Designing Resource-Constrained Embedded Heterogeneous Systems to Cope with Variability

Ian Gray
University of York, UK
Andrea Acquaviva
Politecnico di Torino, Italy
Neil Audsley
University of York, UK

ABSTRACT

As modern embedded systems become increasingly complex they also become susceptible to manufacturing variability. Variability causes otherwise identical hardware elements to exhibit large differences in dynamic and static power usage, maximum clock frequency, thermal resilience and lifespan. There are currently no standard ways of handling this variability from the software developer's point of view, forcing the hardware vendor to discard devices which fall below a certain threshold.

This chapter first presents a review of existing state of the art techniques for mitigating the effects of variability. It then presents the toolflow developed as part of the TOUCHMore project, which aims to build variability-awareness into the entire design process. In this approach, the platform is modelled in SysML, along with the expected variability and the monitoring and mitigation capabilities that the hardware presents. This information is used to automatically generate a customised variability-aware runtime which is used by the programmer to perform operations such as offloading computation to another processing element, parallelising operations, and altering the energy use of operations (using voltage scaling, power gating etc.). The variability-aware runtime affects its behaviour according to modelled static manufacturing variability and measured dynamic variability (such as battery power, temperature, and hardware degradation). This is done by moving computation to different parts of the system, spreading computation load more efficiency, and by making use of the modelled capabilities of the system.

1 INTRODUCTION

It is becoming increasingly difficult to efficiently exploit complex Multiprocessor Systems-on-Chip (MPSoC) architectures using existing programming languages and approaches. This is due to two main issues:

1. Modern MPSoC platforms have a very complex programming model, but the languages commonly used to develop software for them (C, C++ etc.) present a very simple view of hardware. This is the "programming model gap".

2. Hardware variability causes systems that were designed as regular architectures to become irregular once they are manufactured, and to change over time.

Commonly used languages such as C, Java and C++ all assume a homogeneous implementation architecture with a uniform, shared memory space. This is incompatible with the application-specific, heterogeneous architectures of MPSoCs – specifically parallelism, non-uniform memory architectues (NUMA) and non-standard communications (i.e. on-chip networks). This problem is compounded when variability is considered.

Variability is the observation that as the manufacture of integrated circuits moves to lower and lower process nodes, the transistors become increasingly variable. This gate-level variation leads to large differences in the performance of the final design. Therefore, multiple copies of the same design may exhibit considerable differences in static and dynamic power consumption, lifespan and clock frequency. A system designed as a homogenous MPSoC will be heterogeneous after manufacturing variability is considered. This is a major challenge for the development of both the hardware and software of future embedded systems.

This chapter begins in Section 2 by describing in detail the kinds of variability that exist in modern embedded systems. Section 3 then discusses existing approaches that attempt to mitigate the effects of such variability. Sections 4, 5 and 6 detail the approach taken in the ToucHMore project, an EU FP7 research project which is focussed on the development of variability-aware systems. Finally, Section 7 summarises potential areas for future work in this area and Section 8 concludes.

2 BACKGROUND - VARIABILITY IN MULTICORE SYSTEMS

Due to the increasing demands placed on modern embedded devices, multicore devices are now commonplace. They are deployed to address the high performance and energy efficiency requirements imposed by audio, video, mobile telephony, and gaming applications. Moreover, multicore systems are becoming widespread in the automotive infotainment and power-train domains; especially in the context of hybrid and electric vehicles where energy efficiency is critical.

Technology scaling has traditionally offered advantages to embedded systems in terms of reduced energy consumption and increased performance without requiring significant additional design effort. Developers could expect performance improvements "for free". However, scaling to and past the 22 nm and 14 nm technology nodes brings a number of problems. Random intra-die process variability, reliability degradation mechanisms, and their combined impact on system-level quality metrics (i.e. power consumption or maximum clock speed) are prominent issues that will need to be tackled in the next few years. In particular, due to aggressive technology scaling, sub-65 nm CMOS technology nodes are increasingly affected by variation phenomena, and multicore architectures are impacted in many ways by the variability of the underlying silicon fabrics (Flamand, 2009) (Tiwari & Torrellas, 2008).

Variability causes significant perturbations to the performance and power consumption of multicore platforms. This is of particular interest to multicore systems (Bowman, Alameldeen, Srinivasan, & Wilkerson, 2007) (Humenay, Tarjan, & Skadron, 2007) (Sylvester, Blaauw, & Karl, 2006) because it leads to systems that were designed as homogeneous multicore systems but in which each core runs at a different speed and uses a different amount of power to do so. For example, a recent study (Gottscho, Kagalwalla, & Gupta, 2012) found that supposedly identical DRAM chips from the same wafer of the same production run may vary in write power consumption by up to 22%. Furthermore, variations may increase at runtime due to aging and wear-out phenomena. This may cause failure of single components, when the speed of the circuits becomes too slow to be properly sampled by the clock signal. In the rest of this section we provide details about both static and dynamic sources of uncertainty in modern multicore systems. In Section 3 we will outline current countermeasures adopted to face these issues.

2.1 Static Uncertainty: Process Variability

The progress and scaling of CMOS technology has encountered a number of walls. The most obvious is the fact that the dimensions of silicon devices are approaching the atomic scale and are hence subject to atomic uncertainties. According to the International Technology Roadmap for Semiconductors (ITRS, 2012), this becomes of concern at 45nm, and becomes critical at the 22nm technology node and below. Other issues impair technology scaling even before this. Lithography resolution, photo resist and electrical field limits (due to power supply voltage fluctuations, thin oxide breakdowns, etc.) are critical issues for 65nm and 45nm technologies.

Two different types of process variations have different impacts on multicore architecture design. Intra-die process variations result in significant core-to-core frequency variations (Cao & McAndrew, 2007) (Herbert & Marculescu, 2008) (Bowman, Alameldeen, Srinivasan, & Wilkerson, 2007). Simultaneously, global variations lead to inter-die variability. The result is that overall performance differs from the nominal design, and varies across multiple instances of the fabricated chips (Ndai, Bhunia, Agarwal, & Roy, 2008). In the produced chips, critical paths can be faster or slower than designed, meaning that the clock frequency of each core needs post-fabrication calibration. Faster

cores are overclocked and slower cores are clocked at a lower frequency.

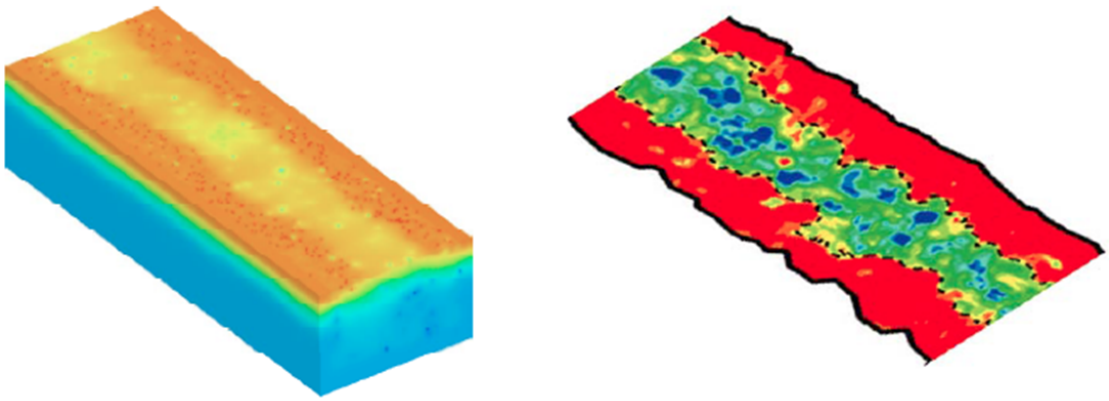

Figure 1: (left) 3D simulation of a 35 nm MOSFET fabricated under the effects of variability. (right) Illustration of the potential distribution in such a device (Fujitsu).

Statistical variability introduced predominantly by the discrete nature of the electron charge and the granularity of matter has become a major limitation to MOSFET scaling and integration. It already adversely affects the yield and reliability of SRAM, causes timing uncertainty in logic circuits and exacerbates on-chip power dissipation problems. Figure 1 illustrates the variability introduced by random discrete dopants and line edge roughness in a 35 nm gate length MOSFET.

2.2 Runtime Uncertainty: Wear-out effects

New technology nodes will also be increasingly affected by runtime uncertainty of performance and power consumption values. This dynamic uncertainty is mainly due to wear-out phenomena and temperature-related effects. Temperature itself may accelerate wear-out and chip degradation in a non-uniform way, especially in the presence of hot-spots.

As a consequence, embedded MPSoCs fabricated in upcoming nanometer technologies will be increasingly affected by aging mechanisms, leading to threshold voltage increase (Karl, Blaauw, Sylvester, & Mudge, 2008) which implies circuit slowdown. Typically, guardbands (GB) are inserted to compensate for circuit delay. These GB will shrink during core activity until their complete consumption will lead to timing violations. In the absence of correction mechanisms, these violations will result in system failure. In multicore platforms an additional reliability issue is that both the initial GB margin and its consumption rate are not uniform across the cores. Thus, nominally homogeneous cores will have drastically different lifetimes. Preventing the less reliable core from dictating the entire system lifetime requires the GB consumption to be equalized as much as possible. At the system level this can be obtained by monitoring the GB consumption (Agarwal, Paul, Zhang, & Mitra, 2007) (Eireiner, Henzler, Georgakos, Berthold, & Schmitt-Landsiedel, 2007) and slowing down the aging process of less reliable cores (Tiwari & Torrellas, 2008).

