Explicit Java Control of Low-Power Heterogeneous Parallel Processing in the TouchHMore Project

Ludovic Gauthier
Atego SAS
Toulouse
France
ludovic.gauthier@atego.com

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

Adrian Larkham
Atego Systems
Cheltenham
United Kingdom
adrian.larkham@atego.com

Gasser Ayad
Politecnico di Torino
Torino
Italy
gasser.ayad@polito.it

Andrea Acquaviva
Politecnico di Torino
Torino
Italy
andrea.acquaviva@polito.it

Kelvin Nilsen
Atego Systems
San Diego
CA
kelvin.nilsen@atego.com

ABSTRACT

This paper describes an approach to deploying Java on low-power, low-memory, heterogeneous multi-core systems. A goal of the effort is to enable the use of such systems in applications that must comply with real-time constraints, some of which must satisfy external certification authorities, thus the work is based on Safety Critical Java [4].

The heterogeneous multi-core system-on-a-chip considered have specialized purpose processors that can perform particular computations quickly and with less energy consumption than general-purpose processors. In order to allow a high degree of parallelism, these systems use partitioned memories, as opposed to the uniform memory access model traditionally supported by symmetric multiprocessors and the Java memory model.

The effort is a work in progress. Syntax and tool chains are being developed and experimentation with the technologies has begun. But the current results are considered preliminary as many planned features are not yet fully implemented and performance optimization has not yet been completed. Consistent with the style of multi-core development in standard edition Java, the software engineer is responsible for orchestrating the division of labor between coprocessors.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages, Object-oriented languages, Java; D.2.11 [**Software Architectures**]: Data abstraction, Information hiding, Languages, Patterns; D.2.13 [**Reusable Software**]: Reusable libraries, Reusable models

JTRES'13, October 09–11, 2013, Karlsruhe, Germany.

ACM 978-1-4503-2166-2/13/10...$15.00.
http://dx.doi.org/10.1145/2512989.2513001

General Terms

Design, Modularity, Reliability, Standardization

Keywords

Java, Real-Time, Object Oriented Development

1. MOTIVATION

In recent years, many microprocessor vendors are deploying multiple heterogeneous processor cores on a single chip [11], [16]. For many applications, these heterogeneous system-on-a-chip architectures offer superior performance and lower power consumption than homogeneous SMP architectures [15]. These benefits are especially valuable in battery-powered devices such as mobile telecommunications, remote data gathering, and wireless controller applications. These benefits are also relevant to large server farms, where improved power efficiency leads to huge savings in cooling and electric power costs.

The rationale for bundling multiple different processor architectures onto a single chip is to exploit the strengths of specialized coprocessor systems. For many applications, a specialized signal processor delivers computational results in far less time than is required by a general purpose processor. Furthermore, in comparison to a general purpose processor, the specialized signal processor consumes less space on the chip die and consumes less electric power to perform the same computations [15]. To support optimal power management, it is typical for such systems to allow dynamic clock and voltage adjustments for individual coprocessors, and to allow certain processors to be turned off entirely. When there is work to be done, individual processors are powered up to run at full speed. But during lulls in the workload, processors are powered off or slowed to conserve energy. Programming heterogeneous computer systems is especially difficult for a variety of reasons. Among the typical challenges faced by developers of such systems are the following:

1. In a heterogeneous environment, different processors execute different instruction sets. Thus, the developer is required to obtain cross compilers for each of the tar-

geted architectures, and must manage the assortment of binaries targeted to each relevant architecture.

2. While compilers for high-level languages like Java are available for common general purpose processing architectures such as ARM and MIPS, they are usually not available for the highly specialized digital signal processors that are often bundled with one or more general purpose processors. Typically, the only language available for programming the specialized digital signal processors is C. If programmers desire to use a higher level languages like Java, they can only use it for those parts of the application that run on the general purpose processors.

3. For any given application workload, deciding on the most efficient division of labor between general purpose processors and specialized digital coprocessors requires an understanding of many low-level details that are difficult to ascertain. Furthermore, the most efficient division of labor may depend on dynamic factors, such as how much work has already been assigned to each of the processors. To support the ability to run the same code on different processors at different times, the application developer must compile the code for each processor and must develop control logic to decide at run time where particular computations will be performed.

4. The effort required to carefully engineer a solution that makes efficient use of a heterogeneous platform is highly dependent on the platform configuration, which may be difficult to predict. For most heterogeneous system-on-a-chip architectures, many configuration options exist, with different numbers of general purpose and specialized coprocessors, different clock rates, different memory sizes, and different interconnection speeds. Furthermore, the configuration options may change from year to year. Managing the distribution and evolution of complex software systems across such a bewildering diversity of deployment platforms is extremely difficult without automated support to facilitate the automatic reconfiguration of software for each deployment possibility.

The ToucHMore project explores the use of Java as an enabling technology to support the development of portable and scalable software applications to run on the GENEPY platform [9]. The project has completed specification and investigation phases and is currently progressing through implementation. The first version of the toolchain would mature by the end of 2013. And a full validation of the toolchain over the target application is due by Q1 2014.

