ToucHMore Toolchain and System Software for Energy and Variability Customisation

Neil C. Audsley
Dept. Computer Science
University of York, UK

Ian Gray
Dept. Computer Science
University of York, UK

Andrea Acquaviva
Dept. of Control and Computer Engineering
Politecnico di Torino, Italy

Ralph Haines
ATEGO
UK

***Abstract*—Run-time platform variability presents a number of challenges to the system software in order that a run-time environment is presented to applications that sufficiently masks dynamic platform variability (including fabrication variability), whilst allowing applications to tune overall system performance to exploit key aspects of dynamic energy usage and platform variability. The approach taken within the Touchmore project is to model key aspects of the platform in order that performance and variability can be understood and exploited by the system software. In turn, the system software (comprising OS and run-time) utilises the model so that aspects of variability and energy usage are abstracted from the platform, then monitored and controlled in order to meet policy goals, eg. energy minimisation. This paper documents aspects of the modeling and system software structure to show how the Touchmore project is managing energy and platform variability using customisation of the application, system software and toolchain.**

I. Introduction

There is great potential variation between different heterogeneous MPSoC architectures and platforms: e.g. different combinations of CPUs, DSPs, accelerators, together with interconnect, communication and differing memory hierarchies. Such a variance poses a challenge to the software toolchains used to map applications to an MPSoC. It is clear that such toolchains need to be customisable, so that they can easily target a different MPSoC with minimal effort. In addition, static and run-time variability within the MPSoC itself is becoming more evident as technologies move towards and beyond 22nm [1], [2]. This requires that the code produced by the toolchain must be amenable to the dynamic changes needed at run-time, eg. offloading operations to different cores for energy reduction purposes. Such dynamic behaviour requires support by a run-time and operating system structure that can monitor the status of the platform and make dynamic decisions regarding where computation is carried out within the platform.

Programming heterogeneous MPSoCs cannot currently be handled entirely at the programming language level due to the gap between the programming model and the platform. For example, 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-standard memory address (NUMA) spaces and non-standard communications (ie. not necessarily via shared memory). This results in a mismatch between the programmers conceptual model and the underlying implementation. In addition, programming languages do not allow applications to control easily where computation is actually executed within the MPSoC – there is minimal control over the mapping of the application to the architecture.

One approach to solving these issues is to extend or modify existing programming languages to better address the needs of MPSoC platforms. For example, Javas Real-Time Specification [3] (RTSJ) and POSIX allow basic mapping of threads to processors and to model physical memory, but as described above a largely homogeneous architecture is still assumed. Developers must rely on language extensions or extra-linguistic techniques (e.g. custom tools and linker scripts) to fully exploit complex hardware. New programming languages are often proposed as a way of solving this problem. Languages exist to target complex memory systems [4], highly-parallel architectures [5] and many others. However, until these become de facto standards they are unlikely to be adopted by industry, and their new languages and tools are a barrier to certification.

Toolchain approaches to programming MPSoC systems (rather than extending programming languages) depend upon abstracting the complexities of the platform whilst providing sufficient power for the programmer to control pertinent parts of the architecture directly. Techniques include Compile-Time Virtualisation (CTV) [6], [7], [8] which offers source-to-source translation to provide the benefits of virtualisation without imposing a large runtime layer as its virtualisation is only applied at compile-time. This technique has found some success in an industrial context [9], but does not attempt to solve issues regarding variability in MPSoC platforms.

Whilst efforts are being made to extend programming languages to better target the basic architecture of MPSoC platforms, the increasing static and dynamic variability introduced into MPSoC chips by current and future manufacturing process is not being addressed at the language level. Essentially, there is increasing variability in the capabilities of individual chips - eg. due to fabrication variability the CPUs within the chip may have different maximum clock speeds. Also, embedded MPSoCs are frequently battery powered, and may need to reduce their energy usage or thermal output (if cooling is an issue). These issues can be handled using techniques such as clock gating, power gating, dynamic frequency and voltage scaling (DVFS), offloading processing to DSP cores, and

software that reacts to the static and dynamic variability of the platform. Building awareness of such variability into the run-times, operating systems and toolchains is a key challenge.

