Compile-Time Virtualisation and the Virtual Platform: Motivation and Methodology

Ian Gray
University of York
iang@cs.york.ac.uk

Neil Audsley
University of York
neil@cs.york.ac.uk

Leandro Soares Indrusiak
University of York
lsi@cs.york.ac.uk

24th September 2010

Contents

Abstract

This document provides an overview of Compile-Time Virtualisation, a technique developed to assist the development of software for complex embedded architectures. The problems that are encountered during standard embedded development are examined and used to motivate the need for CTV. CTV itself is then introduced, along with the CTV Virtual Platform and its system model. The programming model used in a CTV-based system is described and contrasted with the standard model. Finally, CTV implementations are discussed. This document is designed to give the MADES partners an overview of the CTV methodology so it is necessarily brief. If you would like any areas expanding or any further detail please get in contact.

1 Motivation

The development models used by modern programming languages (such as C++, Java and Ada) all use broadly the same abstraction model that has existed since the development of high-level languages decades ago. High-level languages evolved at a time when the underlying implementation hardware consisted solely of a single processor with a single, logically-contiguous, memory space. Software languages began to hide these details from the programmer because they were static, resulting in the assumption of an *implicit target architecture*. As a result of this, modern languages describe the software running on a single processor from within an implicit architecture. They do not describe the system in which that processor is located, or give any details of how the code is mapped into the architecture. Such details are hidden from the programmer.

As a result of this, if the programmer attempts to write a program for execution on a non-standard architecture, the following problems arise:

- There is no way to map the threads of the input program to the processors of the target architecture. Some languages provide *affinities* but these lack the expressive power to handle dynamic behaviour, do not consider the placement of data, and are outside of the programming model of the language.
- The data items of the program cannot be mapped to the memory spaces of the target architecture. Affinities do not consider data, so variables have to be placed throughout a NUMA with the use of manually-generated link scripts. This can be time-consuming and error-prone and it separates mapping information between code and link scripts leading to potential maintenance issues.
- The program cannot effectively use the communications channels (networks, buses, etc.) of the target hardware because such a concept is not included in the programming model of existing languages. Inter-processor communications have to be hidden behind a specially-written middleware layer that can be difficult to produce and introduces inefficiency.
- Custom hardware elements such as function accelerators, I/O devices, and processors with non-standard instructions sets (such as vector processors or DSPs) cannot be effectively exploited. Languages assume homogeneous processors.
- Due to the complexity and diversity of embedded architectures, code reuse can be very difficult. Existing code written for a given architecture is unlikely to operate on a new architecture, requiring it to be ported.
- Operating systems do not work well with non-standard architectures. Because OS kernels tend to only operate on a single processor, a bottleneck is created in the system in which all other processors have to negotiate with the kernel whenever they require OS services.

Because of these problems, software developers are forced use a *multi-program* development model. In the multi-program model, the complex multi-core architecture is viewed as a set of single-core architectures that interact to execute the larger application. These single-core architectures fit the programming model of the development

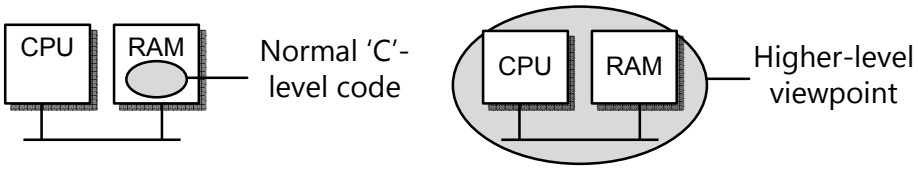

Figure 1: Traditional programming languages describe what runs on a CPU, not the architecture in which the program executes.