The GENEPY architecture is the result of a collaborative research effort by CSEM (Centre Suisse d'Electronique et de Microtechnique) and CEA-LETI (Laboratoire d'Electonique des Technologies de l'Information of the Commissariat a l'Energie Atomique et aux Energies Alternatives). GENEPY is an experimental massively parallel architecture designed to support research on power management and performance.

GENEPY is structured as a two-dimensional mesh of connected clusters. Each cluster consists of control processor(s), signal processor(s), memory, sensors, actuators, and interconnection network as presented in Figure 1.

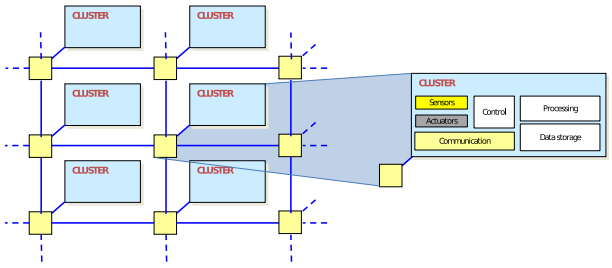

Figure 1: GENEPY Platform architecture

Each cluster is by design simple enough to be implemented as a complete system on a chip (SoC). The ToucHMore project focuses on two particular types of GENEPY clusters, SMEP and icySMEP. The key characteristics of each are detailed in Table 1. The icyflex2, icyflex4, and Mephisto processors use specially designed instruction sets optimized for low-power operation.

Cluster type	Control	Processing	Data storage
icySMEP	icyflex2	2 icyflex4	16KB (control) + 256KB shared (control+processing)
SMEP	Mips R3000	2 Mephisto	32KB (control) + 128KB shared (control+processing) + 2 x 16KB (processing)

Table 1: GENEPY SoC cluster types

This paper focuses on deployment of Java on low-power heterogeneous processors. The emphasis on the severe memory and power constraints of common low-power heterogeneous systems differentiates the ToucHMore work from all related research efforts.

2. THE TOUCHMORE SOLUTION

This section provides a description of the approach we adopt in ToucHMore. The solution is based on a model description of the platform, the application and of the deployment of the application on this platform. Several constructs based on Java annotations have been defined to declare the possible parallelism and offloading points. The deployment diagram is used to configure the runtime that will monitor the platform and decide the optimal dispatching of the work on the different clusters and DSPs.

2.1 Model-Based Development

The ToucHMore model-based development approach uses three interconnecting models: a target platform model, an application model and a deployment model. A target platform model describes a specific heterogeneous multicore platform and is used to generate an XML representation of the platform for use by the ToucHMore runtime. An application

model describes the application structure and behaviour and is used to generate the Java code. A deployment model describes the deployment of the application to the target platform.

Using separate models allows different deployments to be defined for an application targeting a specific target platform. To generate the XML and Java code, a specific deployment model must be selected. The deployment model is used to generate an XML representation of the mapping of the application to the target platform and also TouchMore annotations within the Java code. These identify Java methods that can be offloaded and/or parallelised and execution characteristics. The generated XML is also used to generate the necessary build information for the application.

Figure 2 provides an overview of the use of the models to generate the XML and Java code.

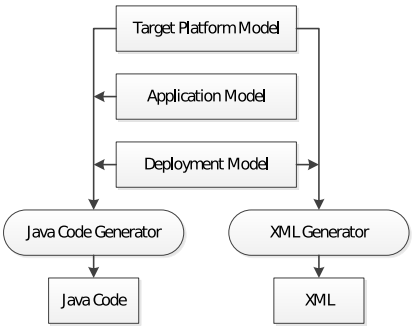

Figure 2: Java Code and XML Generators

Target Platform Model. A target platform model describes a specific heterogeneous multicore platform. SysML is used to model a heterogeneous multicore platform using methodology described in [10]. SysML blocks characterize a set of abstract hardware types for a heterogeneous multicore platform. These include SysML value properties corresponding to the hardware properties of the abstract hardware type. SysML blocks representing concrete hardware types are defined using the SysML blocks representing abstract hardware types when developing a target platform model.

A target platform model describes the hierarchical structure of the heterogeneous multicore platform down as far as the processor cores that execute an application. The hierarchical structure allows individual (instance) processor cores and clusters to be given specific values for hardware properties. Communication connections between hardware elements are also modelled providing information on the communication mechanisms available and their performance.

Figure 3 provides an example of part of the target platform model for GENEPY.

The deployment model is used to generate a XML representation of the target platform for use by the TouchMore runtime.