Within this paper we consider targeting MPSoC platforms that exhibit both static and dynamic behaviour due to manufacturing and run-time variability. We propose a platform modeling and customisable software toolchain designed to bridge the gap between the idealistic platforms assumed by programming languages and the complex reality of todays MPSoC; it allows new platforms to be targeted with minimal effort within the platform modeling. This supports industry because no change is required in the toolset to target a new platform, unlike today where new development processes, modeling processes and tools must be built. The platform modeling and toolflow are currently being developed within ToucHMore [10] – an EU Seventh Framework research project focussed on using a model-driven approach to capture the key characteristics of a heterogeneous Network-on-Chip (NoC) based platform in order that the toolchain may be customised to meet its specific requirements. The particular NoC targeted by ToucHMore is GENEPY [10] a heterogenous architecture containing a number of different types of CPU, together with DSPs, and a non-uniform memory architecture.

The remainder of this paper considers the broader aspects of customisation in section II, including discussion of customisation within the ToucHMore toolchain. Section III considers the overall toolchain, including specific aspects of the platform modeling undertaken. The system software stack is described in section IV, with conclusions offered in section V.

II. Customisation

Software and the toolchain (code generators, compilers, linkers etc) can be customised in a number of ways in order to accommodate the variability in MPSoCs in terms of their static architecture and dynamic run-time behaviour. Generally, customisation can occur in the following main places:

- *Software level* – by means of software design, including choice of algorithm (e.g parallel when mapping to a parallel platform), structure of loops (to exploit data-level parallelism); by re-factoring the software to better map to a particular platform.
- *Compiler level* – by compiling code (which may be designed without customisation in mind) in a way that customises it (optimises it) for a specific platform, or non-functional characteristics (eg. low-energy).
- *Mapping level* – the placement of application functions within the architecture (eg. on CPU, or DSP or synthesised to hardware) has significant impact on the overall performance of the system.

Usually, the customisations above are usually applied offline. However, all can be applied dynamically to react to platform changes. At the software level, dependent upon where it is being executed, a software component may choose an expensive algorithm (for a fast CPU) or an inexpensive algorithm (for a slow CPU), with a clear trade-off in terms of performance. Likewise, code can be recompiled at run-time for a different

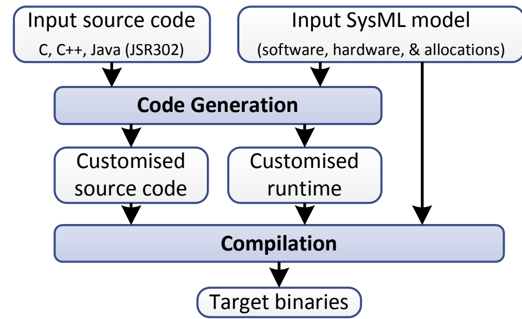

Fig. 1: Customisation within the ToucHMore Toolflow

platform, or for interpreted languages such as Java the bytecodes can be assembled to native code (e.g. if a frequently executed hotspot is detected).

All of the dynamic approaches need support from system software (ie. run-time and / or operating system), and the MPSoC platform itself. Essentially, the platform provides status (e.g. for power, CPU frequencies) which is monitored by the system software, which can then modify characteristics of the platform (eg. turn off a CPU) and the software (eg. by reallocating application software to a different component). The precise details of what status information platform monitoring reveals to the system software vary between MPSoCs, and are not discussed within this paper. However further details of the customisations can be made by the system software are now discussed.

Much of the customisations made by the system software relate to where application functionality is executed, how fast (ie. clock speed) and how many resources are committed (including memory). The initial mapping of application to architecture components as defined within the model can be adapted by the system software in response to the platform status at run-time, or to better fulfil system goals (e.g. energy minimisation). The main issue is the granularity of component moved, which can vary from small functions that are moved to DSPs / GPUs at run-time as accelerators; to complete software components. The system software must ensure that after reconfiguring the system in this way, provision remains for the software to still access the communication paths and memories it requires. This is relatively straightforward if all such accesses occur via the system software, although can reduce performance marginally.

A. Customisation with the ToucHMore Project

Customisation is a central concept to the ToucHMore project. An overview of the toolchain is given in Figure 1, where a UML / SysML model of the platform is used to hold the key platform information necessary to customise the application source code, run-time and operating system (OS) for the platform. The model also contains details of the mapping of application software components to the target platform, so that necessary interface and communications code can be generated (communications handled via the runtime); and also details of the expected static and dynamic

variability in the platform so that policies within run-time can be customised to the characteristics of the platform. The input software may be provided as C, C++ or Java.