The strategy to slow down the aging of cores depends on the considered aging effect. The main aging phenomena affecting nanometer devices are Negative Bias Temperature Instability (NBTI) and Hot Carrier Injection (HCI), for which wear-out takes place only during activity periods. In particular, NBTI has gained much attention from recent research because it is considered a dominant effect (Krishnan, Reddy, Chakravarthi, Rodriguez, John, & Krishnan, 2003). NBTI is due to the dissociation of Si-H bonds along the silicon-oxide interface in presence of a negative bias ($Vgs = -Vdd$) on PMOS transistors, which causes the generation of traps. These traps lead to the increase in the

threshold voltage. Recent studies demonstrate that NBTI leads to up to a 10% voltage increase over a three year lifetime (Kang, Park, Roy, & Alam, 2007).

The NBTI degradation model is characterized by a recovery effect, caused by the reduction of interface traps when the negative bias is removed. As a result, the threshold voltage decreases. Thus, NBTI-induced aging can be partially compensated by imposing a virtual ground (i.e. a logical "1") to PMOS transistors gates for a certain period of time (the recovery period) where the core is idle from a functional viewpoint. As a result, it is possible to slow-down GB degradation by interleaving core activity with idle periods where the core is placed in a recovery state. The impact of NBTI does not depend on the granularity and distribution of stress / recovery periods but only on their total duration (Kumar, Kim, & Sapatnekar, 2006). This makes it possible to efficiently distribute the required idleness with convenient granularity.

In Section 3 we will describe the main techniques used to compensate for NBTI through runtime task allocation.

2.3 Runtime Uncertainty: Temperature effects

Aggressive MPSoC scaling exacerbates thermal effects. Power densities are increasing due to transistor scaling, thereby reducing the chip surface available for heat dissipation. Also, in an MPSoC the presence of multiple heat sources increases the likelihood of temperature variations over time and chip area rather than just a uniform temperature distribution across the entire die (Mulas, D.Atienza, Acquaviva, Carta, Benini, & De Micheli, 2009). Overall, it is critically important to control temperature and bound the on-chip gradients to preserve circuit performance and reliability in MPSoCs.

2.4 Challenges

Given the issues outlined above, the reality of modern multicore platforms is that each core must be characterized by its own clock frequency, static power, and dynamic power, and that these values can vary from the nominal value at runtime depending on wear-out and temperature conditions. Without any compensation and knowledge at software and application level, the consequences on quality of service (QoS) can be severe. Parallel algorithms for video processing, for instance, assume a symmetric workload distribution amongst the cores. However, the heterogeneity caused by variations will cause an asymmetric distribution of execution times of the various threads, leading to the situation where the slowest thread, running on the slowest core, determines the overall execution time.

The next key challenge in this area, therefore, is to integrate process technology into the architecture and system software tool flows. To achieve this target, a deep rethinking is needed of system architectures and design methodologies. In particular, the software development flow should take into account underlying platform uncertainties and at the same time exploit the presence of capabilities to monitor them. Variation-tolerant multicore platforms require circuits to monitor static and dynamic variations. The software must be able to decide when and how to apply compensation in response to static and dynamic perturbations of the nominal operating characteristics.

Software counter-measures are effective in reshaping application workload to account for variability in the underlying multiprocessor fabric. In this context, countermeasures at the software level have to be taken to optimize QoS and energy consumption by selectively allocating workload to the more efficient cores, depending on the target metric. However, such policies may greatly worsen platform lifetime as a side effect, as the most used cores will age faster and dominate the Mean Time To Failure (MTTF). Hence, workload allocation strategies are needed that optimize energy consumption

and performance while preserving reliability by adapting allocation to wear-out conditions. Such compensation policies are presented in the following section.

3 CURRENT APPROACHES

The problems caused by variability must be addressed at multiple levels of abstraction, from the circuit (Drake, Senger, Singh, Carpenter, & James, 2008) (Rebaud, Belleville, Beigne, Robert, Maurine, & Azemard, 2009) to the architectural level (Mutyam, et al., 2009) (Palermo, Silvano, & Zaccaria, 2009) (Verghese, Rouse, & Hurat, 2008). At the software level, a number of solutions have been recently proposed. The aim of these approaches is to hide the effects of both static and runtime variations on the running applications. Most existing systems are runtime-only and tend to be based on a library or middleware layer. There are also "whole-stack" approaches that include the design and compilation of the system. The ToucHMore approach, presented later in this chapter, is one such example.

Research has led to the development of a number of approaches which may be characterized as follows:

- Runtime approaches (Section 3.1) apply decisions at runtime, even if these decisions are taken offline. For example, a scheduler may be created offline which can change the allocation of tasks at runtime according to certain variability metrics.

- Compiler-assisted approaches (Section 3.2) extend this by bringing variability awareness into another piece of the software development toolchain, customizing the compiler itself, or by making code generation variability-aware.

- Finally whole-stack approaches (Section 3.3) go further still and involve all aspects of software development.

3.1 Runtime approaches

To cope with variability, knowledge of platform degradation is of key importance. This implies that it is possible to measure the GB degradation, static power and dynamic power for each core. While static information about these quantities can be characterized at post-fabrication time, wear-out and temperature effects require on-line monitors.

Such online monitors have been proposed to expose core-by-core variability in power and performance at the software level (Drake, Senger, Singh, Carpenter, & James, 2008) (Rebaud, Belleville, Beigne, Robert, Maurine, & Azemard, 2009). Consequently, policies exploiting these monitors have been developed (Chandra, Lahiri, Raghunathan, & Dey, 2007) (Eyerman & Eeckhout, 2010). For instance, if the user has information about per-core frequency, they may change supply voltage and clock frequency to improve system lifespan. However, this will greatly impact performance because in many embedded platforms, core frequency selection is very coarsely-grained and only provides a small selection of disparate speeds.

Another approach exploits task allocation. Tasks may be assigned to cores such that the amount of workload executed by each core compensates for their degradation. This approach depends on the cost function to be minimized. For example, when targeting performance, task allocation aims at compensating for speed differences amongst cores by allocating the fastest core to the largest workload. This avoids bottlenecks, but faster cores are also the most power consuming ones so this is not the best solution to minimize energy use. Equally, a non-uniform workload allocation may result in one core failing before the others because of wear-out effects. Policies which try to improve

lifespan are based on allocation of idleness to cores, such that more idleness is experienced by more degraded cores.

In other work (Teodorescu & Torrellas, 2008), variation-aware task scheduling algorithms are proposed with different power / performance objectives. In their study, the authors consider various platform configurations in which processors may have differing clock frequencies and may or may not support dynamic voltage and frequency scaling (DVFS). DVFS allows the voltage or core clock frequency of processors to be altered during runtime. A core may be slowed down at times of low system load to reduce overall power consumption and system wear. The work uses a ranking approach where tasks are first ranked by either power consumption or Instructions Per Cycle (IPC) and then mapped on the cores depending on the selected metric. Power consumption minimization is achieved by mapping the most power-consuming threads onto the lowest power cores. Maximizing performance is achieved by mapping the highest IPC threads to the highest frequency cores. When DVFS is supported, the authors explore the possibility to maximize the performance with a given power budget by an efficient distribution of voltage levels among cores. In particular, they formulate the problem using linear programming, where the result is the best selection of N voltage levels for N cores to maximize the throughput with a given power constraint. This policy however cannot be applied online due to the time overhead to compute the solution. In similar approaches (Paterna, Acquaviva, Papariello, Desoli, & Benini, 2009), a two-stage heuristic composed of a linear programming step and a bin packing step was proposed which gives a suboptimal solution to the allocation problem. However, the solution is again too expensive to be applied online.

An alternative approach uses an online technique to extract the process variation map of an MPSoC (Zhang, Bai, Dick, Shang, & Joseph, 2009). The estimation is based on temperature and power sensors. This information is exploited to perform task allocation to meet a time constraint and with minimum energy consumption. The problem is formulated using integer linear programming. Even though this is optimal, this solution cannot be computed online and thus cannot be applied on embedded systems. Other similar task allocation approaches have been recently proposed (Hong, Narayanan, & Kandemir, 2009) (Huang & Xu, 2010) (Huang, Yuan, & Xu, 2009) (Paterna, Acquaviva, Papariello, Desoli, Olivieri, & Benini, 2009).

Finally, recent work (Paterna, Acquaviva, Papariello, Caprara, Desoli, & Benini, 2012) in the domain of multimedia processing has applied information from runtime sensors but also application-level time constraints to perform task allocation. A time-constrained, variability-aware, task allocation methodology which compensates for core-level performance and power variations is applied to meet the real-time constraints imposed by the frame rate of the multimedia system, whilst minimizing energy as a secondary objective.

3.2 Compiler-assisted techniques

Compiler-level techniques introduce variability- and reliability-awareness into the compiler. In particular, various approaches have been proposed to extend existing parallel compilers to make their parallel decomposition variability-aware.