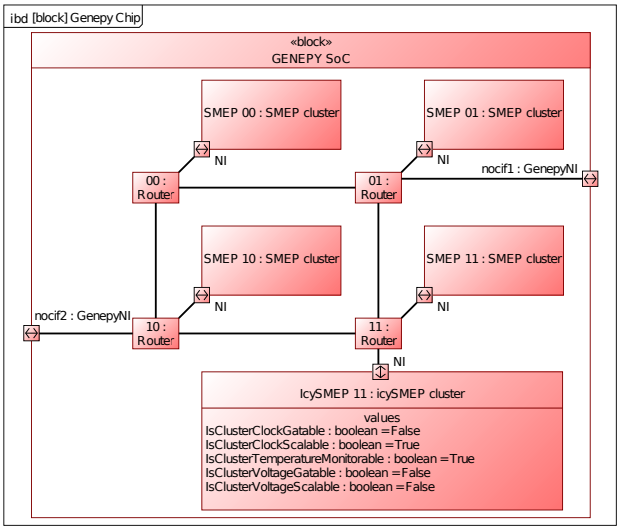

Figure 3: GENEPY system on chip model

Application Model. UML is used to model the application. Class modelling is used to model the structure of a Java application. Java specific information is modelled using the stereotypes and tag definitions provided by a Java profile. The dynamic behaviour of a Java application is either modelled using State Models or Java code is included within the model. Information related to the parallelisation of operations is modelled using stereotypes and tag definitions provided by the TouchMore profile.

Deployment Model. A deployment model describes the mapping of elements (elements can be operations, classes or packages) of an application model to processor core types or instances. It defines which processor core types or instances the generated Java code will be deployed on. A deployment model has a dependency on one application model and one platform model. It contains a set of deployment maps. Each deployment map connects n elements of the application model to m processor core types or instances in the target platform model. A deployment map indicates that the n elements of the application will be deployed on each of the m processor core types or instances. For an operation a deployment map can also specify execution characteristics (minimise power, energy or temperature, worst case execution time and quality of service) and whether it can be offloaded and/or parallelised. Specific stereotypes and tag definitions are provided in the TouchMore profile to support the deployment modelling.

2.2 Safety-Critical Java Subset

Considering the constraints of the GENEPY platform, most specifically the memory size constraints, we decided to rely on a minimal subset of the Java platform based on Safety Critical Java Technology (SCJ) [4].

Safety-Critical Java (SCJ) technology, based on the Real-Time Specification for Java (RTSJ) [7], has been designed to address the general needs of adapting Java technology for use in safety-critical applications. As Java has matured, it has become increasingly desirable to leverage Java technology

within applications that require not only predictable performance and behavior, but also high reliability. The adopted execution model consists of deploying one SCJ virtual machine on each cluster that uses Java.

SCJ has a number of advantages:

- No garbage collection. Scoped memory replaces the Java heap for all temporary memory allocation. The absence of garbage collection makes the runtime smaller and simplifies the potential memory sharing between control CPU and DSPs.
- Real-time execution model. SCJ supports real-time scheduling with full threading and synchronization capabilities in level-2 SCJ applications.
- Hardware support. SCJ supports the possibility to directly access hardware memory and program interrupt handlers. These capabilities are especially useful in the programming of the GENEPY platform's network-on-the-chip services.
- Compatible with a static compilation model. It was difficult to consider the fully dynamic just-in-time compilation traditionally used in Java for the four different kinds of hardware architectures that comprise the GENEPY platform. On the other hand, byte code interpretation would not be efficient enough. A static compilation of Java bytecode into C language and then machine code allows for efficient code generation ahead of time. Atego Perc Pico technology has been used with adaptation of the virtual machine runtime.

2.3 @Offload Annotation

When considering Java execution on heterogeneous system, one of the most closely related previous efforts is the Hera-JVM project [24]. This effort targeted the Cell processor architecture, the high-peformance heterogeneous computer that is the heart of the original Sony PlayStation. Unlike ToucHMore, Power conservation was not a focus for the Hera-JVM effort, and the Cell Processor platform had enough memory to deploy full Java on both the main controller (a PowerPC) and on each of the system's six digital signal processing accelerating cores. Also, the Hera-JVM project did not endeavor to support compliance with real-time constraints or certification by government oversight agencies. Of particular interest is the performance improvement that was reported. By off-loading certain computations to digital co-processors, speedups of up to 13 fold over the speed of doing the full computations on the main control processor were demonstrated on certain benchmarks. ToucHMore intends to demonstrate similar performance gains, once the full tool chain is implemented and integrated.

In ToucHMore, a Java annotation named `@Offload` has been defined in order to identify Java code that may be offloaded to hardware accelerators. This annotation is similar to the notion of codelet defined by OpenHMPP [8].

The principle is that the `@Offload` annotation does not change the semantics. It just changes the runtime behaviour. So a compiler can safely ignore the `@Offload` annotation if it doesn't support it or if no hardware accelerator is available.

Figure 4 illustrates the threading behavior for off-loaded execution. The control thread blocks until the off-loaded computation is completed. In many cases, the control CPU will perform a context switch to allow another thread to run while the first awaits completion of its off-loaded computation. Not shown in this illustration is support for parallel execution of the off-loaded work. In some cases, the off-loaded computation is performed in parallel on multiple DSPs. This is discussed in Section 2.4.