Customisation occurs in a number of places. User software (eg. methods that can be offloaded to DSPs) is rewritten to call the appropriate functions of the generated ToucHMore run-time. User software is compiled appropriately for the target selected – this could be for a range of CPUs within the architecture. The customised source code and runtime are then compiled using a compilation process that is itself aware of the nascent platform variability as expressed within the model. Based on the architecture and allocations, the compiler will produce different sets of binaries, marshal data for intracore communications, and make use of DSPs, DMAs and shared memory as available. Customisation can also occur via dynamic compilation. For example, compilettes [11] is a low-level compiling technique based on a minimal code generator with parametric embedded sections to generate binary code at run-time.

A key part of the ToucHMore customisation approach is to identify functionality within the application software whose methods can be moved to a different CPU or DSP at run-time under the control of the system software. An annotation is used to identify such functions – @offload. The semantics within ToucHMore are that the offloaded function is executed synchronously. Within a Java context, the offloaded method is a pure function (which also aids offloading of a software function to a hardware implementation of that function in hardware [12]). Also, it:

- should be non-recursive;
- should not call other offloaded functions;
- have a fixed number of arguments;
- have non-aliased parameters.

The offloaded function should not use the object model of Java, hence:

- should be a static method;
- should only have primitive types, array of primitive types, or streams of primitive type parameters;
- should not make dynamic memory allocation;
- should not use synchronisation;
- should not throw or catch exceptions.

ToucHMore has defined @offload for the Java programming language, using Java annotations. Within Java, different language constructs can be annotated: classes, fields, methods, method's parameter, and variables. The ToucHMore @offload should only be used for methods (ie. functions). An example is shown in Figure 2.

III. Toolchain Overview

The toolchain is shown in Figure 3. The platform is modeled (section III-A) with sufficient detail that the application software and system software can be customised to the platform as required. The model can also provide two set of implementation goals:

- *Budgets* – limits can be placed on the power usage, memory usage, and execution time of a given task or offload function. These are not guaranteed to be enforced by the runtime, or toolchain, they are for design and analysis. They are, however, available to the other phases of the toolchain for guidance.
- *Goals* – The following goals can be applied to guide the implementation process. Again, these are not guaranteed to be followed, or result in an optimal implementation:
  - *Tasks* – A priority level indicates which tasks gain precedence over other tasks in a low-power situation.
  - *Offload functions* – A priority level indicates which tasks gain precedence over other offload functions in a low-power situation.

```
/**
 * This program computes 1024 random value and compute the average value.
 */
public class OffloadExample {

      public static final int N_DATA = 1024;
      public static final short[] dataArray = new short[N_DATA];
      public static final int[] result = new int[1];

      public static void main(String[] args) {
            for(int i=0;i<dataArray.length;i++)
                  dataArray[i] = (short)i;

            sum_data(dataArray, result);

            if (result[0]!=523776) {
                  System.exit(-1);
            } else {
                  System.exit(0);
            }
      }

      @Offload
      public static void sum_data(@In short[] data, @Out int[] result)
      {
               int sum = 0;
               for(int i=0;i<data.length;i++)
               {
                 sum += data[i];
               }
               result[0] = sum;
      }
}
```

Fig. 2: ToucHMore @offload Annotation Example

Further details of the toolchain include:

- Software is developed in Java or C/C++. Java programs are translated with ATEGO's Perc Pico tool from Java into C, and linked against the Perc Pico runtime libraries [13]. These libraries form a minimal footprint Java run-time designed for hard real-time systems, hence is predictable in terms of time and space usage. In the translation of Java to C, Perc Pico maintains annotations so that they are available for the C compiler and associated tools.
- Along with the software, a model is presented from Artisan Studio UML tool that describes the hardware, software, the allocations between them, and provides budgets and goals.
- The C software with annotations, runtime libraries, OS libraries are provided to the Architecture Optimisation phase. The Architecture Optimisation phase is optional and can be used in a pass-through mode – useful if all mappings of application software to platform CPUs is known and defined in the model. If it is invoked, the Architecture Optimisation phase will iteratively adjust