Consider the fork-join parallel processing model, in which each processing core works on a portion of a data structure and must synchronize with the others on a barrier. OpenMP is the de-facto standard for such a parallel execution model, and it features a number of MPSoC-suitable implementations (Marongiu & Benini, 2009) (Jeun & Ha, 2007). In the OpenMP model, the compiler can manage idleness insertion at the granularity of a single iteration (or chunks of iterations). This allows very fine control over the actual duration of idle and active periods, and thus on the stress and recovery time

applied to cores. Longer idle periods are allocated to processors with smaller GB. The impact of the inserted idleness on loop execution time can be evaluated so that iteration redistribution among the cores can be exploited to minimize it. Performance loss can be compensated for by reallocation of workload to cores depending on the idleness distribution.

3.3 Whole-stack approaches

Compiler-assisted approaches are promising, however recently more holistic approaches that exploit code generation from a high-level system model have also been proposed (Gauthier, Gray, Larkam, Ayad, Acquaviva, & Nielsen, 2013). This enables the insertion of variability-awareness throughout the software and hardware development process. Such an approach is pursued in ToucHMore project, in which variability and energy-aware information is used at all development stages, from system modelling, to application software, to the runtime and compiler. The ToucHMore approach is described in Section 4.

4 THE TOUCHMORE APPROACH

The ToucHMore project argues that the programming of heterogeneous MPSoCs cannot currently be handled entirely at any one level of abstraction. Effectively targeting modern MPSoCs in the presence of variability requires the use of a customisable tool flow-based approach. The approach combines existing runtime and compile-assisted techniques with model-driven engineering (MDE), code generation and customisable compilers.

A key contribution of the toolflow is the ability to make use of model-driven engineering to control the implementation of software with regards to variability. The model-driven flow uses a set of models to describe the target hardware, not just in terms of its architecture or topology, but in terms of the variability aspects present in the system. The model also describes the variability mitigation options that are available, such as voltage scaling or power gating capabilities. Equally, the input software is modelled in a way that allows the programmer to express their optimisation metrics (power saving, performance etc.) and to identify key areas of the software for special attention. From this model, a customised, variability-aware runtime is generated that is specifically targeted towards mitigating variability on the target platform for the modelled input application.

The toolchain of the project is shown in Figure 2.

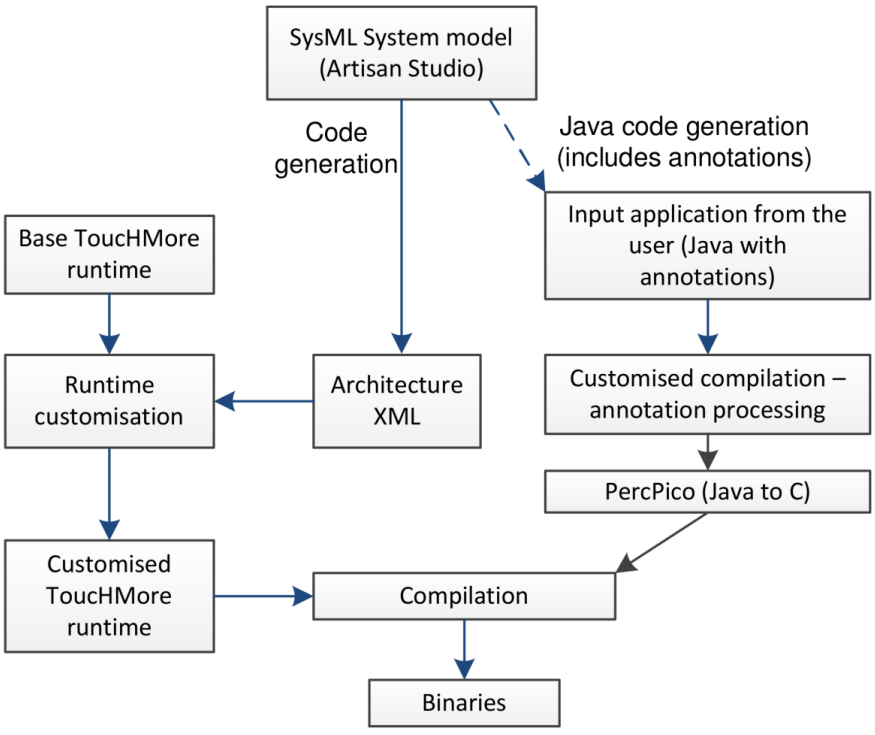

Figure 2: The ToucHMore toolflow

The input to the flow is a SysML (Weilkiens, 2011) model which describes the target hardware and the structure of the input software. The modelling aspects of the project are discussed in Section 5. From this model, code generation is used to generate Java code. The toolflow may either generate the full source code or simply an application structure, depending on the complexity of the input system model. The software may then be completed using more traditional development methodologies. As is also discussed in Section 5, the model is used to generate files that guide the customisation of the runtime to support the application software. Finally, the generated code is processed through a custom

software flow, discussed in Section 6 in order to generate the output binaries for the target architecture.

4.1 Running example

Throughout the rest of this chapter, a small example function called `sum_data` will be referred to in order to show how various aspects of the toolchain operate. `sum_data` is a simple vector sum operation, in which the input is an array of integers and the result of the operation is a single integer which is the numerical sum of all the elements of the array. This function will be generated from the system model (Section 5.1.4), offloaded to other processing cores (Section 6.3), parallelised (Section 6.4) and run in a power-saving mode (Section 6.5). This example is very simple for the purpose of clarity, but through the context of the ToucHMore project the approach has also been successfully applied to an automotive case study in which the computation of audio processing functions is moved automatically in response to system temperature, and a large set of synthetic case studies including a heterogeneous 12-core system built on the Xilinx Zynq-7000 SoC (Xilinx Corporation, 2014).

4.2 ToucHMore Methodology

The language used in this project is JSR302-compliant Java, known as *Safety-Critical Java* or *SCJ* (Schoeberl, 2007). This is a form of Java that is applicable for use in embedded, safety-critical software environments. These are the domains in which variability-awareness is currently most important due to their tendency to use non-standard architectures, battery power, limited cooling, long lifespans, and slower CPUs which demand efficient software. However, this methodology would also be very applicable to almost any embedded development process.

The ToucHMore flow is based around the concept of *operations*. Operations are elements of the input software which are modelled in the system model and represent the smallest unit of software of which the programmer can control the implementation. Operations are allocated to processing elements for execution. This model is illustrated in Figure 3.

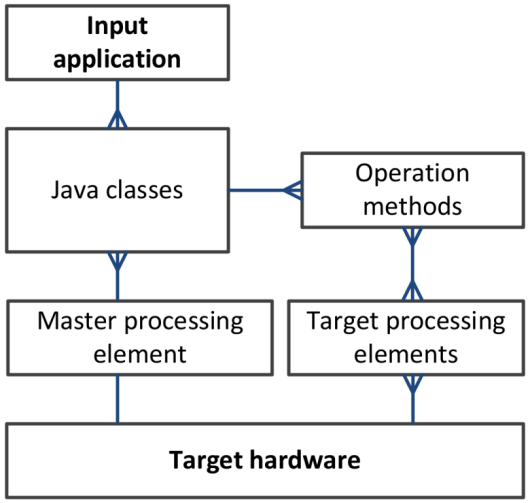

Figure 3: ToucHMore Software Model

The input application is a set of 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, which are defined as hardware capable of executing Java. (Note that in the ToucHMore project the Java code is translated to C and compiled before deployment. Other approaches may choose to use a standard JVM.) 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 operations. One target may contain multiple operations, and each operation can be mapped to a set of targets. These mappings are specified by the programmer in the system model (discussed in Section 5.1.3) and carried into this toolflow as arguments to the `@Offload` annotation, discussed in Section 6.3.

Note that this computational model does not prevent multiple applications executing at the same time on the same architecture. Indeed this is the most common model used to target complex MPSoCs because most programming languages cannot describe a single application that can execute over a heterogeneous architecture without shared memory (and Java is no different). Rather than attempt to redefine the accepted industrial methodology, this approach seeks to augment it with variability-awareness.

4.3 Operation Annotations

The operations of the input application may be tagged with three TouchHMore-specific annotations which affect the way in which code should be generated. These annotations may be added manually by the programmer, or carried into the code from the system SysML model (described in Sections 5.1.1 to 5.1.3). The meanings of these annotations are described later whilst this section concentrates on the general transformation approach. The annotations are:

- `@Offload` (Section 6.3) - Applied to an operation (method) to mark the method as suitable for offloading from the master to a target computation resource (such as a DSP).
- `@Parallel` (Section 6.4) - Applied to an operation tagged with `@Offload` to mark the method as suitable for parallel offload.
- `@Energy` (Section 6.5) - Applied to an operation to allow the programmer to control the energy usage characteristics of the operation.

All of these annotations are implemented with a combination of two approaches:

1. Bytecode transformation. The standard Java compiler is used to generate class files from the Java source code. These class files are then parsed and transformed using the ASM bytecode library (Bruneton, 2002) to change their behaviour. In the TouchHMore project, annotations do not change the functional meaning of the code, only its non-functional properties. If all annotations are removed the code will still produce the same result (assuming well-formed code without data races).
2. Code generation. The annotations customise the behaviour of the TouchHMore runtime. The runtime support required for each annotation is described in their respective sections.

