ARGO

WCET-Aware Parallelization of Model-Based Applications
for Heterogeneous Parallel Systems

H2020-ICT-2015

Project Number: 688131

Deliverable D2.2

D2.2 A WCET-aware Multi-core Programming Model

Editors:	Timon ter Braak
Authors:	Timon ter Braak, Leonard Masing, Simon Reder
Version:	1.01
Status:	FINAL
Dissemination level:	Public (PU)
Filename:	D2.2_wcet_aware_multi_core_programming_model_1v01.docx

ARGO Consortium

Karlsruhe Institute of Technology	DE
Scilab Enterprises	FR
Recore Systems B.V.	NL
Université de Rennes I	FR
Technological Educational Institute of Western Greece	GR
AbsInt Angewandte Informatik GmbH	DE
Deutsches Zentrum für Luft- und Raumfahrt	DE
Fraunhofer IIS	DE

Document revision history

Version	Based on	Date	Author	Comments / Changes
1.00		10/10/2016	Timon ter Braak	Final version
1.01	1.00	28/11/2016	Timon ter Braak	Changed dissemination level to public.

About this document

This document describes the ARGO Programming model and the APIs used to program the two ARGO platforms. The programming model defines tasks and communication and synchronization primitives tasks can use in order to interact. The Programming Model is WCET-aware by ensuring that hardware resources used for communication and memory accesses are known at compile time. This enables a tight estimation of the WCET on code- and system-level.

Table of Contents

List of Figures

1. Introduction

The main motivation for the definition of a new programming model is to enable concurrent and parallel programming on the many-core platforms targeted by ARGO, without having to deal with the difficulties involved at low abstraction levels. Because of parallelism in the execution, several models become prominent: the behaviour of the (distributed) memory determines how communication is realized, as defined by the memory model; the concurrency model defines composition and interaction of concurrent computations; and the model of computation defines the fundamentals of the algorithm. Taking full control over all details of these models is almost impossible for the programmer to do by hand. A programming model presents all layers of the underlying platform in a convenient way. Different programming languages make different trade-offs for this model regarding the level of abstraction, ease of programming, and control over the hardware.

A *programming model* is an abstraction of the system that consists of an abstract machine, a programming paradigm, and a subset of features of the underlying models that has to be dealt with by the programmer [2].

A (good) programming model is tailored towards convenient usage by the programmer. That is, the abstract machine should be closely related to the model of computation. The model of computation can be reflected in the paradigm, which is a way of organizing the application. Features of the platform that are not exposed by the programming model should be handled automatically by the compiler or other tooling in a correct way.

Regarding the ARGO project, one of the features that needs to be dealt with by the user is performance related (especially the worst case performance). Applications need to execute and/or complete their functionality in a timely fashion. The expected performance, as well as directives on how to achieve it need to be specified by the programmer.

1.1 The Programming Model in the ARGO Context

In the ARGO project, the programming model will be primarily used by the ARGO toolchain instead of the programmer himself. Since ARGO targets real-time systems, the programming model is designed to be efficiently implementable on well-predictable multi-core platforms.

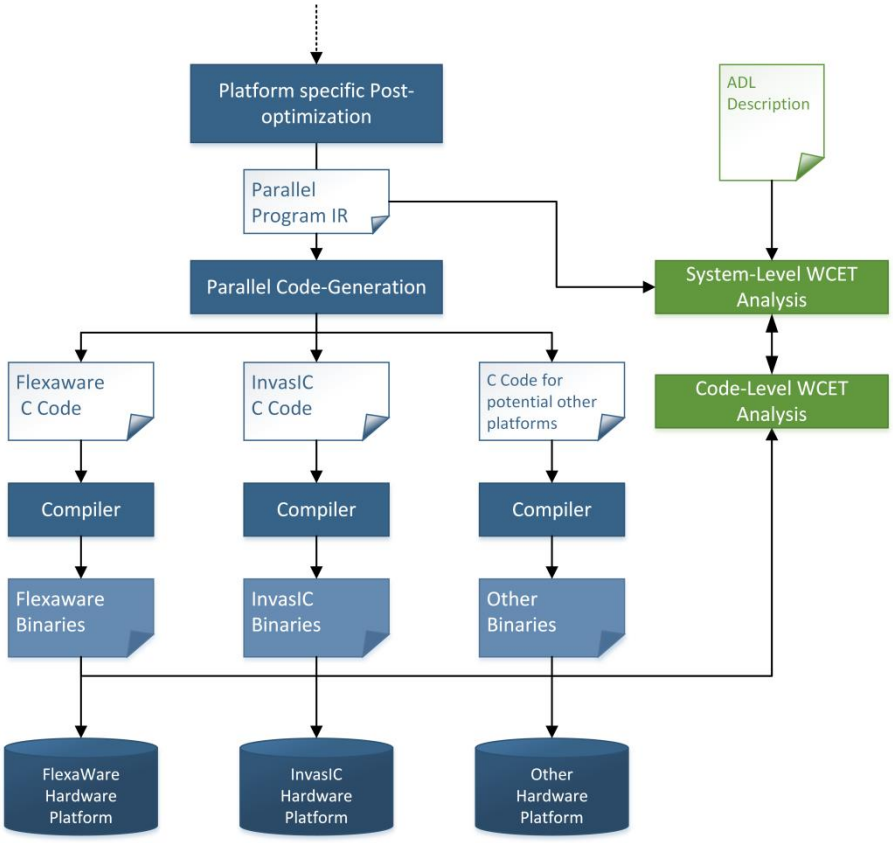

Figure 1: The final stages of the ARGO Tool-Flow

Figure 1 shows the final stages of the ARGO tool-flow starting with the platform specific post optimization of the application which previously has been parallelized. The subsequent Parallel Program IR (PPIR) is an intermediate representation which makes use of the concurrency, communication and memory models defined in the ARGO Programming Model. However, the PPIR it still independent of the actual target API in order to be compatible to all target platforms. Based this IR, the Parallel Code Generation step is responsible for translating the application into C code using the target API. The APIs of the different platforms do not have to be identical, but each of them must implement the features specified in the programming model.

Both the C code and the PPIR underlie several requirements that result from the hardware platforms as well as the WCET analysis tools. The design of the WCET-aware ARGO Programming Model must comply with these requirements in order to ensure compatibility of the parallel program representations with the hardware platform and the WCET analysis tools.

In order to provide good efficiency while still supporting a variety of platforms, the programming model includes several optional features that must be not supported by all target platforms. In order to complement the programming model with such hardware specific

information, ARGO uses an ADL which has been extended for timing predictable multi-core systems.

1.2 Reservation based resource partitioning

Reservation-based resource partitioning for real-time systems, resource reservation (RR) for short, is an emerging paradigm for resource management in real-time systems [4]. Classical real-time schedulability analysis is based on the knowledge of the worst-case execution time (WCET) of all periodic tasks. In modern processors it is very difficult to tightly estimate the WCETs. Figure 2 shows a distribution of possible execution times; using timing analysis it is challenging to derive a tight upper bound on the actual WCET. If a task instance executes longer than assumed by the schedulability analysis, this is a temporal fault. Temporal faults may occur because the WCET has not been computed correctly, because of a deliberate decision not to allocate for the worst case, or because of a temporary or permanent fault in the application. The resource reservation mechanisms provide temporal protection: they prevent temporal faults from affecting the temporal behaviour of other tasks in the system, by enforcing execution time bounds at run time. Temporal protection is the main reason for adopting some form of RR as a de-facto standard for the aerospace industry [4].