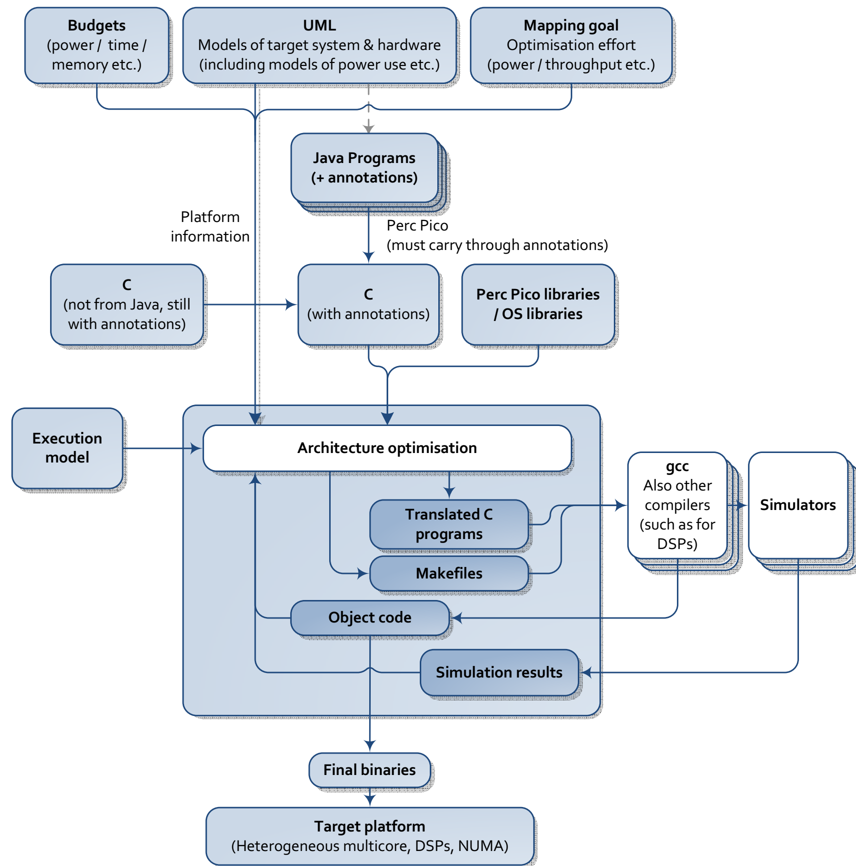

Fig. 3:ToucHMore Toolflow

the mappings of the system to better target the provided budgets and goals. It will use analysis, compilation, measurement, and simulation to extract information about the expected performance of the system given the actual architecture and target mappings. The adjustments it may include changing the mappings of software components to CPUs, moving the targets of offloading annotations (e.g. making a general offload location more specific, or moving a specific offload location to free up a congested hardware resource); removing offloading to allow some resources to be powered down.

The final phase of the toolchain is to emit the binaries of the implemented application, ToucHMore run-time and operating system.

A. Platform Modeling

ToucHMore has adopted a UML/SysML approach to modeling the target platform, utilising the Artisan Studio tool. Separate models are used to represent:

- *Target platform specifics* – consist of the hardware structure and also hardware capability information (e.g. platform energy and variability management).
- *Application specifics* – application structure and behaviour, including type of CPU / DSP to execute upon.
- *Deployment specifics* – mapping of an application to a target platform.

A number of hardware types are contained within the model (eg. Figure 4) to build a platform model. Abstract types require concrete subtypes to be defined – eg. a specific CPU core. Capabilities of the platform to provide status monitoring (eg. clock frequency scaling, power monitoring) are also included. These can be used for subsequent customisation of the toolchain and system software.

The Block Definition Diagram (BDD) is shown in Figure 5. Again, hardware capabilities are expressed. The top level model for the whole Genepy MPSoC is shown in Figure 6.

IV. System Software

The general system software structure is illustrated in Figure 7, over an abstraction of the Genepy MPSoC, where each MIPS will run a full system software stack; the other (DSP) will run a minimal stack applications will access these devices via offloaded computation. The system software is in three layers:

1) *ToucHMore run-time* – this is customised for the target architecture. The API layer is visible to the user program and includes variability calls that support power scaling, power-aware algorithms. DSP offload etc.