5 MODEL-DRIVEN ENGINEERING IN TOUCHMORE

The TouchMore project uses model-driven engineering to integrate variability-awareness into its toolchain. The developer creates the following three models:

- A model of the target platform, describing its structure, communication, and the variability aspects of the hardware (Section 5.1.1).
- A model of the source application (Section 5.1.2) in terms of the operation model described previous in Section 4.2.
- A deployment model of the application on to the platform (Section 5.1.3).

From these models the developer uses automatic model transformations and code generation to perform the following actions:

- Create Java code which implements the source application. This may be complete code generation, or class and method stubs which are filled in manually.
- Generation of configuration files to customise the behaviour of the TouchMore variability-aware runtime.
- Generation of annotations for the Java code to mark that, for example, a given operation should be offloaded to a slave, or executed in a low-power state.

The chosen modelling language is SysML. SysML is already well-established in industrial use and models both hardware and software resources equally well. A commonly-cited weakness of UML is that it was initially software-centric. MARTE (The Object Management Group, 2011) is another common choice for embedded development but its specification is very large and complex. It covers much greater detail than is required by the TouchMore toolflow.

The next three sections briefly give examples of the three kinds of modelling in the project.

5.1.1 Target Platform Modelling

The aim of the target platform model is to describe three main elements:

- The processor cores in the platform (and their capabilities).
- The connections between the cores - in terms of shared memory, busses, or on-chip networks.
- The variability capabilities of the modelled hardware elements.

The target platform hardware is modelled using SysML blocks. In order to provide a generic way to extend the hardware properties that can be modelled without the need for additional profiles, inheritance is used to identify the subtype of a SysML block. Figure 4 shows this using an example of a Block Definition Diagram (BDD) describing the GENEPY platform (Lemaire, Thuries, & Heiztmann), a heterogeneous network-on chip-based architecture.

The 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. The SysML Internal Block Diagram (IBD) shows the internals of a SysML block, potentially in terms of parts typed by other SysML blocks. An example IBD from the GENEPY platform is shown in Figure 5.

The target platform model also describes hardware capabilities and variability. Capabilities currently modelled by the ToucHMore flow are:

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

Figure 6 shows how (a subset of) the capabilities are modelled in a SysML BDD.

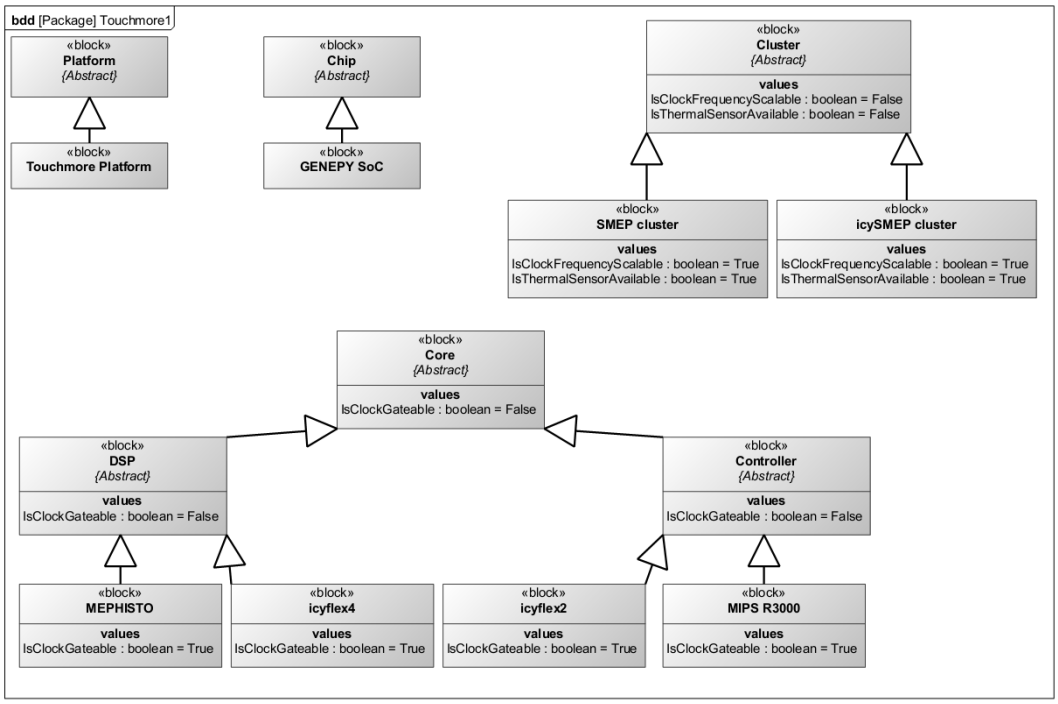

Figure 4: Example of a SysML Block Definition Diagram describing the elements of the GENEPY platform.

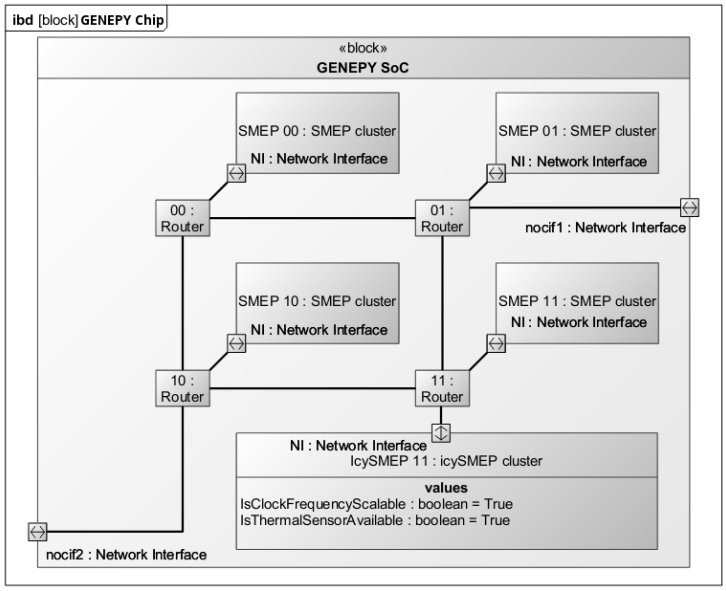

Figure 5: Example of a SysML Internal Block Definition

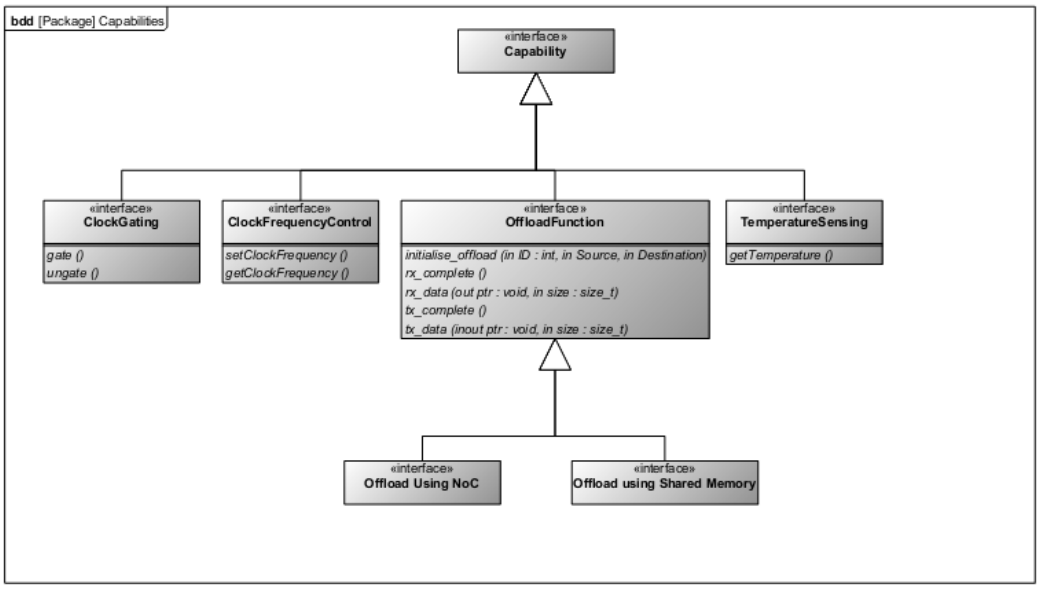

Figure 6: Subset of the capabilities described in the TouchMore flow

5.1.2 Source Application Modelling

The TouchMore approach does not restrict the modelling language used to model the source application. The only requirement is that enough of the application is modelled such that a Deployment mapping (described in Section 5.1.3) can be built. In practice, this means that all offloadable computation (operations) should be modelled.

5.1.3 Deployment Modelling

A deployment map is used to identify which processor core types a given operation is built for and to which it should be offloaded. Each map is represented by a stereotyped package with dependencies on exactly one platform model and exactly one application model. Each map connects *n* elements, where elements can be operations, classes or whole packages of the application, to *m* processor core

instances within the context of the target platform. This indicates that those *n* elements of the application (and anything scoped by them unless another mapping overrides at a lower level) will be built and deployed on each of the *m* processor cores of the target platform. It is also possible to map operations to all processors of a given type, rather than individual processors.