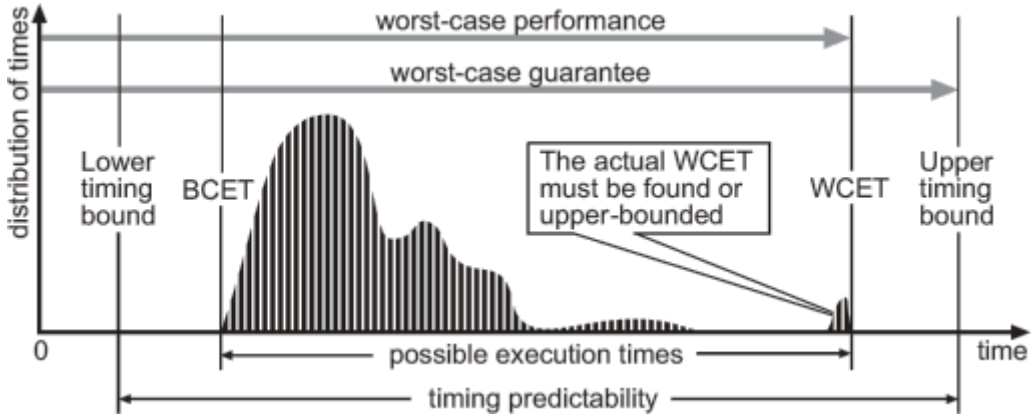

Figure 2: Worst-case execution time estimation [1].

Reservation of CPU time is a fundamental real-time abstraction that guarantees that a thread will be scheduled for a specific amount of time during each period. Reservations are a good match for threads in real-time applications whose value does not degrade gracefully if they receive less processing time than they require. A wide variety of scheduling algorithms can be used to implement CPU reservations, and many different kinds of reservations are possible. The following list describes the properties of these kinds of CPU reservations. In particular, note that every CPU reservation is either basic or continuous, and either hard or soft, and that these properties are orthogonal [5].

1.2.1 Basic CPU reservation

Basic CPU reservations are what would be provided by an earliest deadline first (EDF) or rate monotonic scheduler that limits the execution time of each thread that it schedules using a budget. For a reservation with amount x and period y, a basic reservation makes a guarantee to a virtual processor (VP) that there exists a time t such that for every integer i the VP will receive x units of CPU time during the time interval [t + iy, t + (i + 1)y]. In other words, the reservation scheduler divides time into period-sized intervals, during each of

which it guarantees the VP to receive the reserved amount of CPU time. The value of t is chosen by the scheduler and is not made available to the application.

1.2.2 Continuous CPU reservation

Continuous CPU reservations are those that make the following guarantee: given a reservation with amount x and period y, for any time t, the thread will be scheduled for x time units during the time interval [t, t + y]. A continuous CPU reservation is a stronger guarantee than a basic CPU reservation since every period-sized time interval will contain the reserved amount of CPU time, rather than only certain scheduler-chosen intervals. In other words, a continuous reservation scheduler is not free to arbitrarily rearrange CPU time within a period. Continuous reservations are provided by schedulers that utilize a (possibly dynamically computed) static schedule, where a thread with a reservation receives its CPU time at the same offset during every period. In contrast, a basic reservation scheduler retains the freedom to schedule a task during any time interval (or combination of shorter intervals) x units long within each time interval y units long. Figure 3 depicts two CPU reservations, one basic and one continuous, that are guaranteed to receive 3 ms of CPU time out of every 8 ms. The continuous CPU reservation is also a basic reservation, but the basic reservation is not a continuous reservation.

As an example, the real-time scheduler of Linux uses a default time slice of 100 ms. Assuming a scheduler interval of one second, 10 time slices can be handed out. Then one may specify that the task required 3 time slices every 10 time slices. In a continuous CPU reservation scheme, the task is then guaranteed to have access to the CPU for

$$\frac{3+3}{(10-7)+10} = \frac{6}{13} \sim 46\%$$

of the time. This is calculated by taking two consecutive scheduler intervals, where we assume the worst case latency before the task is allowed access to the CPU in the first interval. This gives a higher estimation compared to the basic CPU reservation scheme where we consider a worst-case scheduler latency in each interval; that would have been 3 time slices out of 10, giving an average processing time of 30%.

1.2.3 Hard CPU reservation

Hard reservations limit the CPU usage of a virtual processor to at most the reserved amount of time, as well as guaranteeing that it will be able to run at least that rate and granularity. Hard reservations are useful for applications that cannot opportunistically take advantage of extra CPU time; for example, those that display video frames at a particular rate. They are also useful for limiting the CPU usage of applications that were written to use the full CPU bandwidth provided by a processor slower than the one on which they are currently running. For example, older CPU-bound games and voice recognition software have been run successfully on fast machines by limiting their utilization to a fraction of the full processor bandwidth.

1.2.4 Soft CPU reservation

Soft reservations may receive extra CPU time on a best-effort basis. They are useful for applications that can use extra CPU time to provide added value. However, no extra time is guaranteed.

1.2.5 Probabilistic CPU reservation

Probabilistic reservations, are a special case of soft CPU reservations that guarantee a thread to receive a specified minimum execution rate and granularity as well as having a chance of obtaining extra CPU time. Probabilistic reservations are designed to give applications with highly variable execution times a high probability of meeting their deadlines without actually giving them a guarantee based on their worst-case execution times.

1.2.6 Non-pre-emptive CPU reservation

Non-pre-emptive reservations guarantee that the amount of reserved time will be provided to the virtual processor in a single block, at the same offset in each period. In other words, for a non-pre-emptive reservation with amount x and period y there exists a time t such that for every integer i, the thread will be scheduled for the time interval [t + iy, t + iy + x]. This guarantee is useful for threads in applications that must interact with sensitive hardware devices requiring the undivided attention of the CPU. Although any reservation provided by the main thread scheduler in an OS is always pre-emptible by hardware and software interrupts, these events are generally very short compared to time quanta for application threads.

1.2.7 Synchronized CPU reservation

Synchronized reservations are a special case of non-pre-emptive reservations that start at a specific time. Synchronized reservations permit the requesting thread to specify a time t such that for every integer i, the thread will be scheduled for the time interval [t + iy, t + iy + x]. Synchronized reservations are useful to synchronize a reservation to a periodic external event (such as a video card refresh--this is a real, and difficult problem), or to implement schedulers that coordinate thread execution on different processors of a multiprocessor, or on different machines.

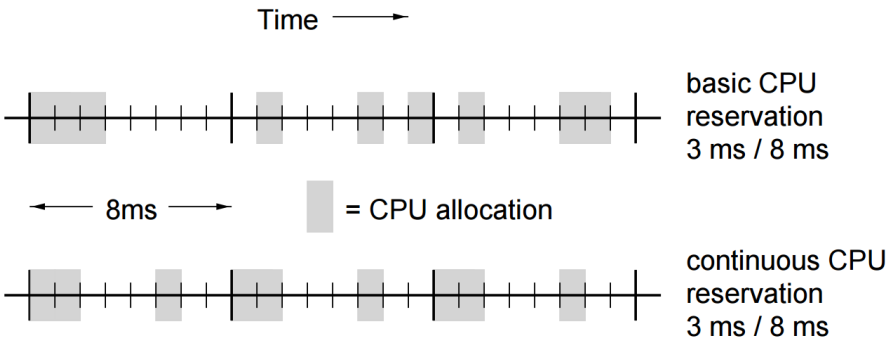

Figure 3: Partial time-line showing a basis CPU versus continuous CPU reservation [5].

The WCET awareness of the programming model is provided through a deterministic means of sharing resources. Interactions with shared resources have to be made explicit, in order to

ensure that the platform is able to regulate interference between concurrently executing entities and to arbitrate between concurrent accesses to resources. The assignment of the individual components of a parallelized application to the resources provided by the platform is performed at design-time by the ARGO toolchain. The programming model and the corresponding API thus (only) provide explicit means to make use of platform resources. Closing the cycle, this again enables worst-case execution time analysis that matches the behaviour of the application on the target platform.

1.3 Outline

The remainder of this document is structured as follows. In the next section, a programming model is presented that is aware of execution times in the context of the resource reservation paradigm. Thereafter, an abstract example API for the programming model is provided on a pseudo-code level. The actual API implementations for the bare-metal target architecture InvasIC is presented, followed by FlexaWare, which is a target architecture including a run-time layer.

2. A WCET-aware programming model

In this section, we first analyse the requirements to the programming model given the ARGO hardware platforms and the need to carry out a multi-core WCET analysis of a parallel program. Based on that, we define the programming model including the abstract machine, its model of computation as well as the programming paradigm, which consists of a memory model and a concurrency model.

