in order to best fulfil its variability mitigation requirements.

Maps do not have to enumerate specific processors, it is also possible to map operations to all processors of a given type, rather than individual processors. This is shown in Figure 4. In this deployment, the main application is mapped to the master ARM core and the `Filter` operation is mapped to the Microblaze cores. If, due to variability, the core frequency of the master drops below 700Mhz then the second ARM core will also be used for offloads.

4.3 Model Transformation and Code Generation

As discussed previously in section 3.1, code generation is used in the approach to convey information from the system models to the implemented design. Generation of stub Java code (with the annotations mentioned in section 3.4) is an

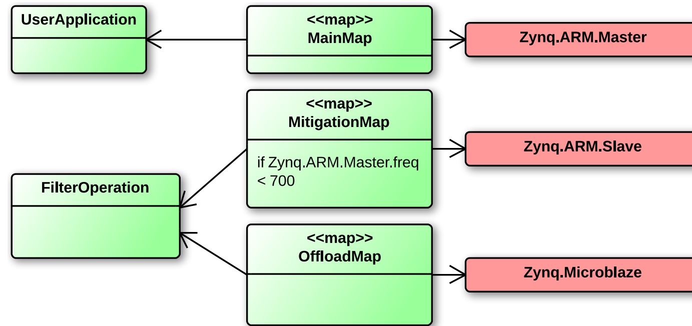

Figure 4: Deployment map of the running example.

established approach in industrial settings and so is not described in this paper. For example, it is supported by the well-known Artisan Studio [37] and Enterprise Architect [38] tool suites. The modelling tool used in the approach supports generation of Java code stubs from a class diagram, but also for edits to that generated code to synchronise back into the class diagram, ensuring that the two artefacts remain consistent.

Code generation is also used to customise the behaviour of the variability-aware runtime according to the system model and deployment mappings. An XML file is generated which contains the required information. This file is then read by the compiler and runtime to customise their behaviour.

The XML contains the logical structure of the application, including the manner of communications between processing nodes. This is used by the code transformations (section 5) and runtime to implement the desired software mapping. Each processing core is also described in terms of a range of variability metrics and the sensors that are available. The application is broken into operations that can be associated with different variability mitigation policies which can be automatically applied by the runtime accordingly.

5 Transformations

This section details the code transformations that are implemented in the described approach. As discussed in section 3, Java's annotation system is used to mark operations (methods) that are to be implemented differently to account for variability. The annotations come either from the programmer, or from a code generation stage in the MDE, and covey extra information than is normally carried in the source code alone.

Annotations are implemented using a code transformation created for this approach called *Java2Java*. This tool operates on compiled Java bytecode. This section details the three annotations that Java2Java implements, showing examples of the transformations as applied to the running example from section 3.7.

Java2Java implements a safety-critical thread pool to implement parallel method offloading. The threads in this pool must therefore be considered when schedulability analysis is performed. This analysis is possible because they are static in number, and all accesses to the pool take bounded time without self-suspension (blocking). The thread pool is only present on the master processor, the slaves implement a minimal bootloader and do not have a significant overhead.

5.1 `@Offload` Annotation

`@Offload` implements synchronous transfer of control from a thread on the master node to a slave node. Control returns once the slave is finished processing. This is implemented in three phases:

- Modification of class files containing the `@Offload`ed methods.
- Generation of class files for each target.
- Generation of native code for each target.

The original `@Offload` method is transformed into a static method and is renamed. This will be executed by the slave to perform the computation. A new native static method with the same name as the original method is created. This method will be called on the master to communicate with the slave. The target location of the offload is determined from the SysML deployment map (section 4.2) and communication code is automatically generated according to the platform XML description (from the SysML model, see section 4.3). This code also calls into the variability-aware runtime to query decisions that were made in the model. For example, to where should the offload be performed, and under what conditions. These are all implemented automatically without intervention from the programmer.

Figure 5 shows an example of target code which has been automatically generated to implement offloading. This is the code which runs on the target of an offload, so it is started as a separate thread by a minimal stub bootloader. It waits to receive the arguments to the method, calls the original method (which has already been converted to C) and then sends back the result. This code was generated because the deployment model showed that this offload would occur across a network with no access to shared memory. If more efficient transfer mechanisms existed, they would be used.