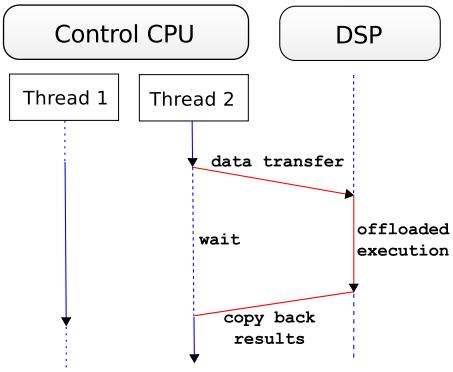

Figure 4: Offload code execution

A number of restrictions are required for off-loaded methods. These restrictions simplify the run-time model and its implementation. The simplifications were necessary in part due to the severe memory constraints on the GENEPY platform. The basic principle is that an off-loaded function should behave as if it is equivalent to a pure function. We also ensure that the execution will be possible on the hardware accelerator which does not necessarily have access to the host memory or to the Java runtime. Off-loaded methods fully respect the Java memory model.

An method annotated with `@Offload` is required to:

- be static
- not refer to any global variables (static fields)
- have a fixed number of arguments
- only have primitive type or array of primitive type arguments which are assumed to be non aliased
- not make dynamic memory allocation
- not use synchronization
- not throw or catch exceptions
- not be recursive
- not call an `@Offload` method
- only call methods that respect the same restrictions

The `@Offload` annotation has `id` and `target` attributes. The `id` attribute associates a globally unique integer identity with the method. This identity is used by the ToucHMore runtime. The unique identity is automatically generated from

the model. The `target` attribute identifies the possible offloading targets for this method. This guides the compilation process, assuring that versions of the method are compiled for each of the signal processors on which the method may be expected to execute.

We also defined three additional annotations: `@In`, `@Out`, and `@InOut` to qualify the array parameters of `@Offload`-annotated methods. These annotations allow the optimization of data copying at method invocation and return. `@In` array arguments are only copied on invocation, and `@Out` array arguments are only copied on return from the `@Offload` method. Primitive type parameters are passed by copy and can only be `@In` parameters. Figure 5 presents the definition of these annotations.

```
@Target({ElementType.METHOD})
public @interface Offload {
   public String[] targets() default {};
   int id() default 0;
}

@Target({ElementType.PARAMETER})
public @interface In {}
@Target({ElementType.PARAMETER})
public @interface Out {}
@Target({ElementType.PARAMETER})
public @interface InOut {}
```

Figure 5: @Offload annotation definition.

2.4 @Parallel Annotation

There are a number of existing approaches to parallelism in the Java language. Aside from explicit threads, Java 7 introduced the fork-join framework [20] which provides a way of decomposing large amounts of work into concurrently executing sub-jobs, the results of which are automatically collected together. The purpose of this framework is to help the programmer to make use of all available processing resources. The general format of a fork-join algorithm in the Java 7 fork-join framework is as follows:

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

Fork-join jobs can be evaluated by a thread pool from the Executor framework in order to bound the actual number of threads that are used, but as can be seen from the structure of the algorithm the number of created jobs is dependent on the input data.

In addition to this, the imminent release of Java 8 will include a wide range of constructs for fine-grained parallel programming [18]. The Collections package has been augmented with parallel collections (i.e. the `ParallelArray` class) that support operations from functional programming such as map and filter. Also, the classes of the `java.util.streams` package allow the definitions of data streams for lazy evaluation of parallel collections.

Outside of the official Java frameworks, there are a number of existing approaches for providing Java programmers with fine-grained parallelism. For example, JOMP [14] and Pyjama [17] both provide OpenMP-like directives for Java. JOMP is the first known project to have done a Java implementation similar to OpenMP [5]. JOMP consists of a pre-compiler that parses Java source code with special directives and generates parallel Java source with calls to a run-time library. This project is not open source and is no longer maintained.

JaMP (OpenMP/Java) [2] is another Java implementation of OpenMP. JaMP supports the full OpenMP 2.0 specification and some aspects of OpenMP 3.0. JaMP is an open source project and is much more actively maintained than JOMP. JaMP depends on full Java.

Jconqurr [3] is a multi-core programming toolkit for Java composed of a set of Java annotations that describe what can be parallelized in a Java program. It supports parallelization of loops, a fork/join pattern, a pipeline pattern with multiple pipeline stages, and a divide-and-conquer pattern. Jconqurr is an open source project. The technology is still under development. Figure 6 demonstrates the Jconqurr syntax.

```
@ParallelFor
public static void main(String[] args) {
 int n = 1000;
 float[] arr = new float[n;
 Directives.forLoop();
 for (int i = 0; i < n; i++) {
  arr[i] = ...
 }
}
```

Figure 6: Jconqurr parallel for loop.