2.1 Requirements

One essential requirement of the programming model is to be implementable with a good performance on the target hardware for both average case and worst case execution times. In general, this requires the efficient use of the available hardware resources. For the worst-case performance it is additionally required to be able to statically determine the worst-case timing behaviour of an application at compile time. For the system-level WCET analysis tools, this is only possible when the hardware resources used by a specific program section are well known before running the program. In particular, this results in the following requirements:

- The mapping of the tasks to processor-cores must be known at compile time in order to statically determine the utilized hardware resources
- The hardware must be initialized such that there is a statically predictable upper bound for the delay of each communication between two tasks
    - The programming model should enforce pre-allocation of communication resources at initialization time in order to implement this efficiently
- In order to analyse worst case memory access timings, the address ranges for each memory access in the program must be inferable at compile time. An address range in this context refers to a continuous range of addresses in the physical address space of the respective core, which refer to the same hardware memory component.

2.2 The abstract machine

We define a platform as the combination of the hardware and software that together shapes the environment for an application. A platform is made up of components. A component may be part of a composite, whereas a composite may contain one or more (processing) elements (sub-components). The components may vary in architecture, making the platform heterogeneous. The memory components in the platform may be accessed through a distributed memory space. Figure 4 illustrates the scope of the programming model with respect to the underlying hardware architecture, where CPU may be replaced with any type of hardware element that contributes to the functionality of an application, and the runtime may or may not be present. The number of tasks that can be assigned to execute on a specific CPU is platform dependent; some platforms may restrict the number of tasks per CPU to one.

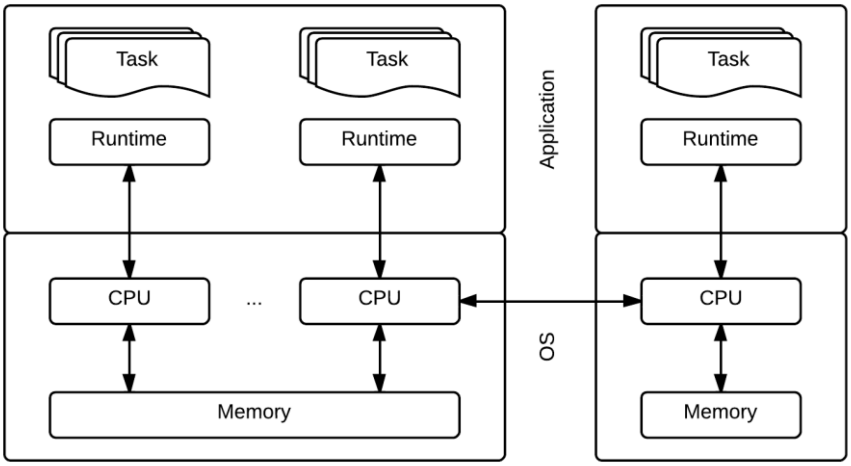

Figure 4: Proposed abstract machine.

This model assumes heterogeneous distributed (shared) memory systems. We consider homogeneous systems and shared memory systems to be special cases of the systems we focus on, and by that, we support them as well. However, alternative models that do not support heterogeneous distributed memory systems may be able to achieve a higher efficiency for these (less constrained) systems.

An application runs on top of an operating system layer that takes care of the (static) resource allocation and management[1]. The run-time, which is not necessarily an active component, provides the application with the concepts defined by the programming model[2].

Tasks execute code that is compatible with the underlying architecture of the processing elements. A task may be compatible with one or more architecturally different processing elements. Communication channels are provided to allow tasks to exchange data and to synchronize in an explicit manner. Any control and status-oriented information required for its implementation shall not be visible to the application programmer.

2.2.1 The model of computation

A model of computation is an abstraction that defines the elementary operations, i.e. transformations, for computation. Such a model allows expressing algorithms, which are sets of transformations, and data, which is used to apply algorithms on. The target language selected for ARGO is the C99 language, as defined by the ISO/IEC 9899:1999 standard [6]. The C Standard formalizes a separation between the language and the library by distinguishing between hosted and freestanding implementations. Informally, a hosted implementation is a C translation and execution environment running under an operating system with full support for the language and library. A freestanding implementation is a C translation and execution environment with nearly full language support but essentially no support for the standard library's runtime components – an environment not uncommon among low-end embedded systems.

[1] A (bare-metal) board support package, even linked-in with the application is also considered to be an operating system. Without it, the application is not functioning correctly.
[2] Note that even the C language incorporates a run-time, unless it is explicitly configured not to do so.

The two forms of conforming implementation are hosted and freestanding. A conforming hosted implementation shall accept any strictly conforming program. A conforming freestanding implementation shall accept any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`,`<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, and `<stdint.h>`.

Those seven named headers define only constants, types, and a few function-like macros. They do not declare any functions. A freestanding implementation need not provide headers such as `<stdio.h>` or `<stdlib.h>`. Thus, it need not support input and output functions such as `fgetc` and `printf` or memory management functions such as `malloc` and `free`. A freestanding implementation does not even need to support the string handling functions in `<string.h>`, such as `memcpy` and `strlen`. Thus, the C standard's notion of a freestanding implementation defines a narrower subset than what most embedded toolchains offer and what most embedded programmers actually use.

The C99 language exhibits an imperative or structured paradigm, and the machine belongs to the class of register machines. Because the language is single-threaded, memory consistency is not relevant. Therefore, the C99 programming model only defines a fairly simple sequential machine. This implies that the compiler can handle all machine specific memory issues, such as alignment, and memory-related optimizations. In the proposed programming model, C99 may be used to implement tasks using sequential programming code. The same C99 language may be used as a coordination language to create a coherent collection of tasks that together compose a multi-threaded application. More on this will be explained in the next section; the programming paradigm.

2.3 The programming paradigm

A programming paradigm is the fundamental style of computer programming, a way of building the structure and elements of computer programs. As the platform model is defined as a set of processing elements that are only loosely coupled, a (software-defined) task is implemented with a piece of sequential programming code. Synchronization with other tasks is achieved implicitly through channel communication, and explicitly through explicit synchronization statements (for example, a statement that waits for the completion of a task).

Tasks are considered to be objects instead of functions. Objects allow for associating functions with data and metadata. This makes heterogeneous computing more easy to incorporate into the model. One purpose of metadata is to inform the run-time of specific properties derived by the compiler or specified by the user that allow for more efficient deployment of the application compared to the case where the run-time cannot make any assumptions on the expected behaviour. In a simplified scenario, a task can be made up of a reference to the function to be executed together with an address to the stack memory.

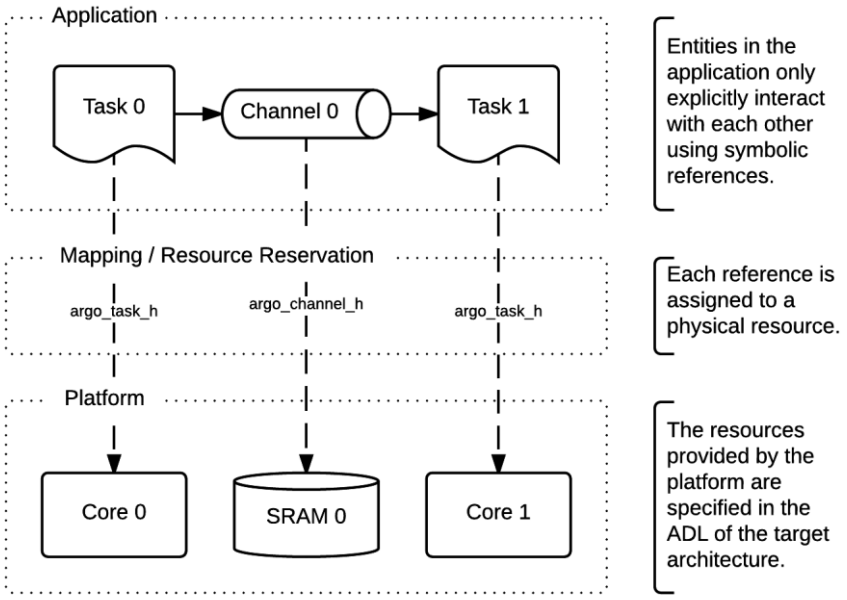

Figure 5: Mapping of application components to hardware components

In ARGO, we assume a one to one mapping of the tasks t_j to processor cores p_i, while the mapping must be known at compile time according to the requirements in Section 2.1. Besides tasks, the programming paradigm specifies two other classes of application components, which are *logical memories* and *channels*. Just like tasks, those components must also be statically mapped to one or more corresponding hardware resources. The mapping of Application components specified by the programming model to hardware resources is show for a simple example in Figure 5.