5.2 `@Parallel` Annotation

The `@Parallel` annotation allows multiple `@Offload`s to be executed concurrently. Unlike existing parallel frameworks (i.e. OpenMP) the distribution of work can be automatically optimised to be variability-aware, such as by moving more work to cores that are currently faster, display lower power usage, or longer lifespan.

```
void do_offload_2201(int channel_id, jint shared_memory_flag) {
   // Receive arg1
   jobject arg1;
   if(shared_memory_flag) { // Receive only the array address:
     offload_receive(channel_id, &arg1, sizeof(arg1));
   } else { // Receive array length
     jint arg1_length;
     offload_receive(channel_id, &arg1_length, sizeof(arg1_length));
     if (arg1_length >= 0) {
       // Allocate array in local stack:
       // ...detail omitted...
       // Receive array content
       offload_receive(channel_id, arg1, arg1_length * sizeof(jshort));
   }

   // Receive arg2
   // ...detail omitted...

   //Call the actual offloaded code
   _pico_VectorSum2_vectorSum_12201___3S_3I(arg1, arg2);

   // Send back @Out parameters content
   if(!shared_memory_flag) {
     // Send back arg2
     offload_send(channel_id, arg2, PICO_arrayLength(arg2)*sizeof(jint));
   }
}
```

Figure 5: Fragment of code generated for the target of an offloaded method.

Due to the target application domain of embedded and safety-critical environments, the use of SCJ places restrictions on the parallel execution model. The presented model is small, predictable, and analysable through its SCJ implementation, but it does not allow for the rich parallel programming features of, for example, the Java Concurrency Framework or similar. It is designed to be a first-step towards low-overhead, embedded concurrency which is variability-aware.

- When `@Parallel` is applied to a method, every invocation of that method may result in a number of concurrent invocations of the method at runtime. Computation may be executed on other slaves of the architecture.
- These invocations are identical, except for their parameters. Scalar parameters are copied to all invocations. Array parameters may be passed in their entirety, but more commonly they will be passed as sub-arrays (termed chunks) with different invocations receiving different chunks.
- At the point of the method invocation, the invoking thread is suspended and a set of threads spawned to execute the concurrent invocations of the method. For clarity, these threads are called *threadlets*.

- The variability-aware runtime is queried to determine how many threadlets should be used (and therefore the number of chunks that array parameters are split into).
- The invoking thread remains suspended until all the threadlets have completed and the results of the work have been aggregated (un-chunked). This is an implied barrier synchronisation on the completion of the method.

Work-stealing is not used. The parallel annotation framework uses an application-wide static thread pool to spawn the threadlets of the parallel method. This pool is implemented using `javax.safetycritical.MangedThread` and the pool size is static, determined by the developer at compile time, and serves all concurrent `@Parallel` calls. The transformation process is as follows:

- Modify the `main()` method to create a global immortal instance of `ThreadPool`
- For each `@Parallel`-annotated method `m`, rename `m` to `_m_Threadlet` and create a replacement method `m` which:
  - Determines the number of threadlets to use by querying the runtime
  - Determines the target locations and work distribution from the runtime
  - Splits the input array parameters into chunks according to distribution
  - Creates one `Runnable` per target, passing the input chunks. These runnables call `_m_Threadlet`.
  - Submits the `Runnable`s to the thread pool
  - Implements a barrier synchronization which passes when all threadlets are complete.
  - Collects the resulting work, and unchunks it into the output arrays.

`_m_Threadlet` is still annotated with `@Offload` so it will be processed as a normal offloadable method.

5.3 `@Energy` Annotation

The modeller can set attributes in the SysML deployment map to define the execution characteristics for operations. This information is then carried into the code through the `@Energy` annotation. The annotation is used to trigger mitigation strategies (such as those described in section 2.2.2) that are coded into the variability-aware runtime. The features of the target platform to be utilised (such as thermal sensors, wear sensors, DVFS, etc.) are not carried in the annotation but the system XML (section 4.3). There are a range of power schemes available and a discussion of such is outside of the scope of this paper, but a set of schemes is exemplified in section 6.