The Parallel Java Library [6] is an API-based way to describe parallel execution. It supports shared memory, distributed memory, and hybrid systems. Shared memory programming adopts the parallelization approach of OpenMP and distributed parallel programming is inspired by MPI Message Passing Interface). A strength of this approach is that the parallel Java application is pure Java. A weakness is the volume of code required to describe common parallel execution flows, as illustrated in Figure 7.

```
public static void main(String[] args) {
 final int n=10000;
 new ParallelTeam().execute(new ParallelRegion() {
  public void run() throws Exception {
   execute (0, n, new IntegerForLoop() {
    public void run(int first, int last) {
     for (int i = first; i <= last; i++) {
      ...
     }
    }
   });
  }
 })
}
```

Figure 7: Parallel Java Library parallel for loop

Deterministic Parallel Java (DPJ) [1] is a prototype of a Java programming language extension which adds a parallel

construct that guarantees a deterministic behaviour. This extension forms a type system that can ensure determinism by construction. The language adds fork/join parallelism with `foreach` and `cobegin` keywords. It also adds the notion of memory regions with the use of parametric classes. It analyses the effects of methods to ensure that concurrent read/writes occur in disjoint memory regions. A sample of the special syntaxes is shown in Figure 8.

```
public static void main(String[] args) {
  final int n=10000;
  new ParallelTeam().execute(new ParallelRegion() {
    public void run() throws Exception {
      execute (0, n, new IntegerForLoop() {
        public void run(int first, int last) {
          for (int i = first; i <= last; i++) {
            ...
          }
        }
      });
    }
  })
}
```

Figure 8: Deterministic Parallel Java parallel for loop

Ateji PX is another extension of the Java programming language for parallel computing that targets both shared and distributed systems. It supports task parallelism, data parallelism, message passing, and data flow. The Ategi company that created these language extensions has ceased operations and the technology is not distributed in open source. An example of the Ategi syntax is shown in Figure 9.

```
public static void main(String[] args) {
  int n = 1000;
  float[] arr = new float[n];
  for || (int i = 0; i < n; i++) {
    arr[i] = ...
  }
}
```

Figure 9: Ateji PX parallel for loop

The presented approaches are important developments in the field of Java parallel programming, but they display some significant limitations when the embedded or safety-critical domains are considered.

- As used in this paper, the term *variability* describes the reality that the characteristics of particular processing cores are not constant. Clock rates for individual processors in a heterogeneous system may vary due to manufacturing inconsistencies, transient processor overheating considerations, and electric power conservation. Variability is a major concern in the embedded domain, and this is not currently accounted for in the existing Java frameworks.
- Existing Java frameworks generally assume a shared memory model. This is reasonable in the desktop server, cluster, or supercomputer space for which they are designed, but frequently are unimplementable in embedded systems which rarely demonstrate a global shared memory space.
- The amount of parallelism that is actually deployed can be difficult to analyse in the general fork-join algorithm presented above as it is dynamic and depends on the input data of each run.
- The fork-join framework makes use of work-stealing, which is a very difficult technique to analyse for determining the worst-case response time of a piece of code [22]. This means that such systems are less applicable to code which must be certified.
- The frameworks of Java 7 and 8 are quite large and complex when considered for embedded use. The code base increases memory requirements, but more importantly it also increases testing and certification requirements.

The TouchHMore project aims to address these concerns.

2.4.1 The TouchHMore parallel model

The TouchHMore project is creating a variability-aware framework for the programming of embedded and safety critical systems in Java. Because of the target domain the project uses Safety-Critical Java (SCJ) [19]. The choice of SCJ places a range of important restrictions on the designed parallelisation framework:

- 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 its maximum size. Overflowing any allocation context results in a runtime exception.
- Garbage collection may not be present in an SCJ system, meaning that memory is only reclaimed when an allocation context is exited.
- 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.

Consequently, the parallel computation model in TouchHMore is described as follows:

- An annotation, `@Parallel`, 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 multiple concurrent invocations of the method at runtime.
- These invocations are identical, but their parameters may vary. Normal 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 ToucHMore runtime is queried to determine how many threadlets should be used (and therefore how many chunks array parameters are split into). This number can be specified by the programmer from the SysML system model (see Section 2.1) or left to the runtime to decide.

- 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 is not optimal in the average case, but it allows a much tighter bound on the worst-case response time of an operation.

Any methods annotated with `@Parallel` must also be annotated with `@Offload` (see section 2.3), meaning that threadlets may be executed on other locations of the architecture. Shared memory is neither required nor assumed, but will be used if present to reduce communication overhead.

Method parameters. Due to the fact that any methods annotated with `@Parallel` must also be annotated with `@Offload`, `@Parallel` methods are subject to the restrictions of `@Offload`. This means that parameters must be primitive types, or arrays thereof and annotated as either `@In` parameters or `@Out`.