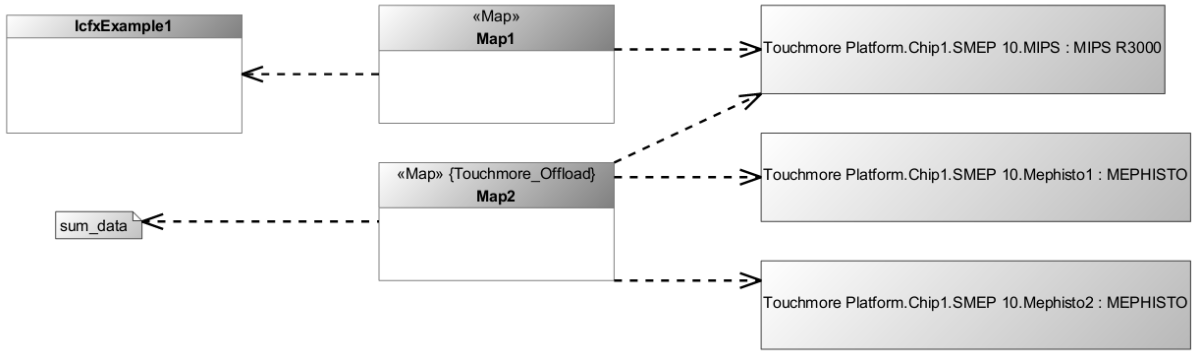

Figure 7: Sample application mapping for the `sum_data` example

Figure 7 shows an example mapping for the `sum_data` example (Section 4.1) to an example architecture. In this mapping, all operations of the class `IcfxExample` except `sum_data` are built and deployed on the MIPS processor in the SMEP cluster at location 10. The operation `sum_data` is built and deployed on the same MIPS processor but also built for the two Mephisto processors in the SMEP cluster at location 10.

5.1.4 Model Transformation and Code Generation

There are two kinds of code generation that are used in the ToucHMore toolchain:

1. Generation of Java code for implementation on the target platform.
2. Customisation of the variability-aware runtime.

Generation of Java code is handled by standard model transformation tools which transform software models into stub code for completion by the developer. The generated code is then passed to the software toolchain described in Section 6.

Code generation also customises the variability-aware runtime according to the description of variability in the platform (Section 5.1.1) and the deployment mappings (Section 5.1.3). An XML file is generated which contains this information. This is then linked in to the runtime to customise its behaviour. A complete description of this process is outside the scope of this chapter, but the following list describes the kinds of features that are affected:

- The runtime handles offloading computation to remote processors. In order to do this, the runtime must know the structure of the platform and the methods available (message passing, shared memory etc.). The architecture XML contains this information.
- The user code merely needs to call an offloadable operation. The location of where to offload and when (i.e. “always”, “only when power is below *X*%” etc.) is contained in the XML. To change these mappings and conditions, only the model needs to be updated, the software remains the same.
- The amount of work assigned to parallel operations is automatically scaled according to the variability in the platform. In a platform without runtime monitoring, this is static according

to the model. If the XML details the presence of runtime monitoring sensors then this can be dynamic.

- The ToucHMore API exposes to application software all the sensors that are described in the XML, and all the power and clock gating features present.

Figure 8 shows a fragment of the generated XML for one of the ToucHMore project's evaluation architectures. Observe that the structure and variability of the architecture are both encoded into the XML.

```
<Platform name="ToucHMore Platform" id="1">
      <Chip name="Chip1" type="GENEPY SoC" id="2">
            <Connector end1="3" end2="18" id="5" latency="none"
                  Type="GENEPYNoC">
            <Router name="00" id="17"></Router>
            <Cluster name="SMEP00" type="SMEPcluster" id="21"
                  IsClusterClockGatable="true"
                  IsClusterClockScalable="true"
                  IsClusterTemperatureMonitorable="false"
                  IsClusterVoltageGatable="false"
                  IsClusterVoltageScalable="false">
                  <Core name="Mephisto1" type="Mephisto" id="28"
                        ClockScalingDelay="none"
                        CoreClockFrequencyCurrent="manufacture"
                        ...
```

Figure 8: Edited fragment of generated architecture XML

6 THE TOUCHMORE SOFTWARE TOOLCHAIN

As was discussed in Sections 4.2 and 4.3, the ToucHMore project uses Java's annotation system to annotate operations and thereby allow the programmer to configure the deployment and runtime behaviour of code. These annotations impart information about variability-awareness to the software toolchain.

The software flow is shown in Figure 9. Recall that the input application may be generated from the system SysML model, or coded directly using traditional software development.

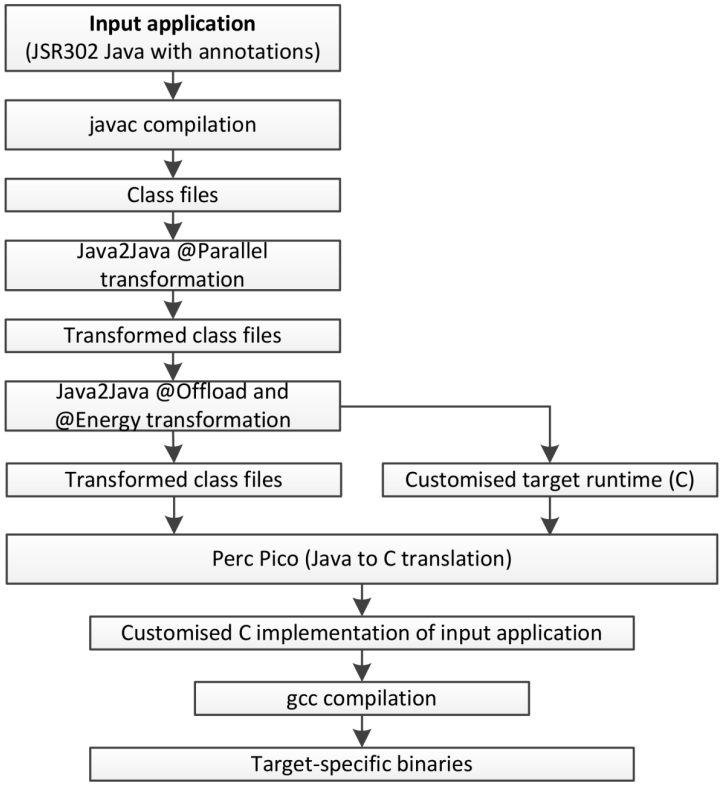

Figure 9: Customised compilation in the ToucHMore tool flow

6.1 Operations in Software

As can be seen in Figure 9, the input Java application is passed through a standard Java compiler and then a string of transformations are applied to implement the ToucHMore variability annotations.

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.
- No dynamic memory allocation.
- No synchronization.
- Cannot throw exceptions.
- Arguments must be primitive types, or arrays thereof.
- May not call other `@Offload`-annotated methods. May call other methods if those methods also obey these restrictions.

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

- *Input arguments* will be read, so their contents must be passed in to the operation. Without this annotation the runtime may omit this copy phase.
- *Output arguments* will be generated by the operation, so their contents should be read out after completion. Without this annotation the runtime may discard the contents of the argument.
- *InOut* is the default state of an argument, and implies both input and output.

These transformations are applied by a tool called *Java2Java*.

6.2 Source Transformation - the Java2Java Tool

The Java2Java tool transforms Java class files using the ASM bytecode transformation framework. It can support any Java code that conforms to the ToucHMore system model and that obeys the restrictions listed in Section 6.1. It requires a minimal runtime, but this runtime is designed to be compliant with Safety Critical Java and so has bounded execution time and memory use. In the toolchain, after Java2Java is complete a Java to C compiler is used to create C code which is finally compiled by a normal C compiler to produce target binaries.

Java2Java implements a *master-slave* computation model for computation offloading. The *master* processing element is where the majority of the application's classes execute (see Section 4.2). A *slave* is a processing element to which operations may be offloaded for calculation. Thus, Java2Java transforms the user's annotated application into several executables; one version for the master and one version for each of the designated targets for offloading.

On the master, Java2Java's runtime implements a thread pool that it uses to implement parallel method offloading. This means that the user's application will require more threads than are specified in the initial source code. Any real-time schedulability analysis must be performed on the output of Java2Java rather than the input. The thread pool implementation is compliant with Safety Critical Java. For the slaves, Java2Java's runtime only implements a minimal bootloader and so does not have any significant effect on resource usage.

A full discussion of the internal operations of Java2Java is outside the scope of this chapter. However, the following sections describe the specific transformations that are implementation by Java2Java in more detail.

6.3 Annotations for Offloading - `@Offload`

As discussed in Section 6.1, there are a range of limitations on the Java code that may be part of an `@Offload`-tagged method. In addition to these, for consistency with the input Java, the implemented offloading mechanism uses a synchronous model in which the calling thread is blocked until the operation completes.

The `@Offload` annotation is implemented by Java2Java in three main phases.

1. Modification of the class files containing the `@Offload` methods.
2. Generation of new class files for each target.
3. Generation of new native files for each target.

For each `@Offload` method, the transformation performs the following:

- The original `@Offload` method is transformed into a static method and is renamed. This part will be executed by the slave side to perform the actual computation.
- A new native static method with the original `@Offload` method's name is created. It embeds generated C code to implement the master side of the offloading mechanism. This side makes calls into the variability-aware runtime to determine whether or not an offload should take place, and if so, to where the computation should be offloaded.