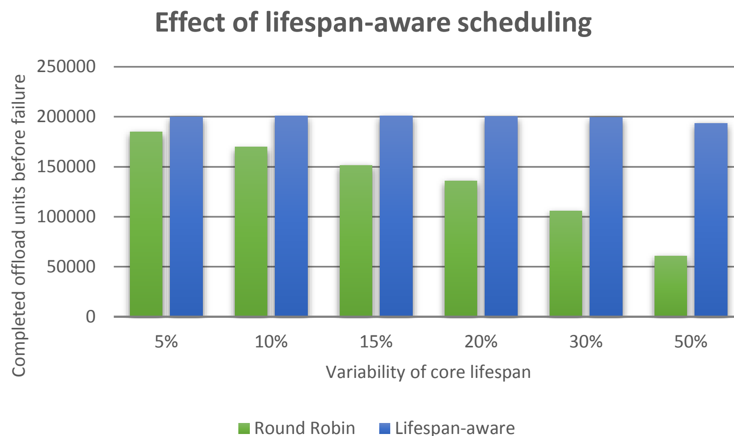

Figure 6: Effect of life-span aware scheduling in the presence of simulated NBTI variability (101 trials, $p < 10^{-16}$).

As with the other annotations in the presented toolflow, Java2Java processes the `@Energy` annotation. The processing adds calls to the variability-aware runtime at the entry and exit of the annotated method to set and reset execution characteristics that are specified by the programmer.

6 Evaluation

This section will demonstrate the use of the proposed approach on the running example. In all tests, the audio processing is offloaded in parallel from the ARM cores to the ten Microblaze cores. Different kinds of variability will be simulated on these cores.

Recall that the purpose of this approach is not to demonstrate novel variability mitigation or to show the relative benefit of one type of mitigation against another. This work uses existing 'off-the-shelf' techniques detailed in section 2.2.2. The contribution made by this work is to demonstrate the ease with which such approaches can be dropped in and used on existing code and architectures. In all examples, no code modification was necessary.

6.1 Wear-out Mitigation

In this test, NBTI, and HCI effects (see section 2.1) are emulated on the Microblaze cores, resulting in cores that will 'wear out' over time. As these cores are used their guardbands are gradually exhausted. At guardband exhaustion, the core is deemed to have failed. In the modelled situation, each core starts with a random guardband from a normally-distributed range (measured in capacity

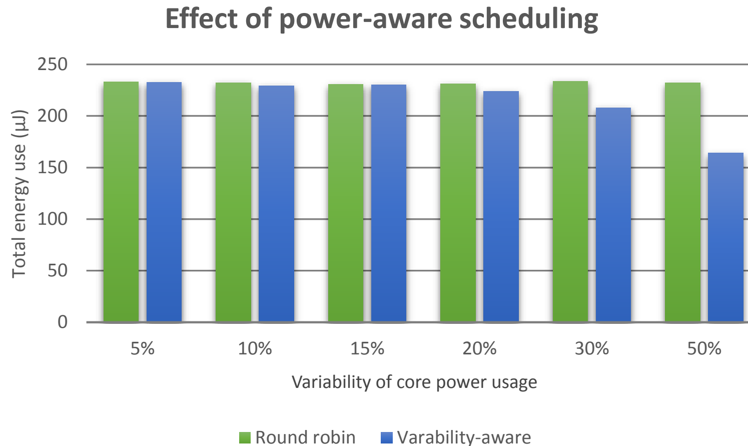

Figure 7: Effect of power-aware scheduling over variable core power use (100 trials, $p < 0.001$).

to perform work). Different variances are tested to show the effects of differing levels of manufacturing uncertainty.

In a normal SoC the slowest core dictates system lifespan. By applying the variability-aware runtime can be instructed to, when performing `@Parallel`-annotated operations, implement variability-aware offloading which gives cores with the largest guardbands the most work to perform. No programmer intervention is required.