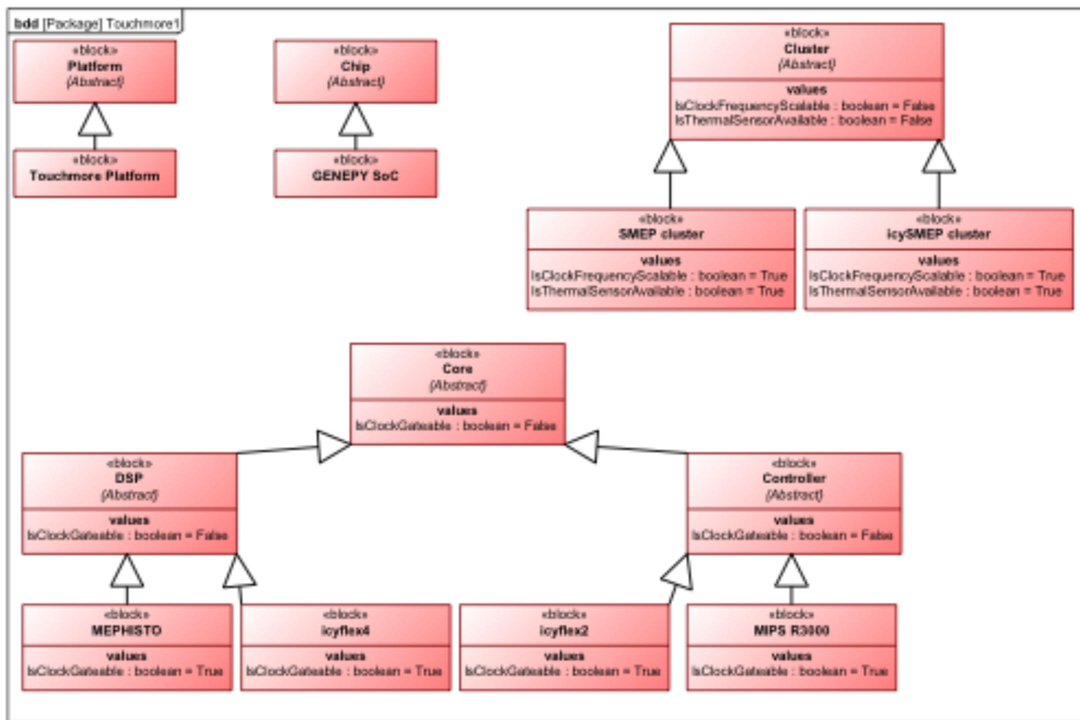

Fig. 5: Block Definition Diagram for the Genepy Chip

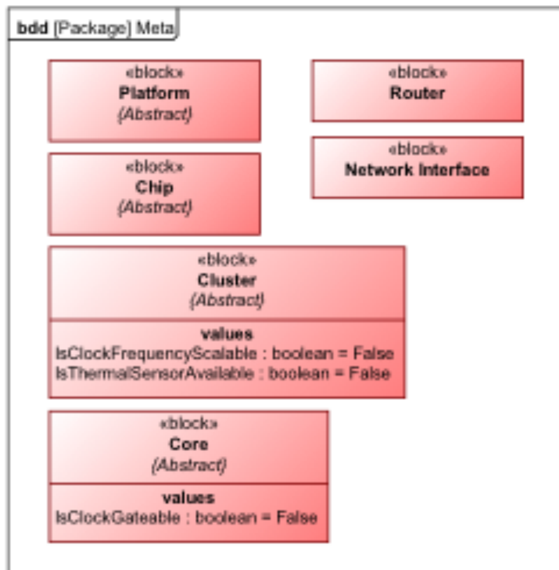

Fig. 4: Top Level Hardware Types

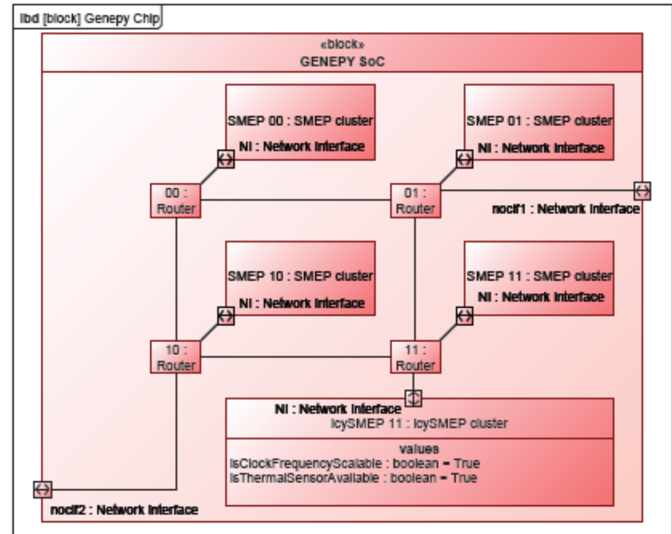

Fig. 6: Top-Level Model of Genepy Chip

2) *Operating System* – the OS layer is a minimal kernel (FreeRTOS) that supports the features of the user software.
3) *Communications* – The communications layer allows the OS layer to communicate with other OSes on other cores of the architecture to provide system- wide services. It is based on the Multicore Associations MCAPI and MRAPI APIs.

The system software is discussed further in the following sections.

A. The ToucHMore API

The system software combines to create the ToucHMore API. This API serves the following purposes:

- Provides a consistent interface for the ToucHMore compilers and toolchain to generate software for.
- Defines a consistent set of services which are available for the application programmer.