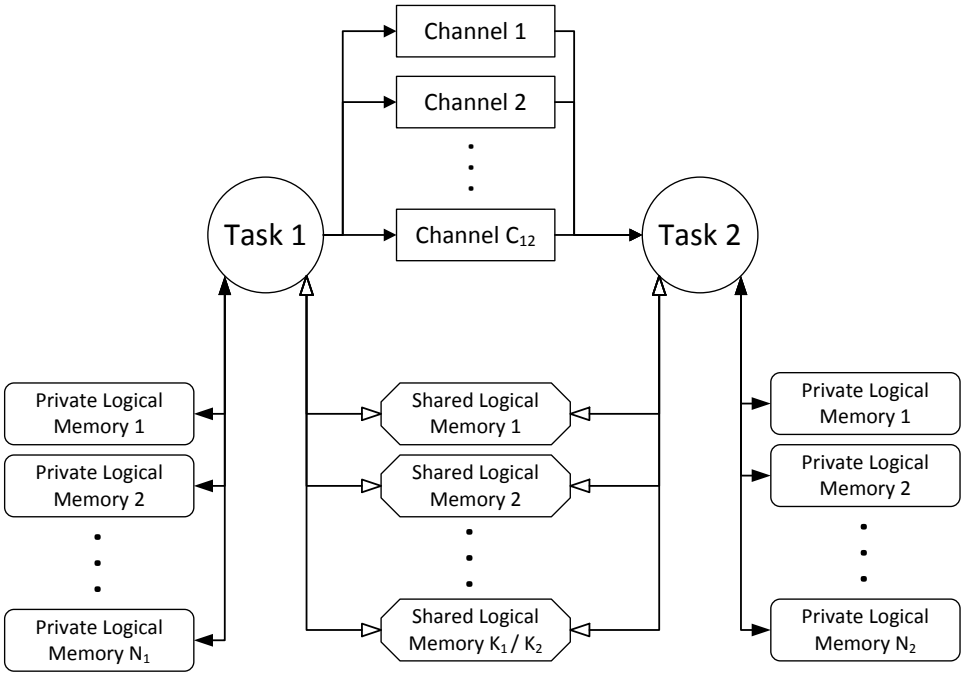

Figure 6: Communication channels and logical memories for two tasks

Logical memories on the application level are linked to a specific address range (see definition in Section 2.1) in the address space of the processor core which runs the respective task. Logical memories are classified into private and shared logical memories. The private logical memories are only accessed by a single task, while the shared ones can be accessed by different tasks. Given this, each task t_j can access a set of N_j private and a second set of K_j shared memory segments. Note that shared logical memories are not guaranteed to be mapped to the same address on each task/processor core. Figure 6 shows the relations of application components for a two-task example.

The third component type in the ARGO programming model is the *channel* which represents the primary communication primitive in ARGO. A channel represents a unidirectional FIFO with fixed capacity which is known at compile time. Channels must be explicitly instantiated by an application. They must have exactly one sending task t_k and exactly one receiving task t_j, such that a well-known set of C_{jk} connecting channels exists between the two tasks.

2.3.1 The communication model

The communication model defines the way tasks may communicate with each other regarding data flow and synchronisation. Communication in the ARGO programming model is designed around the concept of unidirectional channels. A channel connects two communication partners over some communication resource by providing a fixed-size FIFO buffer which can be read or written to respectively. The buffer may be present in different forms: e.g. in a shared memory or as separate buffers in an interconnect (e.g. NoC). For the hardware components that are used to implement the channels, a model for the calculation of interference and a worst-case time needs to be available. A task is blocked when it attempts to perform a communication operation that cannot proceed; that is, reading from an empty channel or writing to a full channel. However, the blocking of channels may only occur due to the involved tasks and never due to any unrelated channel or task. A task may have n input channels and m output channels, limited only by the available hardware resources.

Channels can either be configured to transmit raw data or to accept data of a given data type. The latter is especially relevant for heterogeneous systems, since data type conversions then are implicitly done by the channel (e.g. if the involved processor cores have a different native endianness). The support for implicit data type conversions however is optional and depends on the target platform.

2.3.2 The memory model

The memory model can be seen as a contract between hardware and software. Software relies on the semantics of the memory as defined by the model, and the hardware implements that model. Any cache coherency protocol should follow or implement the requirements that are imposed by the intended memory model of the architecture. A list of (memory) consistency models is found at [1].

A *memory model* is an abstraction that defines the constraints on the order in which memory operations must appear to be performed, i.e. become visible to the processors, with respect to one another. It defines the semantics of shared variables.

At the application level, it is convenient to have a strong memory model. When the memory model that is implemented by the hardware is weaker than required, software can complement the hardware's memory model. This may come with high costs when it is not a deliberate decision. We address the needs of the model on the memory architecture of the

platform, because many hardware components follow a relatively weak memory consistency. model.

As introduced previously (see Figure 6), memory is represented in the programming model using several logical memories which can be either private to a task or shared by multiple tasks. The links between tasks and the logical memories they can access are assumed to be known at compile time. Each logical memory has a size which is defined for the whole system, but the start address of the memory depends on the accessing processor core. Memory sizes, addresses as well as non-functional properties (such as performance information or caching behaviour) of the logical memories can be derived from the ARGO ADL using the mapping to the underlying hardware resources. Logical memories are not mapped to exactly one hardware memory. However one memory component of the hardware might contain several private or shared logical memories.

For the WCET analysis, interference between memory accesses from different concurrent tasks is of particular interest. Shared logical memories are obviously always prone to interference. However, interference might also occur for private logical memories, since these might be mapped to physical memories which host other logical memories. An additional source of interference on private logical memories could be e.g. a memory bus which is shared by different processor cores.

The (call) stack is (mostly) managed by the implementation language of the task and the details are hidden from the programmer. The programming model requires that the (maximum) size of the stack is known at the creation time of a task in order to allocate or reserve the worst-case required stack size. Logical memories that are not occupied by the stack or instructions can be used freely by the application. To do so, the application can retrieve the start address and size of a memory segments using respective API calls. Optionally, a predictable dynamic memory allocation API might be supported on some target platforms in order to allocate and deallocate parts of a private logical memory (dynamic allocation of shared memory is not supported).

To access shared memory, the respective logical memory segment is either directly mapped it into the processor's address spaces and/or the application has to use explicit calls to read/write functions. It depends on the available hardware support if the former method is available. For the later method, the programming model distinguishes between synchronous and asynchronous memory read calls. The availability of asynchronous reads again depends on the hardware.

The main purpose of the shared memory segments is to share data efficiently and to reduce the memory capacity that would otherwise be required for duplicate copies of the same data. It is, however, not guaranteed that any locking primitives or exclusive access means are made available. As such, the shared memory is for sharing data, not for synchronization purposes. In an idiomatic way, shared memory may be altered first, after which the update is communicated over communication channels. Then, it is guaranteed that the receiver only receives the 'message' after the memory is updated. Note that additional synchronization is required to avoid any overwriting of data in subsequent iterations of the producer task, due to buffering in the communication channel.

Due to the focus on WCET-awareness, the use of memory for caching purposes should be limited as much as possible, unless its behaviour is predictable.