Cores must be marked as having the ability to measure or estimate their own guardbands in the hardware capabilities of the system model (see section 4.1). On the Zynq FPGA fabric this is simulated, but a real target would provide critical path estimation peripherals that can be queried by the runtime.

Figure 6 shows the result of 101 executions of the running example, which is an audio processing application. The graphed data shows the mean amount of data processed before any single core exhausts its guardband (and therefore is considered to have failed). Standard round robin scheduling is compared with variability-aware scheduling, and all comparisons display statistical significance with $p < 10^{-16}$.

As core lifespan variability increases, the mean work completed decreases when round robin scheduling is used because it does not avoid the weakest cores. The lifespan-aware scheduling can avoid this and mitigate the effects of variability, up until 50% variability when work done begins to reduce due to the scheduling granularity supported by the runtime.

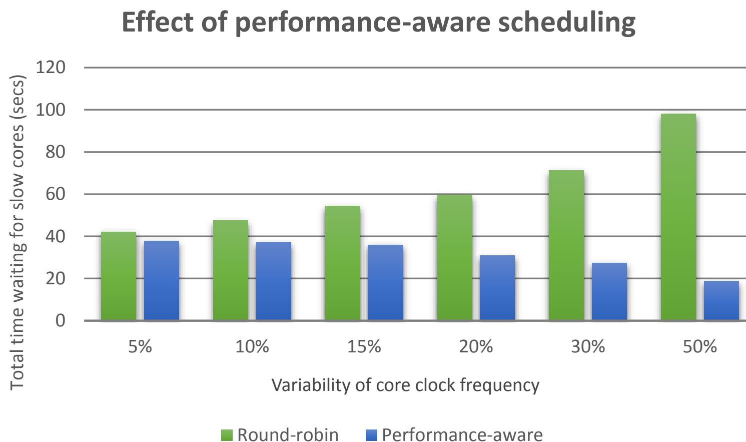

Figure 8: Effect of performance-aware scheduling over variable speed cores (49 trials, $p < 10^{-16}$).

6.2 Low-Power under Variability

In this test, we show how the variability-aware runtime can be used to reduce power usage in the presence of uncertainty. The power consumption of the cores are modelled as being normally distributed about a mean of 1200pJ per instruction. This value is chosen for comparison with similar embedded processors. Execution time analysis was performed on the audio example to determine the average number of instructions it performs. This gives a simple measure of energy use which is scaled by the amount of work offloaded to the core. Power-use per instruction is then varied. More complex power analysis exists, but is outside of the scope of this paper.

Again, the core must be able to measure its own power usage according to the system hardware model. Figure 7 shows power-aware scheduling (averaged over 100 trials) which allocates less work from `@Parallel` operations to cores which consume more power, compared with the same round-robin scheduling as above.

In this test, at variability 20% and above the power-aware scheduling shows lower power use with a confidence of $p < 0.001$. Below 20% no significant difference can be demonstrated, again likely due to the granularity of offloadable work.

6.3 Performance-aware Scheduling

Finally, core clock frequency variability can be emulated by inserting a variable amount of waiting time into each core after it performs some offloaded computation. The waiting time inserted is normally-distributed with a mean that is equal to the average time taken to perform one unit of offloaded work. The

waiting time is scaled linearly by the amount of work offloaded to the core. This allows a core's apparent speed to be either faster or slower than the system mean clock speed.

The runtime can measure the core's apparent speed and a performance-aware scheduler can offload more work to the faster cores with the aim to reduce latency in the design. The result of this can be seen in Figure 8.

As can be seen, as variability increases, the standard round robin scheduler causes the system to experience an increasing amount of blocking time (when the faster cores have completed their work and are waiting on a barrier synchronisation for the slower cores to finish). Conversely, the performance-aware scheduler results in better system performance as variability increases. This is because at high variability the runtime can preferentially exploit cores that are randomly faster than average, whilst avoiding the negative effects of cores that are slower than expected. For this test 49 trials are performed and all comparisons are significant with $p < 10^{-16}$.