An example of this transformation for the `sum_data` example is shown in Figure 10.

```
/* Original user-annotated method */
public static sum_data([S[I)V
@Loffload/Offload;(targets={"Mips"},id=2201,sharedMemory=Loffload/Offload$S
haredMemory;.RUNTIME_DECISION)
    @Loffload/In;() // parameter
    @Loffload/Out;() // parameter
   L0
    LINENUMBER 51 L0
    ICONST_0
    ISTORE 2
      //bytecode continues...


/* Original method transformation */
public static native sum_data([S[I)V
@com.percpico.util.mc.NativeMethodImpl(nativeCode="native_implementation”)
@com.percpico.util.mc.NativeMethodDep(depends={@com.percpico.util.mc.Native
MethodDep.Dependency(clazz=SumData, methods={"sum_data_2201([S[I)V"})})
    @Loffload/In;() // parameter
    @Loffload/Out;() // parameter

public static sum_data_2201([S[I)V
@Loffload/Offload;(targets={"Mips"},id=2201,sharedMemory=Loffload/Offload$S
haredMemory;.RUNTIME_DECISION)
    @Loffload/In;() // parameter
    @Loffload/Out;() // parameter
   L0
    LINENUMBER 51 L0
    ICONST_0
    ISTORE 2
   //bytecode continues...
```

Figure 10: An example of `@Offload`-annotated method transformation. Original method (above) and its transformation (below).

In addition to the transformation, a new C file is generated for each target. It contains a loop in which it waits for incoming offload requests, calls the appropriate native function to perform the requested computation, and returns the results. An example of this transformation is shown in Figure 11.

```
int 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_SumData2_sum_1data_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));
    }

    // End
    return 0;
}
```

Figure 11: Native C function to implement offload communication on the target side for the `sum_data` *example*

6.4 Annotations for Parallel Execution - `@Parallel`

The `@Offload` annotation does not imply parallel execution. During the execution of an operation tagged with `@Offload` the thread of control moves from the host computing element to the target element and only returns once the operation is complete. The purpose of `@Parallel` is to avoid this, and to allow a single operation to be offloaded to multiple targets concurrently. The work of the operation is split amongst the targets according to the variability parameters provided from the XML generated in Section 5.1.4.

`@Parallel` may only be applied to methods tagged with `@Offload`. The restrictions that are already applied to operations (discussed in the introduction to Section 6) are sufficient to ensure that the translated code will maintain functional correctness. The developer, however, should be aware that a single call to a `@Parallel` method may result in multiple threads of computation being spawned throughout the application. They can control this through the deployment map in the system model (Section 5.1.3).

6.4.1 Parallel Execution Model

The use of SCJ in the toolflow places some restrictions on the parallel execution model.

- Memory use is strictly controlled in SCJ. New object instances are created in specific *allocation contexts*, each of which is specified with hard limits on their maximum size. Overflowing any allocation context results in a runtime exception.
- Garbage collection is not present in an SCJ system, meaning that memory is only reclaimed when an allocation context is destroyed.

- These points imply that the programmer must be able to analyse any library or framework they use and statically analyse the total number of allocations made.

Consequentially, the parallel computation model is described as follows:

- The `@Parallel` annotation is used by the programmer to mark methods that are considered for parallel execution. 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. Shared memory is neither required nor assumed, but will be used if present to reduce communication overhead.
- 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. Work is balanced by the variability-aware runtime at the point of invocation but once execution has started it is not redistributed. This allows a much tighter bound on the worst-case response time of an operation.

The presented model is designed to be small, predictable, and analysable. Consequentially it does not allow the same rich parallel constructs available in the Java 8 concurrency framework. Instead it is designed to be a first step to variability-aware, low-overhead concurrency in an embedded domain.

6.4.2 Method Parameters

Most parallel methods will operate on large arrays of data. Rather than pass the entire array to each threadlet, it is usually optimal to split arrays into chunks and pass only a subset of the chunks to each threadlet. The `@Parallel` annotation marks every array parameter with an integer `chunkSize` which describes the smallest amount of each array which is required by any given threadlet. The programmer can assume that after chunking, the length of array parameters to the `@Parallel` method will be at least their `chunkSize`, but they are likely to be longer. The exact length will be a multiple of the `chunkSize`, may vary between different invocations at runtime, and is adjusted by the variability-aware runtime according to runtime parameters. For example, if the runtime is offloading a parallel operation to two remote DSPs, it may choose to pass a larger volume of data to the DSP which is cooler, or which due to design-time variability is slightly faster or has a lower power usage than the other. Details of the kinds of schemes that may be implemented were described in Section 3.

An example of the `@Parallel` annotation used to perform the `sum_data` example in parallel is shown in Figure 12.

```
@Parallel
@Offload{targets = {"Mips"}}
public void sum_data(@In(chunkSize = 1) int[] input, int[] output) {
      for(int i = 0; i < input.length; i++) {
            output[0] = output[0] + input[i];
      }
      return total;
}

public void main(void) {
      //Create the arrays
      int[] input = ...
      int[] output = ...

      //Call the parallel method
      sum_data(input, output);

      //Total the collected return values
      int total = 0;
      for(int i = 0; i < output.length; i++) {
            total = total + output[i];
      }
}
```

Figure 12: An example of the `@Parallel` annotation

6.4.3 Implementation of `@Parallel`

The parallel annotation framework uses an application-wide static thread pool to spawn the threadlets of the parallel method. After the transformation, these threadlets will execute concurrently and call `@Offload`-tagged methods. The `@Offload` transformation will then transform these methods as described in Section 6.3. Currently there is no standard thread pool in SCJ so the framework includes an implementation of one. The thread pool uses instances of `javax.safetycritical.MangedThread`, which is part of SCJ level 2. The size of the thread pool is fixed, determined by the programmer, and specified during the build process. The thread pool can serve threadlets to multiple concurrent parallel invocations.

The transformation process is as follows:

- Modify the `main()` method to create a global immortal instance of `ThreadPool` for use by the parallel methods.
- For each `@Parallel` annotated method `m`:
    - Rename `m` to `_m_Threadlet`
    - Create a replacement method `m` which does the following:
        - Call the runtime to find out how many threadlets to use, `n`, and where to offload each one to.
        - Split the input parameters of the parallel method into `n` sub-arrays.
        - Create an array of `n` `Runnables` where each:
            - Recieves a split of each input parameter
            - Calls `_m_Threadlet` passing the input parameters
        - Submit the `Runnables` to the global thread pool.
        - Collect the resulting work into output arrays.

The threadlet method is still annotated with `@Offload` after this transformation, which is then processed as described in Section 6.3.

6.5 Annotations for Energy Awareness - `@Energy`

The deployment model (described in Section 5.1.3) can define the execution characteristics for operations. These include goals such as ‘energy minimization’, ‘power minimization’ and ‘hotspot reduction’. The `@Energy` code annotation captures these goals and passes them to the Java2Java translator. The generation of customization information from the hardware platform model (described in Section 5.1.4) allows the runtime to implement these goals based on decisions which depend on the energy, power and thermal characteristics of the platform.

These features of the target platform are passed through the customization path of the toolchain (the generated XML) and not through the `@Energy` annotation itself. The purpose of this annotation is to define whether or not a certain method is to be given specific "care". The customization path provides to the runtime the instruments which allow it to implement decisions in a platform-dependent way. For example, the customization path may inform the runtime about average dynamic and static power consumption of cores, presence and quantity of temperature sensors. This can then be used to offload computation to minimise power use or similar. Since minimization of energy, power or temperature may degrade performance, the flow allows the programmer to define bounds in terms of timing constraints and QoS degradation. This information is passed through the annotation.

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. An example of this is shown in Figure 13 in which the `sum_data` example is annotated to be executed with energy minimisation optimisations.

```
@Energy(energyMinimization=true)
@Offload(targets = {"Mips"}, id = 2201, sharedMemory = RUNTIME_DECISION)
public static void sum_data(@In short[] data, @Out int[] result) {
      TouchmoreRuntime.energy(true, false, false, 0, 0, 0);

      int sum = 0;
      for (int i = 0; i < data.length; i++) {
            sum += data[i];
      }
      result[0] = sum;

      TouchmoreRuntime.energy_end();
}
```

Figure 13: The `sum_data` example transformed by @Energy. The two italicised lines have been added to pass energy optimisation information to the runtime.

If used alone, `@Energy` instructs the runtime to apply a frequency and voltage scaling policy. In order to define the minimum frequency level, the runtime will exploit any time constraint information that is passed in the `@Energy` annotation. Also, the `@Energy` annotation (when minimising energy use) attempts to deactivate as many system components as possible. This relies on the variability-aware runtime. The techniques that are employed are described in Section 3. Section 5.1.4 described how the system model is transformed into an XML file which details the variability capabilities of the system. This file is used by the variability-aware runtime to determine what system components can have their power disconnected, which do not support it, and which do support it but are currently in use (perhaps from an `@Offload`, or `@Parallel` method) and so they should be deactivated later.

When used together with `@Offload` and `@Parallel`, the `@Energy` annotation indicates to the runtime how the decision of where to offload to must be performed. This impacts the target task allocation. When using `@Offload` and `@Energy`, the target is selected to minimize energy or minimize temperature hotspots. Energy and temperature optimization are not always the same. For instance, a hotspot reduction policy may execute code in two cores which are physically far apart from each other, even if this consumes more energy through communication than two closer cores.

The annotation supports deadline and QoS parameters that are used as inputs to the defined optimizations. Frequency and voltage scaling and workload to core allocation are currently used, but many other potential policies are possible, as discussed in Section 3.