Most parallel methods operate on large arrays of data. In systems without shared memory, it is usually optimal to split arrays into chunks and pass only the chunks to each threadlet to minimise data movement rather than passing the entire array to each threadlet. To do this with `@Parallel`, the `@In` and `@Out` annotations can take an integer parameter `chunkSize`. This is used for array parameters and describes the smallest amount of each array which is required for one exeuction of the threadlet.

When the threadlet is called, its chunked array parameters will all contain the same number of chunks, but if their `chunkSize`s are different then the lengths will therefore be different. Examine the following threadlet:

```
@Parallel
@Offload{targets = {"Linux-x86"}}
public void threeAverage(@In(chunkSize = 3) int[]
    input, @Out(chunkSize = 1) int[] output) {
 for(int i = 0; i < output.length; i++) {
 output[i] = (input[i*3] + input[i*3 + 1] + input[
     i*3 + 2]) / 3;
 }
}
```

The `chunksize`s of the two parameters are different, but the programmer can assume that for every chunk of `input` there is a corresponding chunk of `output`.

The number of chunks 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. For more information on this see section 2.5.

Because array parameters are chunked, this means the programmer should check the `length` method on the array parameter inside a threadlet rather than assuming the length of the array. This cannot be checked by the compiler. By using `length`, code will operate correctly with and without the `@Parallel` annotation, only its non-functional behaviour will change.

Dealing with the return values of threadlets can be problematic. Due to the fact that the programmer may not know how many threadlets will be spawned, there will be an unknown number of return values to process, which makes it difficult to allocate storage beforehand. To solve this, the programmer can attach a *reduction method* to a parallel method. If the parallel method's return type is X, the reduction method's return type is also X and it takes one argument of type $X[]$ (an array of X). If the parallel method spawns more than one threadlet, the reduction method will be called by the runtime with the argument array containing the return types of each threadlet. Reduction methods are attached to a parallel method by adding the name of the reduction method to the string parameter `reduction` of the `@Parallel` annotation.

An example of the `@Parallel` annotation used to perform a parallel sum of a large array of data is shown in Figure 10. If this is executed without `@Parallel` the code is normal Java and will operate as expected. If executed with `@Parallel`, the runtime will spawn a set of threadlets which will each receive a variability-adjusted size of the input array. Each threadlet will sum its array in parallel, and then the reduction method aggregates the totals. The `chunkSize` is set to 1 because this is the smallest amount of `input` that `sumVector` requires to operate correctly.

Limitations of this model. 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 that achieves variability-aware, low-overhead concurrency in an embedded domain.

Unlike other concurrency frameworks, threadlets cannot self-suspend or directly communicate. Algorithms which rely on barrier synchronizations therefore cannot be deployed in a single parallel loop and must instead use two separate parallel methods and use the implied barrier that occurs when a parallel method completes.

```
@Parallel{reduction = "sumReduction"}
@Offload{targets = {"MIPS_R3000"}}
public int sumVector(@In(chunkSize = 1) int[]
    input) {
  int total = 0;
  for(int i : input) {
    total = total + i;
  }
  return total;
}

private int sumReduction(int[] vals) {
  int total = 0;
  for(int i : vals) {
    total = total + i;
  }
  return total;
}

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

  //Call the parallel method
  int total = sumVector(input);
}
```

Figure 10: An example of the `@Parallel` annotation.

2.4.2 *Implementation strategy*

The `@Parallel` annotation is implemented using the ASM bytecode transformation framework [13] in the same manner as `@Offload` (see section 2.3). Class files are first transformed to process the `@Parallel` annotation, and then to process the `@Offload` annotation. This is shown in figure 11.

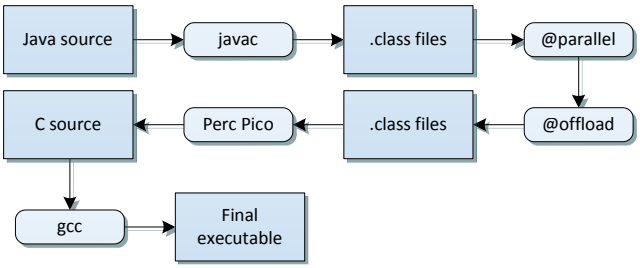

Figure 11: The bytecode transformation flow

The parallel annotation uses an application-wide static thread pool to spawn the threadlets of the parallel method. 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.ManagedThread`, which is part of the SCJ level 2. The size of the thread pool is fixed and set at mission initialisation time. It can serve threadlets to multiple concurrent parallel invocations.

After construction, the thread pool is issued requests through the following method:

```
ThreadPool.forkAndJoin(Runnable[] work)
```

Work is added to the thread pool by passing an array of instances of `Runnable`. This is all transpartent to the programmer however, as the bytecode transformation inserts the thread pool and transforms the bytecode to use it. The transformation process is as follows:

1. The application's `main` method is modified to create a global instance of `ThreadPool` for use by the parallel methods. This is placed in immortal memory.
2. For each `@Parallel` annotated method m:
   (a) Rename m to `_`m`_Threadlet`. Note that its `@Offload` annotation is preserved.
   (b) Create a replacement method called m which:
   - Calls the ToucHMore runtime to determine the number of threadlets to use, n.
   - Splits the input parameters of the parallel method into n sub-arrays.
   - Creates an array of n Runnables to call `_`m`_Threadlet`, passing the sub-arrays to each.
   - Passes the Runnables to the thread pool.
   - Collects the outputs of the Runnables into their output arrays.