2.3.3 Application structure

The generic structure of an application consists of a number of inputs and a number of outputs, with processing logic in between. Figure 7 shows such a structure. Typically, the

input and output functionality is platform specific, and as such, should not be considered in the code generation, parallelization or optimization steps. However, the processing logic should be compatible with the externals, and any predefined logic required for the input and output functions should be included in the application, but should only be added during the final compilation stages.

A valid use-case to be considered is that any predefined functionality (such as I/O) has *formal* parameters requiring the application to supply suitable *actual* parameters. For example, if an input function is able to read content from a file, the formal parameter required may be ‘filename’, whereas the actual parameter that needs to be provided in order to use the function could be ‘/dev/stdin’.

It is envisioned that a target architecture may provide a board support package or a library with predefined functions available to the user. To inform the user of the existence of these functions, they have be encapsulated by entities that fit together with the model based design approach. This encapsulation entity may then even be configured by the user to integrate the predefined functionality into the application. The various steps in the toolchain then need to be aware that the functions are ‘built-in’.

In this concept, applications are considered to be constructed hierarchically. In ARGO, we restrict the hierarchy to two layers. The first layers is considered to be the *main* task, or the initialization procedure, depending on the specific target platform. The first layer instantiates the second layer, which is composed of all the tasks and channels in the application. Doing so, functionality can be instantiated together with the correct parameters and required data. This is especially relevant when a task is implemented in hardware; the hardware needs to be configured such that is matches the required functionality. In Figure 7, the outermost shell is considered to be the first layer that instantiates the input and output tasks and the processing logic in between.

Note that the complexity of the first layer depends on the specific requirements of the target architecture. In a minimal use-case, the system needs to be informed on what task needs to be executed on which core, and each core has to be configured to start the execution of its assigned function.

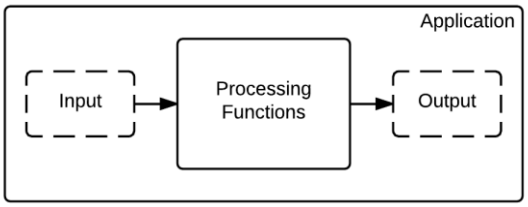

Figure 7: Generic structure of an application.

3. ARGO target API

Based on the specification of the ARGO programming model, an implementation defines an API which consists of several functions that initialize the application and perform communication operations. In the following an exemplary API is presented as pseudo code in order to illustrate the functionality supported by the Programming Model. Both ARGO platforms implement this functionality (at least the required parts) even if the APIs itself are not identical. More details on platform specific APIs will be provided in the subsequent sections 4 and 5 which are dedicated to the individual platforms.

3.1 Initialization functions

The following functions must only be called during the program initialization phase.

```
/*
 * Task creation. Returns a task handle.
 * Parameters are platform specific.
 */
function <task_handle> argo_task_create( ... );

/*
 * Create a new channel featuring FIFO communication.
 * The FIFO has a fixed size which is either platform dependent
 * or can be specified as function parameter.
 * Parameters are platform specific.
 */
 function <channel_handle> argo_channel_create(...);

/*
 * Functions to get the start address and size of the
 * logical memory segments.
 * The functions are free of side effects and the
 * result can also be determined at compile time.
 *
 * - The mem_start functions return a NULL pointer if the
 *   memory segment is not address mapped.
 * - The size functions return 0 if the segment level
 *   is not available on the current processor core.
 */
function <pointer> argo_priv_mem_start( <logical_memory_id> );
function <size>    argo_priv_mem_size( <logical_memory_id> );

function <pointer> argo_shared_mem_start( <logical_memory_id> );
function <size>    argo_shared_mem_size( <logical_memory_id> );

/*
 * Open a DMA handle to a shared memory segment which can then be
 * accessed by explicit read and write calls.
 */
function <dma_memory_handle> argo_open_dma_mem( <logical_memory_id> );


// Other platform dependent functions...
```

3.2 Runtime functions

The following functions offer timing predictability and can be called after initialization of the application.

```
/*
 * Push (write) a data item into the specified channel.
 * This function blocks until the channel has free capacity
 * for the pushed item.
 */
function <void> argo_channel_push(<channel_handle>, <item_to_push>);

/*
 * Pop (read) a data time from the specified channel.
 * This function blocks until at least one item is
 * available in the channel.
 */
function <popped_item> argo_channel_pop(<channel_handle>);

/*
 *  Read data from a DMA memory at the specified offset and copy the
 *  data into the target buffer.
 */
function <void> argo_dma_read( <dma_memory_handle>,
            <offset>, <target_buffer> );

/*
 *  Write data from the source buffer into a DMA memory at the
 *  specified offset.
 */
function <void> argo_dma_write(<dma_memory_handle>,
            <offset>, <source_buffer>);

/*
 *  Asynchronously read data from a DMA handle at the specified
 *  offset and copy the data into the target buffer.
 *  If there are pending requests on the DMA handle, the function
 *  implicitly calls argo_dma_wait.
 */
function <void> argo_async_dma_read( <dma_memory_handle>,
            <offset>, <target_buffer> );

/*
 *  Asynchronously write data to a DMA handle at the specified
 *  offset and copy the data into the target buffer.
 *  If there are pending requests on the DMA handle, the function
 *  implicitly calls argo_dma_wait.
 */
function <void> argo_async_mem_write(<dma_memory_handle>,
            <offset>, <source_buffer>);

/*
 * Wait until all pending asynchronous operations
 * on the given DMA handle have been completed.
 */
function <void> argo_dma_wait <dma_memory_handle> );
```

```
/*
 *  Identification routines.
 */
function <id> argo_board_id();
function <id> argo_subsystem_id();
function <id> argo_core_id();
function <id> argo_task_id();
```

4. InvasIC

4.1 The InvasIC platform

The InvasIC platform is a tile-based distributed many-core platform. The tiles are homogeneous clusters that include a local memory and one or more processors connected to a shared bus. Each tile is then interfaced to each other in a meshed structure via an on-chip network. A tile can be considered to be a subsystem being a subcomponent of an InvasIC board. Since InvasIC is realized on an FPGA-prototype, parameters may be varied and different designs of the general concept exist.

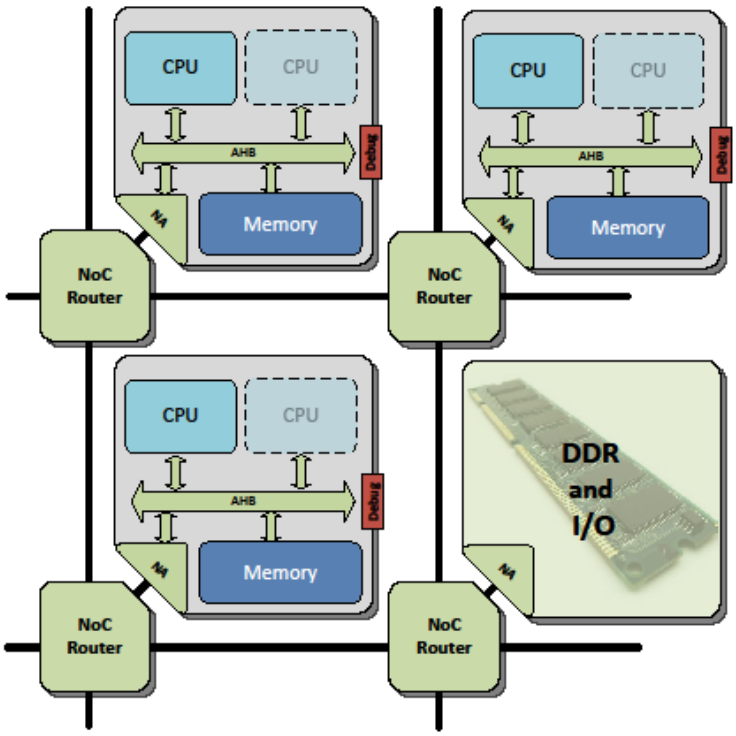

Figure 8: The InvasIC architecture

In contrast to the FlexaWare platform, InvasIC offers a bare-metal system that will be targeted by the ARGO toolchain on a static, low level model of programming. However, the InvasIC target platform will also conform to the programming model and the ARGO toolchain will contain the necessary backend adaptations to bridge all possible gaps. InvasIC will only provide a minimal runtime system and use the constraints and the metadata in the ARGO toolchain, allowing a high level of flexibility for optimizations.