7 Conclusions

This paper has discussed a novel approach to the integration of variability-mitigation techniques through the use of model-driven engineering. By explicitly considering variability as part of the development process, developers can more easily build systems that are much more resilient to variability effects (such as wear-out, or uncertain clock speeds and power use). This can allow the development of systems with greater yields than offered by toolchains that are not variability-aware. The proposed approach leverages code generation and transformation, combined with a simple programming model suitable for safety-critical systems, to easily integrate existing variability mitigation techniques in a way that is transparent to the developer. This can simplify design space exploration and reduce time-to-market.

The approach is illustrated with an example of audio processing on a complex MPSoC with simulated variability, and it is shown to be increasingly effective as system variability becomes more significant.

References

[1] Christian El Salloum, Martin Elshuber, Oliver Hftberger, Haris Isakovic, and Armin Wasicek. The ACROSS MPSoC A new generation of multi-core processors designed for safetycritical embedded systems. *Microprocessors and Microsystems*, 37(8, Part C):1020 – 1032, 2013. Special Issue on European Projects in Embedded System Design: EPESD2012.

[2] M. Gottscho, A. Kagalwalla, and P. Gupta. Power Variability in Contemporary DRAMs. In *IEEE Embedded Systems Letters*, 2012.

[3] E. Flamand. Strategic Directions Toward Multicore Application Specific Computing. In *Proc. IEEE Conf. Design, Automation and Test in Europe*, page 1266, 2009.

[4] A. Tiwari and J. Torrellas. Facelift: Hiding and Slowing Down Aging in Multicores. In *Procceedings of the IEEE/ACM International Symposium on Microarchitectures*, pages 129–140, 2008.

[5] G. Leung and Chi On Chui. Variability of Inversion-Mode and Junctionless FinFETs due to Line Edge Roughness. *Electron Device Letters, IEEE*, 32(11):1489–1491, Nov 2011.

[6] J.A. Croon, G. Storms, S. Winkelmeier, I. Pollentier, M. Ercken, S. Decoutere, W. Sansen, and H.E. Maes. Line edge roughness: characterization, modeling and impact on device behavior. In *Electron Devices Meeting, 2002. IEDM '02. International*, pages 307–310, Dec 2002.

[7] Greg Leung and Chi On Chui. Variability impact of random dopant fluctuation on nanoscale junctionless FinFETs. *Electron Device Letters, IEEE*, 33(6):767–769, 2012.

[8] A. Krishnan, V. Reddy, S. Chakravarthi, J. Rodriguez, S. John, and S. Krishnan. NBTI impact on transistor and circuit: models, mechanisms and scaling effects. In *Technical Digest. IEEE International Electron Devices Meeting*, pages 14.5.1 – 14.5.4, 2003.

[9] K. Kang, S.P. Park, K. Roy, and M.A. Alam. Estimation of statistical variation in temporal NBTI degradation and its impact on lifetime circuit performance. In *ICCAD 2007: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 730–734, 2007.

[10] V.P.-H. Hu, Ming-Long Fan, Chien-Yu Hsieh, Pin Su, and Ching-Te Chuang. FinFET SRAM Cell Optimization Considering Temporal Variability Due to NBTI/PBTI, Surface Orientation and Various Gate Dielectrics. *Electron Devices, IEEE Transactions on*, 58(3):805–811, March 2011.

[11] M. Agarwal, B. Paul, M. Zhang, and S. Mitra. Circuit failure prediction and its application to transistor aging. In *Proceedings of the 25th IEEE VLSI Test Symposium*, pages 277–286, 2007.

[12] S. Eyerman and L. Eeckhout. A Counter Architecture for Online DVFS Profitability Estimation. *IEEE Transactions on Computers*, pages 1576–1583, 2010.

[13] M. Eireiner, S. Henzler, G. Georgakos, J. Berthold, and D. Schmitt-Landsiedel. Delay characterization and local supply voltage adjustment for compensation of local parametric variations. *IEEE Journal of Solid-State Circuits*, pages 1583–1592, 2007.