After this transformation the threadlet method is still annotated with `@Offload`, which is then processed according to the processing described in section 2.3.

2.5 Details of Run-Time Support

The runtime developed in ToucHMore is responsible for monitoring the platform and allocating the workload efficiently.

Due to the advancement in CMOS technology, the sub-65 nm cores are increasingly affected by the phenomenon called 'intra-die'[1] process variation, which impacts the overall performance of the architecture. So it has become necessary to compensate or tolerate this variability effect, which can be done on different levels, from circuit level (i.e. gate level) going the way up to system level (i.e. architecture level). And a variability-tolerant platform is assumed not only to monitor variations in nominal operating characteristics (such as clock frequency or power consumption), but also to compensate them in such a way that achieves performance maximization or power consumption reduction.

As the variability factor can lead to suboptimal or inefficient task allocation at runtime, the need to develop variability-tolerant applications has risen. However, writing such applications has proven challenging and has required excessive time and effort from programmers. And here comes the need for an innovative approach for variability compensation from a high-level model of the target hardware platform. Starting from a high-level model of the target platform, the abstraction or description of the platform should necessarily be detailed enough to capture information relevant for the runtime manager[2] to make task allocations. Variability-relevant information is annotated in the model as properties, e.g. clock frequency and power consumption. These quantities

[1] Also referred to as 'within-die' or 'core-to-core'.
[2] The terms 'runtime manager', 'runtime library' and 'runtime engine' bear the same meaning and are used interchangeably through the text.

are obtained from the post-manufacturing characterization, thus taking process variability into account.

This customization information is passed to the runtime library to be accessed by a state-of-the-art probabilistic policy to make variability-aware allocation decisions at runtime. Hence it becomes possible to mitigate the impact of uncertainties due to the intra-die process variation [21]. The aim is to have variability-aware task allocations toward achieving performance gains and power optimization.

In addition to the implemented annotations described in this paper, the project has also defined an annotation named `@Energy` which is under active development. This annotation will be used by the programmer to specify to what extent an operation's performance can be impacted by optimizations aimed to reduce energy use (or temperature). The intent of the annotation is to provide a model-driven, code-independent way to inform the runtime how aggressively it should apply energy/power/temperature-aware policies to different parts of the designed system. The annotation supports `deadline` and `qos` (Quality of Service) parameters that are used as inputs to the defined optimizations. The project is currently investigating the use of frequency/voltage scaling (DVFS) and workload to core allocation, but many other potential policies are being considered. 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 [23] and the use of such an annotation can assist the programmer in the investigation of this trade off.

3. EXPERIMENTAL RESULTS

As most of the components of the ToucHMore tool chain are still under development, experimental results to date are very limited.

The runtime algorithms used for workload distribution between coprocessors have been evaluated with experimental simulation of a matrix multiplication kernel as a benchmark - using three different configuration sets: i) Homogeneous, where all cores are equal; ii) Quasi-homogeneous where cores are grouped into two frequency classes; and iii) Fully heterogeneous, where all cores are different. These cases are representative of variability scenarios. In the first, we assume the platform is homogeneously degraded. In the second, the degradation is localized while in the third, the degradation is randomly distributed across the cores. Note that frequency values are normalized with respect to the maximum frequency. Results reported in Table 2 for probabilistic rank frequency and in Table 3 for rank power.

Results show that the percentages of workload allocated to each processor depend on the speed or power differences between the processors. In the case of probabilistic frequency policy, to each core a probability of allocation is associated which is proportional to the speed difference among the cores, to achieve overall execution time equalization [12]. We report in the third column the weighted frequency values that account for the distance from the controller or master core. We opted for not weighting the power values, however,

Homogeneous			
	Norm. Freq.	Weighted Values	Allocation Percentage
Slave Core 1	0.95	1	34.34%
Slave Core 2	0.95	1	34.08%
Slave Core 3	0.95	0.9	31.58%
Quasi-Homogeneous			
	Norm. Freq.	Weighted Values	Allocation Percentage
Slave Core 1	1.00	1.05	35.66%
Slave Core 2	0.95	1.00	33.69%
Slave Core 3	0.95	0.90	30.65%
Fully Heterogeneous			
	Norm. Freq.	Weighted Values	Allocation Percentage
Slave Core 1	0.95	1.00	34.00%
Slave Core 2	0.90	0.95	33.03%
Slave Core 3	1.00	0.95	32.97%

Table 2: Results about allocation of matrix multiplication threads on the MIPS cores in GENEPY platform simulator using the Probabilistic Rank Frequency policy.