The InvasIC architecture employs Sparc v8 conforming LEON3 cores by Gaisler research as regular processing elements (CPU). As such, a C/C++ cross compiler, provided by Cobham Gaisler [9], based on the GCC and the Newlib embedded C-library version 1.13.1 is used for compilation of regular C source code. Each tile contains a register that can be read to

determine its Tile ID and each processor can read its processor ID. This functionality is provided by two functions specified below.

```
/**
 *  returns the id of the core that currently executes the code
*/
int get_core_id ();

/**
 *  returns the ID of the Tile on which the core resides that currently
 *  executes this code. Tile ID is calculated as (y*x_Dim + x) based on x
 *  and y coordinates in the meshed structure.
*/
int get_tile_id ();
```

Realizing the programming model on this target architecture depends on the language support of the tiles (especially the processors and peripherals) and the communication infrastructure. For the latter, a communication library will be designed that implements the channel concept while for the former, the ARGO toolchain is free to create static, predictable tasks from an input C application and map it to designated cores in the architecture. Note that the InvasIC API will conform to the ARGO programming model and the ARGO API by providing implementations and wrappers for all required features, although not all of them might be shown in detail throughout this document.

4.2 Fundamental programming of each tile

All application code can be incorporated into a single ELF file which is loaded onto each tile. However, it is typically useful to create specific ELF files for each tile since the ARGO toolchain will provide parallelized code that is not necessarily executing the same functions, saving memory in the process. In a single core (per tile) design, there is no contention on the bus and the local memory is free to be used for the single application that is encapsulated in the elf file. In a multi-core (per tile) design, the hardware will use a Round Robin scheme to allow predictable behaviour and the compilation process needs to allocate separate stack and heap resources in the shared memory. At bootup, only core 0 is active and needs to initialize the platform, including the communication resources, and finally activating the other cores by configuring their entry point, specifying the task each core should process.

The InvasIC platform includes debugging IP-cores that allow access to the tile-local bus from external sources. This is used to load applications into the memory of the tiles and monitor execution. It also enables the use of I/O functionality in the application code, e.g. the "*printf*" function is supported and the output will be printed to the console that is connected via the debugging link.

4.3 Data types

Since InvasIC uses homogeneous tiles, the architecture is tailored around the 32bit LEON3 processors. As such, the AHB bus and the NoC typically work on 32bit data and address width lines and links. Besides the regular integer unit, the LEON3 cores also come with an additional floating point unit (FPU) based on the IEEE-754 standard.

4.4 Communication library

Communication between tiles needs to be handled explicitly. For accessing the resources provided by the networks-on-chip a communication library will be developed that allows the communication to be set up and used consequently. The library will be designed to match the ARGO API and thus implement the concept of channels. The network itself already offers guaranteed service connections, for which a worst-case latency and a predictable throughput can be calculated. A predictable network adapter is currently under development that will be accessed by the communication library. This interface between the tiles and the network will be designed to allow predictable creation of channels and consequently their use, enforcing a strict separation from other channels.

The creation of channels is done by calling the “channel_create” function with the same parameters on both communication partners in an initialization routine. Calling this function on the sender will trigger the resource reservation in the network. It is required to specify the ID of the target tile as well as the ID of a local and a remote FIFO-buffer that is not yet used by another channel. The channel_create call is shown with the parameters that are required to configure the hardware and reserve a connection. All the information needs to be generated by the mapping feature of the toolchain and the correct handles assigned to each task.

The push and pop operations may block if the channel is full or empty respectively. This condition is based on the physical implementation that uses buffers at the receiving end of the channel. A schedule that can guarantee, that items will be processed faster than they are generated will not result in any blocking push operations. However, all operations only depend on their own execution and possibly blocking due to connected tasks and never due to unrelated tasks.

```
/**
 *  Creates a channel that transfers integer type data items between the
 *  local tile and a target tile specified by its ID. It is required to
 *  specify the (ID) of the local FIFO and the target FIFO that will be
 *  used by the channel.
 *
*/
int channel_create (int* channel,
                    int target_id,
                    int local_FIFO,
                    int target_FIFO
                    );


/**
 *  Pushes an item onto a channel.
 *  This call blocks until at least one container is available in channel
 *  to put the item into.
 */
int channel_push (int channel, const void * item, int size);

/**
 *  Pops an item from a channel put it in the location referred to by
 *  item.
 *  This call blocks until at least one item is available for consumption.
 */
int channel_pop (int channel, void * item, int size);
```

4.5 Memory hierarchy

The memory model of InvasIC is based on a hierarchy: The first level is represented by an (optional) core specific cache or scratchpad memory. The next level is the tile local memory that is accessible via the AHB. The final level is the large DDR memory that resides on a separate tile.

Application code is typically expected to reside in the tile local memory, together with dynamic memory areas such as stack & heap. Since there is only one task executed on each CPU, there are clear bounds for the allowed memory ranges and their usage. The DDR is intended for large chunks of data that do not fit into the local memory. Accessing the DDR will also be realized based on the channel concept. There will be an additional call in the communication library that triggers a request (read or write) of a specified amount of data at a specified memory location. Following the request, the data is pushed into the output channel or popped from the input channel the same way as tile to tile communication. Initiating a read-request assumes that another channel for the opposite direction was established and is known to both communication partners. An implementation based on global addresses for accessing the DDR, allowing regular load and store operations on this memory, is also envisioned for a future design since this would ease the computation of worst-case delays.

```
/**
 *  Initiates a request on an output channel.
 *  Type can be READ or WRITE.
 *  Size specifies the amount of 32-bit data.
 *  Address specifies the starting address of the request
*/
int channel_request (int channel, int address, int size, int type);
```

5. FlexaWare

FlexaWare is a clustered architecture, consisting of multiple interconnected ‘modules, each hosting one or more subsystems. A subsystem may provide processing elements, hardware accelerators, memories and peripherals. Figure 9 shows an overview of a specific FlexaWare configuration. The interconnect is memory mapped.

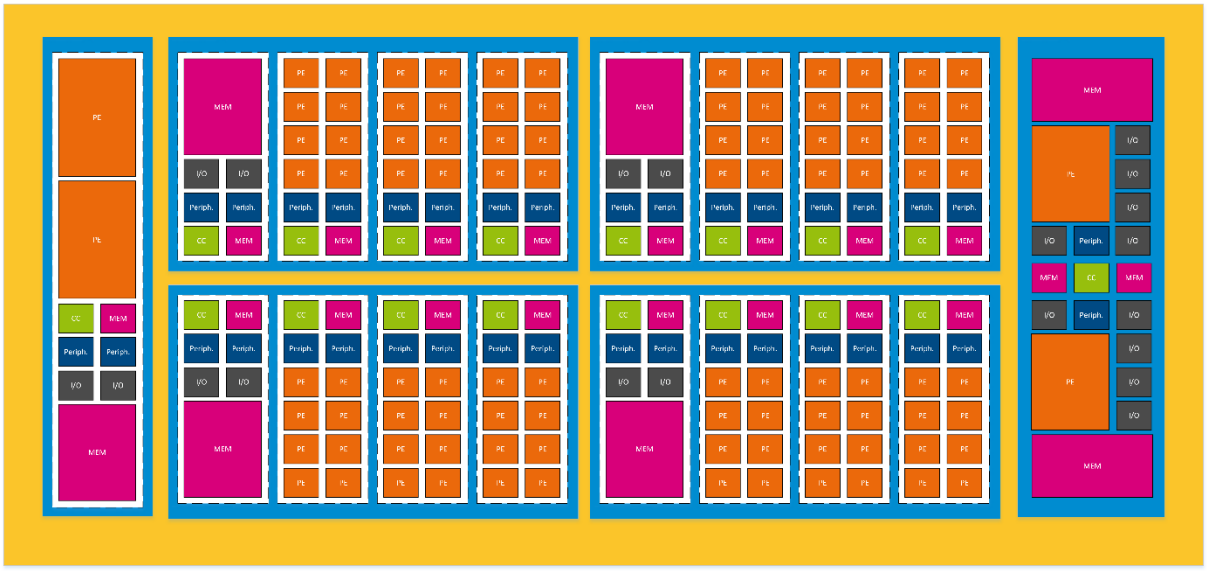

Figure 9: Overview of a specific instance of the Flexaware architecture.