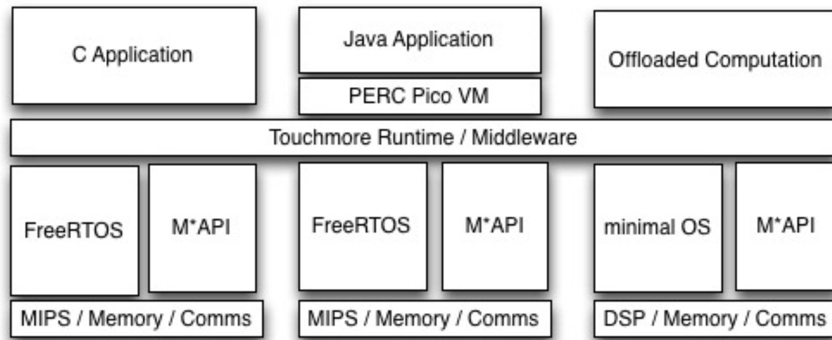

Fig. 7: Structure of the ToucHMore System Software

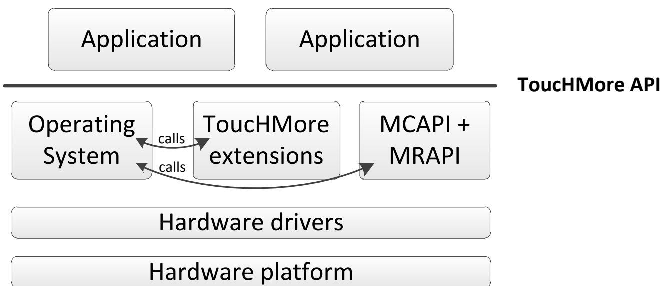

Fig. 8: Structure of the ToucHMore API

- Enumerates a consistent set of features that are to be implemented by the hardware platform.

The full contents of the API will be specified in detail in a future document, but the following overview is provided to enumerate the broad services required:

- *Multiprogramming* – ToucHMore applications are composed of a set of concurrent, coordinating threads (or tasks) (see section IV-C for further details)
- *Communications* – Provides on-chip, inter-thread communications (see section IV-B for further details)
- *Shared resources* – Provides mutual exclusion, shared memory, and synchronization primitives to the threads of the system (see section IV-B for further details)
- *Low-level OS calls and hardware drivers* – where appropriate to interface to specific elements of hardware and I/O.

In addition, the ToucHMore run-time provides facilities to accommodate variability. Two types of variability are considered, fabrication variability and operational variability. Fabrication variability is characterised as variations between fabricated instances of the same artefact and may affect issues such as power consumption, heat dissipation, or maximum clock frequency. This variability is quantified after the system is built. Operational variability concerns the changing environment and the changing state of the system as it is operating, i.e. changes in temperature, damage due to radiation, or silicon degradation though the lifetime of the device. The ToucHMore run-time includes facilities to make decisions as the system is executing in response to variability. Variability awareness may effect the following changes in the system, and these may be applied statically at design-time or dynamically at run-time:

- *Change software to hardware mappings. i.* – i.e. move software to different processing cores in the system; store data in different memory locations.
- *Offloading computation to acceleration hardware* – functions may be offloaded to a DSP or similar if supported by the target platform (i.e. those identified by the @offload annotation).
- *Change offloading parameters* – when a function is offloaded to a hardware accelerator (or another core) the decision of where to offload it to must be made. This may involve "offload anywhere" or more specific versions that limit the offload a given location, a set of locations, or any of a type of target DSP.
- *Change scheduling parameters* – i.e. Increase the priority of some threads or affect the scheduling policy used on some cores.
- *Power down* – signal that a given hardware feature can be powered down to save power.

In order to implement variability policies, the ToucHMore API defines a set of variability metrics that are made available to the ToucHMore system software at runtime. These metrics are also made available to the application software through the API to allow the programmer to develop variability-aware applications if required.