Homogeneous		
	Power Value (mW)	Allocation Percentage
Slave Core 1	15	33.79%
Slave Core 2	15	33.17%
Slave Core 3	15	32.86%
Quasi-Homogeneous		
	Power Value (mW)	Allocation Percentage
Slave Core 1	13	33.79%
Slave Core 2	15	32.37%
Slave Core 3	13	33.84%
Fully heterogeneous		
	Power Value (mW)	Allocation Percentage
Slave Core 1	15	32.05%
Slave Core 2	13	34.77%
Slave Core 3	14	33.18%

Table 3: Results about allocation of matrix multiplication threads on the MIPS cores in GENEPY platform simulator using the Probabilistic Rank Power policy.

due to the relatively negligible power consumption of the network on chip routers.

4. CONCLUSION

The unique needs of low-power heterogeneous computer systems require non-traditional Java approaches in order to optimize performance and power consumption. The SCJ Java subset, in combination with specially designed annotations and static code transformation capabilities based on the use of the ASM tool chain, is sufficiently general to serve as the basis for deploying Java on such systems.

Further work is required to evaluate the benefits of Java in such configurations. Ongoing research focuses on evaluating the performance and power management characteristics of Java applications running on low-power heterogeneous computer systems in comparison with similar applications written in C and C++. An additional aspect of the ongoing research is to evaluate the maintainability and code reuse benefits of the ToucHMore restrictive style of Java application code in comparison with the maintainability and code reuse benefits of C and C++ code on similar deployment platforms.

5. REFERENCES

[1] Deterministic parallel java. http://dpj.cs.uiuc.edu/DPJ/.

[2] Jamp. http://https://www2.informatik.uni-erlangen.de/EN/research/JavaOpenMP/index.html.

[3] Jconqurr. https://code.google.com/p/jconqurr/.

[4] JSR 302: Safety Critical Java^{TM}Technology. http://jcp.org/en/jsr/detail?id=302.

[5] Openmp home page. http://www.openmp.org.

[6] Parallel java library. http://www.cs.rit.edu/ ark/pj.shtml.

[7] The real-time specification for java^{TM}. http://www.rtsj.org/.

[8] Openhmpp: directive-based approach to program accelerators. http://www.openhmpp.org/, 2007.

[9] ToucHMore deliverable D1.1: Report on the specification of the heterogeneous reconfigurable multicore platform. http://www.touchmore-project.eu/index.php?option=com_content&id=5, 2012.

[10] ToucHMore deliverable D1.2: report on the UML/SysML model of the target platform. http://www.touchmore-project.eu/index.php?option=com_content&id=5, 2012.

[11] AMD: Heterogeneous multi-core processor solutions to trigger new wave of embedded product innovations, digitimes supply chain window. http://www.digitimes.com/supply_chain_window/story.asp?datepublish=2012/3/15&pages=PD&seq=207, 2013.

[12] G. Ayad, A. Acquaviva, E. Macii, B. Sahbi, and R. Lemaire. Hw-sw integration for energy-efficient/variability-aware computing. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 607–611, 2013.

[13] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[14] J. M. Bull and M. E. Kambites. JOMPâĂŤan OpenMP-like Interface for Java. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 44–53. ACM, 2000.

[15] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236. ACM, 2010.

[16] S. Cox. The case for heterogeneous multi-core socs. *Chip Design Magazine*.

[17] N. Giacaman, O. Sinnen, et al. Pyjama: OpenMP-like implementation for Java, with GUI extensions. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 43–52. ACM, 2013.

[18] B. Goetz. JSR335: Lambda expressions for the Java programming language. http://jcp.org/en/jsr/detail?id=335.

[19] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*. Citeseer, 2009.

[20] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM, 2000.

[21] C. Nicopoulos, S. Srinivasan, A. Yanamandra, D. Park, V. Narayanan, C. R. Das, and M. J. Irwin. On the effects of process variation in network-on-chip architectures. *IEEE Transactions on Dependable and Secure Computing*, 7(3):240–254, 2010.

[22] L. M. Nogueira, L. M. Pinho, J. Fonseca, and C. Maia. On the use of work-stealing strategies in real-time systems. In *Workshop on High-performance and Real-time Embedded Systems (HiRES 2013), held in conjunction with the 8th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2013), Berlin, Germany*, 2013.

[23] F. Paterna, A. Acquaviva, A. Caprara, F. Papariello, G. Desoli, and L. Benini. Variability-aware task allocation for energy-efficient quality of service provisioning in embedded streaming multimedia applications. In *IEEE Transactions on Computers, vol. 61, no. 7*, pages 939–953. IEEE, 2012.

[24] J. S. R. McIlroy. Hera-JVM A Runtime System for Heterogeneous Multi-Core Architectures. In *Proceedings of the ACM Conference on Object-Oriented Programming, Sys- tems, Languages, and Architectures*. ACM, 2010.