[14] Ayse Kivilcim Coskun, Tajana Simunic Rosing, Keith A. Whisnant, and Kenny C. Gross. Temperature-aware MPSoC Scheduling for Reducing Hot Spots and Gradients. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, ASP-DAC '08, pages 49–54, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[15] Lin Huang, Feng Yuan, and Qiang Xu. Lifetime Reliability-aware Task Allocation and Scheduling for MPSoC Platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 51–56, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[16] F. Paterna, A. Acquaviva, F. Papariello, A. Caprara, G. Desoli, and L. Benini. Variability-Aware Task Allocation for Energy-Efficient Quality of Service Provisioning in Embedded Streaming Multimedia Applications. *IEEE Transactions On Computers*, 2012.

[17] R. Teodorescu and J. Torrellas. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. *ACM SIGARCH Computer Architecture News*, pages 363–374, 2008.

[18] F. Paterna, A. Acquaviva, F. Papariello, G. Desoli, and L. Benini. Variability-Tolerant Workload Allocation for MPSoC Energy Minimization under Real-Time Constraint. In *Proc. IEEE Workshop Embedded Systems for Real-Time Multimedia*, pages 134–142, 2009.

[19] J.M. Levine, E. Stott, G.A. Constantinides, and P.Y.K. Cheung. Online Measurement of Timing in Circuits: For Health Monitoring and Dynamic Voltage and Frequency Scaling. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 109–116, April 2012.

[20] B. Rebaud, M. Belleville, E. Beigne, M. Robert, P. Maurine, and N. Azemard. An Innovative Timing Slack Monitor for Variation Tolerant Circuits. In *Proc. IEEE Conference on IC Design and Technology*, pages 215–218, 2009.

[21] L. Zhang, L.S. Bai, R.P. Dick, L. Shang, and R. Joseph. Process Variation Characterization of Chip-Level Multiprocessors. In *Proceedings of the ACM Conference of Design Automation*, pages 694–697, 2009.

[22] Robit Chandra et al. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[23] Andrea Marongiu, Andrea Acquaviva, and Luca Benini. OpenMP Support for NBTI-Induced Aging Tolerance in MPSoCs. In Rachid Guerraoui and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 5873 of *Lecture Notes in Computer Science*, pages 547–562. Springer Berlin Heidelberg, 2009.

[24] Parastoo Mohagheghi and Vegard Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 432–443. Springer Berlin / Heidelberg, 2008.

[25] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

[26] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[27] Imran Rafiq Quadri, Andrey Sadovykh, and Leandro Soares Indrusiak. MADES: A SysML/MARTE high level methodology for real-time and embedded systems. In *ERTS2 2012: Embedded Real Time Software and Systems*, 2012.

[28] I. Gray, N. Matragkas, N. Audsley, L. S. Indrusiak, D. Kolovos, and R. Paige. Model-based hardware generation and programming - the MADES approach. In *2nd IEEE International Workshop on Model-Based Engineering for Real-Time Embedded Systems Design (MoBE-RTES)*, 2011.

[29] M. Schoeberl. A Profile for Safety Critical Java. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC '07*, pages 94–101, 2007.

[30] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[31] Radio Technical Commission for Aeronautics Inc. (RTCA). DO-178C - Software Considerations in Airborne Systems and Equipment Certification, January 2012.

[32] Xilinx Corporation. Xilkernel. http://www.xilinx.com/ise/embedded/edk91i_docs/xilkernel_v3_00_a.pdf, December 2006.

[33] Real Time Engineers Ltd. FreeRTOS. http://www.freertos.org/, Accessed Aug 2014.

[34] Arcturus Networks Inc. uClinux - Embedded Linux Microcontroller Project. http://www.uclinux.org/, Accessed Aug 2014.

[35] Xilinx Corporation. Zynq-7000 All Programmable SoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/, April 2014.

[36] Xilinx Corporation. Microblaze Processor Reference Guide. UG081 v13.2, 2011.

[37] Atego. Atego Modeler. http://www.atego.com/products/atego-modeler/.

[38] Sparx Systems. Enterprise Architect. http://www.sparxsystems.com/products/ea/, 2014.