B. MCAPI and MRAPI Overview

The class of hardware platforms targeted by the ToucHMore project are multicore embedded systems. It is necessary for such platforms to provide mechanisms for on-chip communication, coordination, mutual exclusion and the provision of shared resources. Therefore, the ToucHMore API includes these features by utilising existing de facto standard APIs from the Multicore Association. The Multicore Association (MCA) [14] was founded in 2005 as an industry forum for establishing a set of application programming interfaces (APIs) to be supported by industry on multicore technologies. The MCA has a number of key industrial members and is widely seen as an important contributor to multicore programming

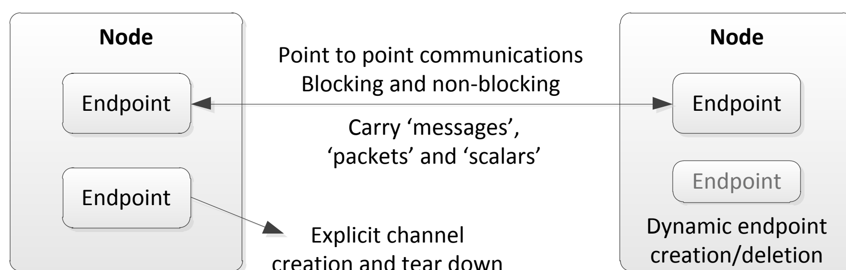

Fig. 9: Structure of the MCAPI API

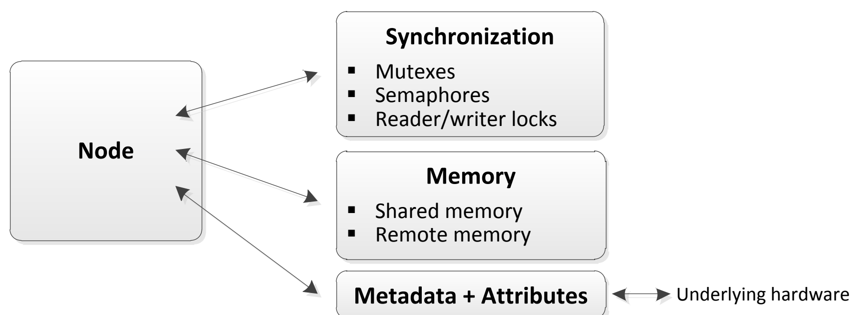

Fig. 10: Structure of the MRAPI API

practice. The MCA APIs have the following advantages over competing technologies with regard to their implementation in the ToucHMore project:

- The MCI APIs are designed from the start to be very lightweight. Unlike an API like POSIX that defines thousands of features with precise behaviour, the MCI APIs are all relatively small and aim to provide a minimal useful subset of features. This is very important in the ToucHMore project where memory use and efficiency are key concerns.
- Unlike dynamic, runtime-oriented systems like CORBA and MPI that target variable platforms, the MCA APIs are more static and require less runtime support. The architectures of embedded systems do not tend to change after deployment, and the MCA APIs reflect this.
- Unlike a self-defined API, the MCA APIs are published standards with example implementations. This aids integration of existing software and tools, and can assist cooperation with other projects and groups though the provision of a common API.

The APIs used in ToucHMore are in two main areas: communications and resource management. Communications, via the MCAPI API [15], is intended to support communication within a closed distributed system - e.g. a multicore system-on-chip. Resource management, via the MRAPI API [16], is intended to support and coordinate shared memory management and synchronisation (eg. mutexes, semaphores). Both standards have been released as de-facto industry standards and are summarised in the following sections.

1) MCAPI Structure: MCAPI defines a communication API for closely coupled systems where there are multiple cores connected in an arbitrary topology with no requirement for symmetry. MCAPI enables point-to-point communication between endpoints that are associated with nodes. An MCAPI node is a logical notion that can be a process, a thread, an OS instance, or hardware component (accelerator, CPU, DSP core etc.). A node can contain several MCAPI endpoints and endpoint identifiers are unique within the system, named with a tuple $< node_id, port_id >$. Endpoints can have a set of attributes describing quality of service features, buffer capabilities and timeouts.

After MCAPI initialisation, a connection is created and established between two endpoints. Then both sender and receiver can open a channel between those endpoints. Once a channel is established, blocking and non-blocking send and receive functions are used to pass data over the channel between the two nodes. Data to be sent is stored in an application buffer and passed to an MCAPI send endpoint. When there is space at the receiver, the data is sent to the receiving endpoint where it is stored in a FIFO buffer for subsequent application use. The overall structure of the MCAPI API is shown in Figure 9.

2) MRAPI Structure: MRAPI defines a set of services for multicore systems that aim to provide the essential capabilities required for managing shared resources in the same multicore, embedded environments as MCAPI. The overall structure of the MRAPI API is shown in Figure 10.

MRAPI is designed to operate alongside MCAPI (although the two APIs are independent and can be implemented separately). It has the same system model of nodes, which can create resources and share them with the other nodes of the system. The resources provided by the API are:

- *Synchronization primitives* – Similar to POSIX, MRAPI provides mutexes, semaphores, and reader/writer locks that operate over complex embedded architectures.
- *Shared memory* – MRAPI allows the definition of shared memory regions that are available to multiple nodes to allow low-overhead sharing of data, when supported by the underlying hardware.
- *System level events* – System-wide events can be defined to encode information such as changing power-savings states or device failures.