Time constraints and QoS bounds are used to tune the aggressiveness of energy and temperature policies. There is a trade-off between performance and energy or thermal optimisation because such optimisation often requires clock frequency reduction or because it imposes additional delays for component shut-down. Many state-of-the-art policies concerning joint variability and energy optimization require some performance constraint information in order to achieve the best trade off between energy and performance. The use of such an annotation can assist the programmer in the investigation of this trade off because changing parameters and deployments requires only changing the SysML model, no software needs to be altered.

7 FUTURE RESEARCH DIRECTIONS

There are a number of areas of future research that could be followed from the work discussed in this chapter.

The purpose of the ToucHMore project was to investigate how to integrate variability-awareness into the embedded development flow. Consequentially, the programming model used is not as expressive as some existing systems. To extend the applicability of the toolflow, the following extensions could be explored:

7.1 Asynchronous Offload Semantics

Currently offloaded methods are synchronous, meaning that when an offload is called, the caller is blocked until the method completes. This is consistent with the basic Java model and so is the approach taken. However, Java also has support for asynchronous method invocations through the use of the `Future` interface (Oracle Corporation, 2013). Futures represent the result of a computation that may not yet have completed. They are returned immediately from an asynchronous call, and will be updated with the result asynchronously once the computation completes.

Asynchronous semantics may help the programmer to use their available hardware more fully, and are used in a number of languages such as Go (Google, 2014) and Javascript (The jQuery Foundation, 2014). Interesting future work would investigate extending the programming model to support such semantics.

7.2 Extending the Parallel Programming model

In order to better concentrate on the variability issues explored by the ToucHMore toolchain, the current parallel programming model is very simple. In order to extend the applicability of the approach, the model should be unified with an existing parallel programming framework such as OpenMP (Chandra R. , 2001). The use of GPUs and other powerful accelerators could be tackled through the integration of OpenACC (OpenACC-Standard.org, 2013). Neither of these frameworks are currently variability-aware.

7.3 Online Energy Profiling

It is very challenging to develop accurate models of the power usage of modern embedded SoCs. This is because of both their complexity, and because of industrial secrecy. Often the approach taken is to attempt to measure the power consumption of a subset of operations (such as the opcodes of a processor) and then develop a power model from those measurements. However this can be time-consuming and inaccurate, and crucially does not account for variability between devices. Currently, this limits the accuracy of the profiles implemented by the `@Energy` annotation.

A possible solution to this is to use online profiling to learn about how the target system is actually performing, and then to feed this information back into the runtime for use by an adaptive energy profile. This could lead to a power-aware runtime with greater accuracy and predictive power than can be currently achieved.

8 CONCLUSIONS

Due to increasing consumer demands, modern embedded architectures are becoming increasingly complex, leading to the adoption of designs based around the use of heterogeneous, multiprocessor systems on chip. These designs require billions of transistors on a single die but with minimal power consumption, thereby motivating the use of smaller and smaller fabrication processes. As fabrication moved from the 90nm process, through 45nm, 32nm, 22nm and smaller, manufacturing variability became an increasingly significant issue. At these smaller scales, the variation in transistors that should be identical is so large that designs exhibit large deviations from their designed power consumption, clock speed, and lifespan. It has therefore become necessary to design systems with variability in mind.

This chapter has described the sources of variability in modern systems and summarised many existing state-of-the-art approaches to addressing the problems that it causes. One such approach is the customisable toolflow that is implemented as part of the ToucHMore project. The tool flow uses model-driven engineering to describe the target architecture in terms of its variability, and to deploy the user's application over it.

The toolchain allows the programmer the use of three special annotations to perform the following actions:

- Optionally offload computation to a remote processing node, depending on variability parameters.
- Parallelise multiple offloaded computations for simultaneous execution, adjusting parallelism according to variability.
- Adjust the execution of software in response to features such as battery life, temperature, or silicon wear.

These annotations are supported by a customisable ToucHMore runtime which is generated automatically from the model-driven flow to be variability-aware. This means that the runtime can affect the operation of the above features in response to manufacturing variability. For example, if in a multicore system, one processor is slightly faster due to manufacturing variability (or runtime degradation), then it may receive a correspondingly higher amount of computation from offloads and parallel operations. Similarly, a programmer may provide multiple offload locations for a given offloadable operation and allow the runtime to decide where to offload to given the variability of the system.

Together, these approaches allow the programmer to target complex architectures in the presence of variability in an efficient and portable way.

9 FURTHER READING

1) Andy Wellings, *Concurrent and Real-Time Programming in Java*. Published by Wiley, 2004, ISBN 0-470-84437-X.
2) M. Faugère, T. Bourbeau, R. de Simone and S. Gérard, *MARTE: Also an UML Profile for Modelling AADL Applications*, proceedings of ICECCS 2007, IEEE Computer Society, Auckland, New Zealand, July 11-14, 2007.
3) N. Audsley, I Gray, N. Matragkas, L. S. Indrusiak, D. Kolovos, R. Paige, *Embedded and Real Time System Development: A Software Engineering Perspective*, Springer-Verlag, 2014. András Vajda, *Programming Many-Core Chips,* Springer, 2011.
4) Ahmed Jerraya, Wayne Wolf, *Multiprocessor Systems-on-Chips*, Morgan Kaufmann, 2004. Michael Hübner and Jürgen Becker, *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, Springer, 2010.
5) H. Aydin, R. Melhem, D. Mossé, and Pedro Mejia Alvarez, *Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time System*s, Real-Time Systems Symposium, London, England, Dec 2001.
6) M. Gottscho, A. A. Kagalwalla, and P. Gupta., *Power Variability in Contemporary DRAMs,* IEEE Embedded Systems Letters.
7) L.A.D. Bathen and N.D. Dutt, *E-RoC: Embedded Raids-on-Chip for Low Power Distributed Dynamically Managed Reliable Memories*, UC Irvine. Proc., IEEE/ACM 2011 Design, Automation and Test in Europe.
8) C. Piguet, *Ultra-Low Power Processor Design*, Chapter 1 in *High Performance Energy-Efficient Microprocessor Design*, Editor Vojin Oklobdzija and Ram Krishnamurthy, Springer 2006, pp. 1-30, ISBN-10: 0-397-28594-6.
9) L. Balmelli, D. Brown, M. Cantor, M. Mott, *Model-driven systems development*, IBM Systems Journal, Volume 45 Issue 3, July 2006.
10) Chihhsiong Shih, Chien-Ting Wu, Cheng-Yao Lin, Pao-Ann Hsiung, Nien-Lin Hsueh, Chih-Hung Chang, Chorng-Shiuh Koong, William C. Chu, *A Model-Driven Multicore Software Development Environment for Embedded System*, Computer Software and Applications Conference, Annual International, pp. 261-268, 2009 33rd Annual IEEE International Computer Software and Applications Conference, 2009.
11) Xavier Guerin, Frederic Petrot, *A System Framework for the Design of Embedded Software Targeting Heterogeneous Multi-core SoCs*, IEEE International Conference on Application-Specific Systems, Architectures and Processors, pp. 153-160, 2009.
12) Schattkowsky, T.; Muller, W., *Model-based design of embedded systems*, Proceedings of Object-Oriented Real-Time Distributed Computing, pp. 113-128 2004.
13) Poletti, F., Poggiali, A., Bertozzi, D., Benini, L., Marchal, P., Loghi, M., Poncino, M., *Energy-Efficient Multiprocessor Systems-on-Chip for Embedded Computing: Exploring Programming Models and Their Architectural Support*, IEEE Transactions on Computers, vol. 56, pp: 606-621 2007.
14) Mosterman, P., *Model-Based Design of Embedded Systems*, Proceedings of International Conference on Microelectronic Systems Education 2007.

10 BIBLIOGRAPHY

Agarwal, M., Paul, B., Zhang, M., & Mitra, S. (2007). Circuit failure prediction and its application to transistor aging. *Proceedings of the 25th IEEE VLSI Test Symposium*, (pp. 277-286).

Bowman, K. A., Alameldeen, A. R., Srinivasan, S. T., & Wilkerson, C. B. (2007). Impact of Die-to-Die and within-Die Parameter Variations on the Throughput Distribution of Multi-Core Processors. *Procceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, (pp. 50-55).

Bruneton, E. (2002). *ASM 4.0: A Java bytecode engineering library.* Retrieved from http://download.forge.objectweb.org/asm/asm4-guide.pdf

Cao, Y., & McAndrew, C. (2007). Mosfet Modeling for 45 nm and Beyond. *Proceedings of the IEEE International Conference on Computer-Aided Design*, (pp. 638-643).

Chandra, R. (2001). *Parallel programming in OpenMP.* Morgan Kaufmann.

Chandra, S., Lahiri, K., Raghunathan, A., & Dey, S. (2007). System-on-Chip Power Management Considering Leakage Power Variations. *Proc. ACM/IEEE Design Automation Conference*, (pp. 877-882).

Drake, A., Senger, R., Singh, H., Carpenter, G., & James, N. (2008). Dynamic Measurement of Critical-Path Timing. *Proc. IEEE Conf. Integrated Circuit Design and Technology and Tutorial*, (pp. 249-252).