FlexaWare provides an application programming interface (API) compliant with the programming model specified in Chapter 2. The API completely supports heterogeneous architectures through the concepts of data types and activities. Per data type defined, information is provided on its (memory) size on each supported architecture and on the serialization of the data type to enable interoperability between different types of architectures. Activities are the functions that can be executed, either as software or as a hardware implementation (accelerator). The API allows for the creating of tasks and channels, which are the two main constituents to define and organize applications.

5.1 Data types

To support the use and exchange of specific data types across architectural boundaries, a serialization format is adopted to describe data in a form that is understandable in any architecture. The specific serialization format adopted is the eXternal Data Representation (XDR) [7]. XDR is designed to work across different languages, operating systems, and machine architectures. This format is specified in RFC 4506, making older version of the specification obsolete. Converting from the local representation to XDR is called encoding. Converting from XDR to the local representation is called decoding. XDR is implemented as a software library of functions which is portable between different operating systems and is also independent of the transport layer. XDR uses a base unit of 4 bytes, serialized in big-

endian order; smaller data types still occupy four bytes each after encoding. Variable-length types such as string and opaque are padded to a total divisible by four bytes. Floating-point numbers are represented in IEEE 754 format [8].

This format is specified to be an implicitly typed encoding for efficiency purposes. The type of the data is already known in the application itself, and does not need to be added to the serialized data. Some types of data do bring additional fields as metadata. For example in XDR, arrays have a size field. Strings have a field encoding the length of the string. Next to these additional fields that transfer the minimal required metadata, the maximum overhead in size of the encoded data is 3 bytes.

5.2 Activities

Heterogeneous computing entails both processing on elements of various architectures, and communication between them, possibly over a heterogeneous interconnect. In a heterogeneous system, an application may consist of functionality that is implemented on various architectural elements. Additionally, a single function may be implemented on multiple different architectures. This gives additional freedom to the system to execute an application. The heterogeneous nature of applications require an indirection in the requests for functionality that is external to the executing task. An activity is a unique identifier that embodies the functionality of a task implemented for a specific architecture. Both the system and applications may provide a collection of activities that may be used to compose applications. Therefore, the unique identifiers have to be reproducible, such that multiple entities may reference the same functionality.

It is not possible for a FlexaWare application to provide function pointer as arguments to the runtime kernel to support heterogeneity in the architecture of processing elements. Allowing this makes it (too) hard to instantiate functionality on a different core than the function using function pointers was called. On another core, the memory map may be different and/or the hardware architecture may differ. This requires an indirection between the application providing arguments to the runtime kernel in case of functions. A natural choice is to refer to functions with integers or strings instead of function pointers.

In the figure below, a default compilation flow is represented. As the input, C language source files are pre-processed in order to simplify the code and macros in them. An analysis pass over the pre-processed C code extracts the metadata on the activities and data types used in the code. This metadata is stored in two lookup tables; one for the activities and another for the data types. The pre-processed C language code may be compiled for various architectures, which determines whether or not an activity is compatible with the architectures in the target platform. Together, these artefacts are compiled into a single ELF-formatted executable.

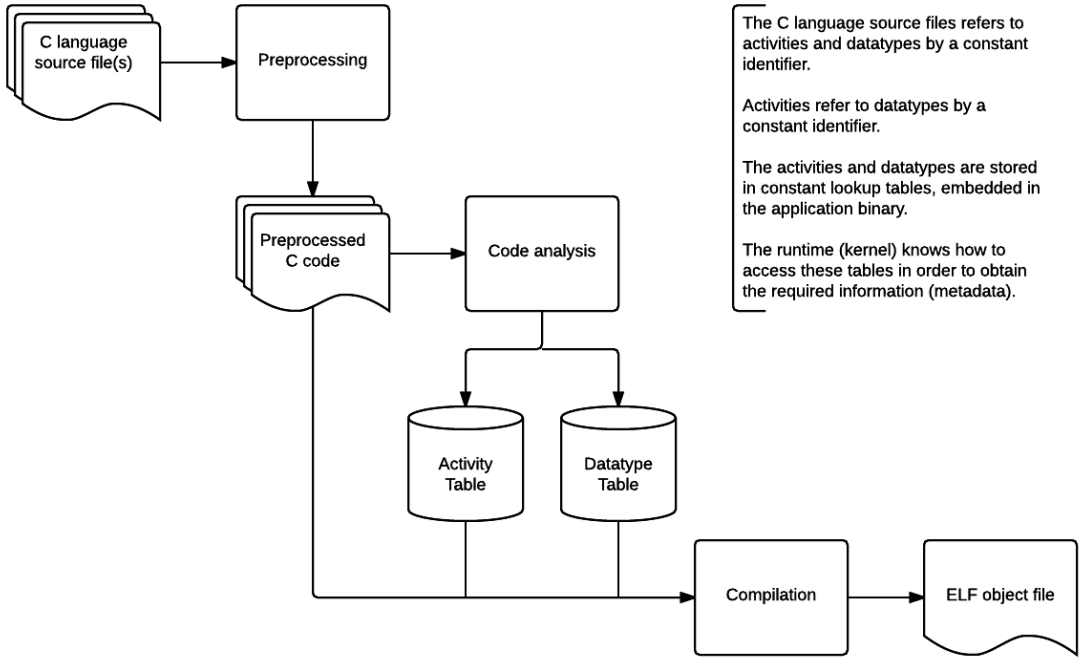

Figure 10: Proposed compilation flow for FlexaWare executables.

5.3 Tasks

The programming paradigm defines that applications are decomposed into tasks. A task instantiates an activity by associating it with a parameter set and a, possibly empty, set of communication channels. Multiple tasks can instantiate the same activity at any time. Due to this association with data and channels, tasks are objects, created by their parent task. A task is first created, then attached to a set of incoming and outgoing communication channels, after which it is started. These three invocations are done by a parent task. Figure 11 shows the application structure of a FlexaWare application, consisting of an initialization part and a processing part; here, the 'Main' task is the parent task that creates the tasks and channels. After starting a task, the run-time is allowed to execute the task's activity. Note that the run-time has some freedom in what activities are performed in parallel, and in which order they are performed. It is not guaranteed that the task starts executing immediately after the creation function returns.

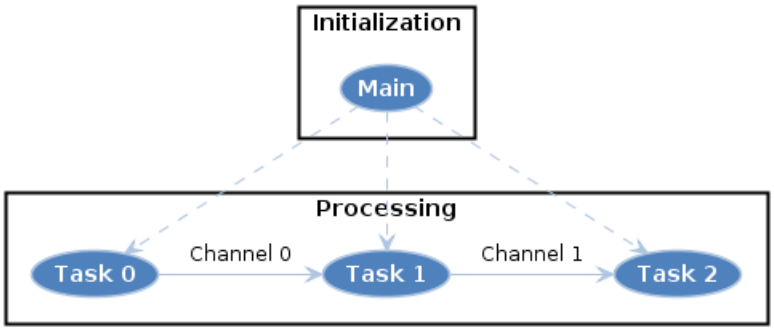

Figure 11: FlexaWare applications consist of an initialization part that dynamically create the structure of the processing part.

Tasks are identified by an integer number. These identifiers are invariant of how and where the application is executed; it is defined at design-time, based on the application structure. If the application structure is static, the identifiers will not change. The task identifier is then defined by its location in the hierarchical graph. The task identifiers are required for their association with metadata contained in the application.