C. FreeRTOS Real-Time Operating System

The operating systems within the ToucHMore system software stack to implement low-level features such as thread management, memory management and device control. As can be seen from Figure 7 and Figure 8, the OS is visible to the application. This is because the ToucHMore API will extend the OS with features that are useful for tackling variability of complex embedded architectures; it will not attempt to reimplement the entire API of all embedded operating systems as this would be infeasible an unsustainable. Some features (for example file I/O or timers) will be accessed directly through the OSs native API.

The FreeRTOS [17] real-time OS is used in the ToucHMore project. FreeRTOS is a real-time operating system designed to be simple and lightweight. It provides basic threading and concurrency primitives, which will be used by the other parts of the ToucHMore API to implement richer features for the application programmer. The scheduler supports pre-emptive scheduling, making it useful for the implementation of real-time systems. FreeRTOS is already ported to a wide range of embedded processors, and the ToucHMore partners are in the process of porting it to the ToucHMore Genepy platform.

For CPUs that do not require a full system software stack (such as the DSP in Figure 7) then a more lightweight OS can be used. This has yet to be fully defined for ToucHMore, although we note that given that the DSP elements within the architecture are to be used solely as the targets for applications to offload computation to, the required OS elements will be very small.

V. Conclusion

The paper has presented some of the work of the ToucHMore EU Framework project, focussing upon aspects of platform modelling and toolchain customisation for MPSoC platforms. Currently, the project is modelling the GENEPY MPSoC platform, and enabling input applications written in Java to be customised to run effectively on the platform. This has required development of a ToucHMore run-time, providing an lightweight abstraction over the complex MPSoC. The run-time is also aimed to monitor the run-time status of the MPSoC, eg. so that computation can be moved for improved overall power usage.

The aims of the ToucHMore project are to evaluate the toolchain targeting both the GENEPY platform, and other platforms. This will establish the efficacy of one of the project goals, namely that a customisable toolchain requires minimal work to target different complex MPSoCs.

Acknowledgment

The authors would like to thank their host institutions and companies, and also the EU 7th Framework programme for providing funding for the ToucHMore project under contract 288166.

References

[1] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (ntv) design: opportunities and challenges," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1153–1158. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228572

[2] T. Miller, R. Thomas, and R. Teodorescu, "Mitigating the effects of process variation in ultra-low voltage chip multiprocessors using dual supply voltages and half-speed stages," *Computer Architecture Letters (CAL)*, 2012.

[3] J. Gosling and G. Bollella, *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[4] K. Fatahalian *et al.*, "Sequoia: programming the memory hierarchy," in *SC '06*, 2006, p. 83.

[5] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[6] I. Gray and N. Audsley, "Exposing Non-Standard Architectures to Embedded Software Using Compile-Time Virtualisation," *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '09)*, 2009.

[7] ——, "Supporting Islands of Coherency for highly-parallel embedded architectures using Compile-Time Virtualisation," in *13th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2010.

[8] ——, "Targeting complex embedded architectures by combining the Multicore Communications API (MCAPI) with Compile-Time Virtualisation," in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, 2011.

[9] 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.

[10] The ToucHMore Consortium, "The ToucHMore Project," http://www.touchmore-project.eu/, 2012.

[11] K. Brifault and H. p. Charles, "Efficient data driven run-time code generation for multimedia applications," in *In: LCR 04: Proceedings of the 7th workshop on Workshop on languages, compilers, and.* ACM Press, 2004, pp. 1–7.

[12] J. Whitham, N. Audsley, and M. Schoeberl, "Using Hardware Methods to Improve Time-predictable Performance in Real-time Java Systems," in *Proc. JTRES*, 2009, pp. 130–139.

[13] Atego, "Perc Pico," http://www.atego.com/products/aonix-perc-pico/, 2011.

[14] The Multicore Association, "http://www.multicore-association.org."

[15] ——, "Multicore Communications API Specification V1.063 (MCAPI)," http://www.multicore-association.org/workgroup/mcapi.php, March 2008.

[16] J. Holt, "Designing an Industry Standard API to Manage Multicore System Resources," http://www.multicore-association.org/webinar/090811_MRAPI.pdf, August 2009.

[17] The FreeRTOS Project, "The FreeRTOS Project," http://www.freertos.org/, 2012.