Eireiner, M., Henzler, S., Georgakos, G., Berthold, J., & Schmitt-Landsiedel, D. (2007). Delay characterization and local supply voltage adjustment for compensation of local parametric variations. *IEEE Journal of Solid-State Circuits , 42* (7), 1583-1592.

Eyerman, S., & Eeckhout, L. (2010). A Counter Architecture for Online DVFS Profitability Estimation. *IEEE Transactions on Computers , 59* (11), 1576-1583.

Flamand, E. (2009). Strategic Directions Toward Multicore Application Specific Computing. *Proc. IEEE Conf. Design, Automation and Test in Europe*, (p. 1266).

Gauthier, L., Gray, I., Larkam, A., Ayad, G., Acquaviva, A., & Nielsen, K. (2013). Explicit Java Control of Low-Power Heterogeneous Parallel Processing in ToucHMore. *International conference on Java Technologies for Real Time Embedded Systems.*

Google. (2014). *The Go Programming Language*. Retrieved January 2014, from http://golang.org/

Gottscho, M., Kagalwalla, A., & Gupta, P. (2012). Power Variability in Contemporary DRAMs. *IEEE Embedded Systems Letters*, *4.*

Herbert, S., & Marculescu, D. (2008). Characterizing Chip-Multiprocessor Variability-Tolerance. *Proc. ACM Conf. Design Automation Conference*, (pp. 313-318).

Hong, S., Narayanan, S., & Kandemir, M. (2009). Process Variation Aware Thread Mapping for Chip Multiprocessors. *Proceedings of IEEE Design Automation and Test in Europe*, (pp. 821-826).

Huang, L., & Xu, Q. (2010). Energy-Efficient Task Allocation and Scheduling for Multi-Mode MPSoCs under Lifetime Reliability Constraints. *Proceedings of IEEE Design, Automation and Test, Europe*, (pp. 1584-1589).

Huang, L., Yuan, F., & Xu, Q. (2009). Lifetime Reliability-Aware Task Allocation and Scheduling for MPSoC Platforms. *Proceedings of IEEE Design, Automation and Test, Europe*, (pp. 51-56).

Humenay, E., Tarjan, D., & Skadron, K. (2007). Impact of Process Variations on Multicore Performance Symmetry. *Proc. Conf. Design, Automation and Test in Europe*, (pp. 1653-1658).

ITRS. (2012). Retrieved from The International Technology Roadmap for Semiconductors - 2012 Update: http://www.itrs.net/Links/2012ITRS/Home2012.htm

Jeun, W.-C., & Ha, S. (2007). Effective OpenMP implementation and translation for multiprocessor system-on-chip without using OS. *Asia and South Pacific Design Automation Conference, ASP-DAC*, (pp. 44-49).

Kang, K., Park, S., Roy, K., & Alam, M. (2007). Estimation of statistical variation in temporal NBTI degradation and its impact on lifetime circuit performance. *ICCAD 2007: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, (pp. 730-734).

Karl, E., Blaauw, D., Sylvester, D., & Mudge, T. (2008). Multi-mechanism reliability modeling and management in dynamic systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems , 16* (4), 476-487.

Krishnan, A., Reddy, V., Chakravarthi, S., Rodriguez, J., John, S., & Krishnan, S. (2003). NBTI impact on transistor and circuit: models, mechanisms and scaling effects. *Technical Digest. IEEE International Electron Devices Meeting*, (pp. 14.5.1–14.5.4).

Kumar, S., Kim, C., & Sapatnekar, S. (2006). An analytical model for negative bias temperature instability. *ICCAD 2006: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, (pp. 493-496).

Lemaire, R., Thuries, S., & Heiztmann, F. A flexible modeling environment for a NoC-based multicore architecture. *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, (pp. 140-147). Huntington Beach, CA, USA.

Marongiu, A., & Benini, L. (2009). Efficient OpenMP support and extensions for MPSoCs with explicitly managed memory hierarchy. *Proceedings of the 12th International Conference on Design, Automation and Test in Europe*, (pp. 809–814).

Mulas, F., D.Atienza, Acquaviva, A., Carta, S., Benini, L., & De Micheli, G. (2009). Thermal Balancing Policy for Multiprocessor Stream Computing Platforms. *In Transactions on Computer-Aided Design of Integrated Circuits And Systems , 28*.

Mutyam, M., Wang, F., Krishnan, R., Narayanan, V., Kandemir, M., Xie, Y., et al. (2009). Process-Variation-Aware Adaptive Cache Architecture and Management. *IEEE Transactions on Computers , 58*, 865-877.

Ndai, P., Bhunia, S., Agarwal, A., & Roy, K. (2008). Within-Die Variation-Aware Scheduling in Superscalar Processors for Improved Throughput. *IEEE Transactions on Computers , 57*, 940-651.

OpenACC-Standard.org. (2013, June). The OpenACC Application Programming Interface, Version 2.0.

Oracle Corporation. (2013). *Java Platform, Standard Edition 7 API Specification - Future Interface*. Retrieved from http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html

Palermo, G., Silvano, C., & Zaccaria, V. (2009). Variability-Aware Robust Design Space Exploration of Chip Multiprocessor Architectures. *Proceedings of the IEEE Asia and South Pacific Design Automation Conference*, (pp. 323-328).

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

Paterna, F., Acquaviva, A., Papariello, F., Desoli, G., & Benini, L. (2009). Variability-Tolerant Workload Allocation for MPSoC Energy Minimization under Real-Time Constraint. *Proc. IEEE Workshop Embedded Systems for Real-Time Multimedia*, (pp. 134-142).

Paterna, F., Acquaviva, A., Papariello, F., Desoli, G., Olivieri, M., & Benini, L. (2009). Adaptive Idleness Distribution for Non-Uniform Aging Tolerance in Multiprocessor Systems-on-Chip. *Proc. IEEE Conf. Design, Automation and Test in Europe*, (pp. 906-909).

Rebaud, B., Belleville, M., Beigne, E., Robert, M., Maurine, P., & Azemard, N. (2009). An Innovative Timing Slack Monitor for Variation Tolerant Circuits. *Proc. IEEE Conf. IC Design and Technology*, (pp. 215-218).

Schoeberl, M. (2007). A Profile for Safety Critical Java. *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC '07*, (pp. 94-101).

Sylvester, D., Blaauw, D., & Karl, E. (2006). Elastic: An Adaptive Self-Healing Architecture for Unpredictable Silicon. *IEEE Design and Test of Computers*, *23*, pp. 484-490.

Teodorescu, R., & Torrellas, J. (2008). Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. *ACM SIGARCH Computer Architecture News , 36* (3), 363-374.

The jQuery Foundation. (2014). *The jQuery API Documentation - Deferred*. Retrieved January 2014, from http://api.jquery.com/jQuery.Deferred/

The Object Management Group. (2011, June). *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Retrieved from http://www.omg.org/spec/MARTE/1.1/PDF/

Tiwari, A., & Torrellas, J. (2008). Facelift: Hiding and Slowing Down Aging in Multicores. *Procceedings of the IEEE/ACM International Symposium on Microarchitectures*, (pp. 129-140).

Verghese, N., Rouse, R., & Hurat, P. (2008). Predictive Models and CAD Methodology for Pattern Dependent Variability. *Proceedings of the IEEE Asia and South Pacific Design Automation Conference*, (pp. 213-218).

Weilkiens, T. (2011). *Systems engineering with SysML/UML: modeling, analysis, design.* Burlington, MA, USA: Morgan Kaufmann.

Xilinx Corporation. (2014, January). *Zynq-7000 All Programmable SoC*. Retrieved from http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/

Zhang, L., Bai, L., Dick, R., Shang, L., & Joseph, R. (2009). Process Variation Characterization of Chip-Level Multiprocessors. *Proceedings of the ACM Conference of Design Automation*, (pp. 694-697).

11 KEY TERMS AND DEFINITIONS

Code Generation – The automatic generation of software (or another form of computer input language) from a higher-level description. Used to accelerate development by reducing the effort required by developers, and reducing the possibility for errors.

Embedded System – A generic term for a computer system that is part of a larger system. Unlike a desktop or laptop computer, an embedded system will operate either wholly or partly as a component of a larger device, for example an aeroplane or car. Embedded systems are generally size, cost, and power constrained.

Guardband – In semiconductor manufacture, one way of accounting for uncertainty in the design and manufacturing process is to weaken the guarantees on certain design criteria (such as power consumption or minimum clock speed). This weakening creates a 'margin of error' known as a guardband.

Model-Driven Engineering – A development process which makes use of high-level abstract models to aid development and communication between team members, rather than focussing solely on the creation of software and hardware.

SysML - Systems Modelling Language. A general-purpose modelling language for systems engineering applications. SysML supports the specification, analysis, design, verification and validation of a broad range of systems.

Technology Node – A term used in semiconductor device fabrication to describe the size of the features in the finished product. Quoted in terms of nanometres (or larger for earlier nodes), the node name refers to half the distance between identical features in a memory cell. However, for many process nodes this is not a precise measurement and should be understood to be indicative only. Smaller nodes are more recent.

Variability – The observation that features in fabricated silicon devices that were designed as identical will not be identical after manufacture. A wide range of effects contribute to variation in the features' power consumption, maximum clock frequency, and lifespan.

Yield – In semiconductor manufacture, after manufacture and testing, the ratio of products which meet their designed specification against the total number produced. A high yield is important to ensure minimal wastage and a cost-efficient design.