The following API is defined for the instantiation of tasks.

```
/**
 *  Creates a task, that when started, performs the specified activity.
 *  Note that the activity can only be specified with a symbolic reference
 * (name) of the function implementing the activity. It cannot be a pointer
 *  to a memory address.
 */
fw_result_t fw_task_create (fw_task_h *handle, fw_activity_t activity);

/**
 *  Attaches a task to a list of input channels and output channels.
 */
fw_result_t fw_task_attach (fw_task_h handle,
                            unsigned nr_inputs, fw_channel_h in[],
                            unsigned nr_outputs, fw_channel_h out[]);

/**
 *  A task (referenced by handle) is started by providing it with a
 *  matching set of activity parameters. A task definition can be started
 *  multiple times, but only after the task has finished its previous
 *  invocation. The type of parameters supplied has to match the prototype
 *  of the task implementation.
 */
fw_result_t fw_task_start (fw_task_h handle, const void * parameters);
```

5.4 Channels

Tasks communicate with each other through channels. Channels are point-to-point connections, defined by the type of data items that will be exchanged (having a fixed size) and the number of items (the count) that fit into the buffer associated with the channel. A task blocks if it attempts to read data from an empty channel, or when it attempts to write data to a full channel. The capacity of a channel determines the synchronization between communicating tasks. These blocking operations on channels are implicit synchronization primitives. The degree to which tasks execute concurrently is, therefore, directly related to the communication topology and the associated buffering in the channels.

```
/**
 *  Creates a channel that transfers items of a specific data type. The
 *  channel has a maximum capacity for items that are in-flight.
 *  Operations that attempt to transfer items while the channel is full
 *  will block. Operations that attempt to obtain items from the channel
 *  while it is empty will also block.
 *
 *  The lifetime of a channel is identical to the lifetime of its creator.
 *  The user can not tear it down by any explicit means. The FW runtime
 *  takes care of the teardown of the channel when it is no longer useful;
 *  it is destructed when the creator task is finished.
 */
```

```
fw_result_t fw_channel_create (fw_channel_h* handle,
                               fw_type_h type, unsigned capacity);


/**
 *  Pushes an item onto a channel. The channel has to be a member of
 *  the output channels of the calling task. The item to transfer must be
 *  compatible with the serializer specified upon creation of the channel.
 *  This call blocks until at least one container is available in channel
 *  to put the item into.
 */
fw_result_t fw_channel_push (fw_channel_h channel, const void * item);

/**
 *  Pops an item from a channel put it in the location referred to by
 *  item. The channel has to be a member of the input channels of the
 *  calling task. The location referred to by item shall be a valid piece
 *  of memory that is at least the size of a single item as defined by the
 *  channel.
 *  This call blocks until at least one item is available for consumption.
 */
fw_result_t fw_channel_pop (fw_channel_h channel, void * const item);
```

5.5 Standard library functionality

The hosted C environment ships with a collection of functions known as the *standard library*. We provide a subset of functions for all practical purposes. The functions supported are listed below, and relate to output contents on the standard output stream and to manage the memory in the task-private dynamic allocable memory.

```
/**
 *  A function to produce output on the standard output stream according to
 *  the specified format. Whether or not a standard output stream is
 *  available and in what form is implementation specific.
 */
int fw_printf(const char* format, ...);
int fw_vprintf(const char* format, va_list args);

/**
 *  Allocates an amount of bytes from the task-local memory pool.
 *  For a task, this is the only mechanism provided for dynamic memory
 *  allocation.
 */
void* fw_malloc (size_t bytes);

/**
 *  Release the memory obtained from an earlier memory allocation.
 */
void fw_free (void* ptr);

/**
 *  Allocate initialized memory.
 */
void* fw_calloc (size_t nmemb, size_t size);

/**
```

```
 *  Reallocate memory.
 */
void* fw_realloc (void* ptr, size_t size);
```

6. Conclusion

This document describes the first specification of the programming model and APIs that will be targeted by the ARGO toolchain in order generate code for the respective platform. Both the ARGO platforms and the programming model rely on pre-allocation of hardware resources on program initialization. Once allocated, the interference and timing behaviour stays fixed, such that tight worst case times can be computed at compile time. Furthermore the programming model specifies a basic set of primitives to be targeted by the ARGO toolchain while staying platform agnostic in the earlier stages.

The initial programming model and the APIs described in this document will be adapted and revised during the project in order to implement new optimizations and to meet newly arising requirements.

References

[1] WCET aware compilation, TU Dordmund, 2016, http://ls12-www.cs.tu-dortmund.de/daes/en/research/wcet-aware-compilation/motivation/wcet-basics.html.

[2] Rutgers, J.H. (2014) Programming models for many-core architectures: a co-design approach. PhD thesis, University of Twente. CTIT Ph.D.-thesis series No. 14-292 ISBN 978-90-365-3611-0

[3] http://www.cs.colostate.edu/~cs551/CourseNotes/Consistency/TypesConsistency.html

[4] Resource Reservation in Real-Time Operating Systems - a joint industrial and academic position, L. Steffens, G. Fohler, G. Lipari, G. Buttazzo, July 2003, ARTOSS-2003, http://www.win.tue.nl/~rbril/education/2IN20/2004-2005/artoss-2003-reservations.pdf

[5] Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems, J.D. Regehr, PHD thesis, University of Virginia, 2001, https://www.cs.utah.edu/~regehr/papers/diss/regehr-diss-single.pdf

[6] ISO/IEC 9899:1999, International Organization for Standardization, December 99, http://www.iso.org/iso/catalogue_detail.htm?csnumber=29237

[7] XDR: External Data Representation Standard, The Internet Society, May 2006, http://tools.ietf.org/html/rfc4506

[8] ISO/IEC/IEEE 60559:2011, International Organization for Standardization, June 2011, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57469

[9] http://www.gaisler.com/index.php/downloads/compilers

Glossary of Terms

ADL	Architecture Description Language
AMBA	Advanced Microcontroller Bus Architecture
AMP	Asymmetric MultiProcessing
API	Application Programming Interface
ASG	Abstract Syntax Graph
AST	Abstract Syntax Tree
AVX	Advanced Vector Extensions
BUG	Bottom-Up-Greedy
CDFG	Control and Data Flow Graph
CFG	Control Flow Graph
CIL	Common Intermediate Language
CPU	Central Processing Unit
CRISP	Cutting edge Reconfigurable ICs for Stream Processing
DSL	Domain-Specific Language
DTSE	Data Transfer and Storage Methodology
GCC	GNU Compiler Collection
GPP	General Purpose Processor
GSP	General Streaming Processor
DSP	Digital Signal Processor
ELF	Executable and Linking Format
EULA	End User Licence Agreement
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
HDL	Hardware Description Language
HIR	High Level Intermediate Representation

HLS	High Level Synthesis
HPC	High Performance Computing
IMS	Integrated Modulo Scheduling
IR	Intermediate Representation
ISA	Instruction set architecture
JIT	Just In Time
LIR	Low Level Intermediate Representation
LISA	Language for Instruction Set Architecture
LISP	Is a family of computer programming languages
LLVM	Low-Level Virtual Machine
LTI	Linear Time-Invariant
MCA	Multicore Association
MMX	Multi Media Extension
MPSoC	Multiprocessor System on Chip
MPPB	Massively Parallel Processor Breadboarding
NLP	Nested Loop Programs
NP	Non-Polynomial
NoC	Network on Chip
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language
PCCA	Partial Component Cluster Assignment
PIS	Pragmatic Integrated Scheduling
PIP	Parametric Integer Programming
PTX	Parallel Thread eXecution
RHOP	Region-based Hierarchical Operation Partitioning
RTL	Register Transfer Level
RFD	Reconfigurable Fabric Device

SCoP	Static Control Part
SIMD	Single Instruction Multiple Data
SMP	Symmetric MultiProcessing
SoC	System-on-Chip
SSA	Static Single Assignment
SWP	Sub-Word Parallelism
UAS	Unified Assign and Schedule
UMA	Uniform Memory Access
VHDL	Very high speed integrated circuits Hardware Description Language
VLIW	Very Long Instruction Word