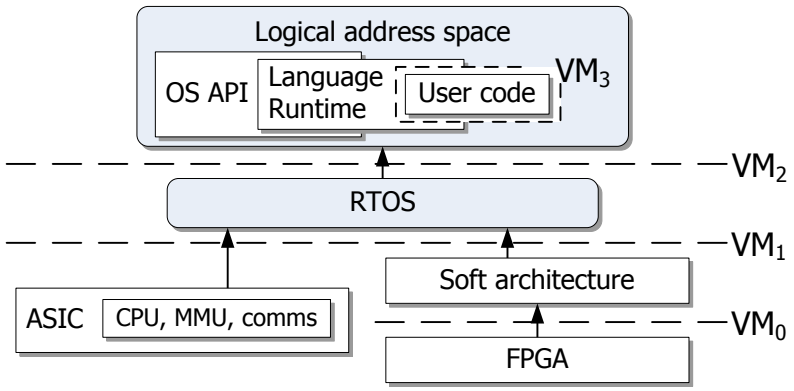

Figure 2: The stack of VMs in general-purpose software development.

languages used (C, Ada, etc.) but now the programmer has to provide a separate program for each execution unit (processor, co-processor, etc.) of the system. Each program must be specially-written for each processor and must be separately designed, compiled, tested, and debugged. The problems with this model are numerous:

- The programmer is forced to manually split their software requirements into a set of separate programs. This must be done early in the software development cycle in order to have time to develop the code, so after testing if the split is found to be suboptimal it can be very difficult and costly to refactor the system.
- Related to the above point, if the hardware design team wish to change the implementation architecture they will force the software design team to refactor their code.
- The system cannot respond well to changing software requirements.
- The communications protocols between the interacting programs must be manually defined and programmed. This can be error-prone and consumes valuable development time.
- It is outside of the programming model of the development language. For example, Ada allows tasks to communicate using a rich set of interaction paradigms designed to minimise development time, increase analysability, and create an expressive programming model. None of these can be used however because they are designed to allow tasks inside a single program to communicate. They do not consider inter-program communications, which are required by the multi-program model.

The alternative to the multi-program model is a *single program* development model, in which the operation of the entire architecture is controlled by a single program. This is the way that software developers are used to programming and has a wide range of advantages:

- The high-level expressive features of modern languages (such as multithreading, interthread communications, etc.) can be used.
- The approach is flexible to changing architectures. The underlying hardware can change without redeveloping the software.
- IDEs, debuggers and other tools are normally not developed for the multi-program paradigm so they are most effective when using the single-program model.
- It is easy to move threads and data throughout the target architecture to balance load in response to changing requirements.
- Code reuse is greatly facilitated.
- Because the programming model os not being broken, rapid functional testing is possible. The program can be tested at a high level of abstraction before it is mapped to the hardware to obtain preliminary profiling information and to test algorithms.

Compile-Time Virtualisation (CTV) is a technique developed to support the development of software for complex embedded architectures using a single-program model without requiring the development of new languages or compilers. CTV allows the programmer to write normal architecturally-neutral code with distributed operating system features such as threading, shared memory, cache coherency etc. and to distribute that program over a complex target architecture automatically.

The mapping to the architecture can be performed *after* the program is written because the programmer's input code is architecturally-neutral – written in the programming model of the chosen language. CTV allows many different code mappings to be explored quickly. For example, if testing determines that a given processor is overloaded, threads can be moved to another processor simply by adjusting the architecture mappings. CTV will split the code appropriately, handling all communications and shared memory use to ensure that the code still operates correctly.

Note that CTV as described in this document will assume the presence of a pre-existing target architecture. An overarching modelling framework (of the kind proposed in the MADES project) can provide the programmer with the capability of describing and generating hardware descriptions. This architecture description can then be fed into CTV to be used as the target architecture.

2 Compile-Time Virtualisation

2.1 CTV Overview

The layers of virtualisation and abstraction that are present in standard software development do not support changing, non-standard architectures, and they insulate the programmer from the architectural information required to efficiently software to the target hardware. CTV replaces these layers with a single virtualisation layer across the entire architecture, termed the *Virtual Platform* (VP), that contains more appropriate abstractions for embedded development.

The VP sits between the programmer's code and the target hardware (or the embedded operating system if one is present). Its purpose is to provide the abstractions that the programmer is used to and are assumed by the programming language (such as a single shared memory space) but in a way that does not lose the architectural information required for an efficient, single-program, implementation. The VP allows the programmer to view mapping decisions at a high level of abstraction, for example where to place threads and data or which communication channels or custom hardware elements to use. The VP ensures that the low-level implementation details that result from these high-level decisions are kept hidden from the programmer.

For example, consider a C program executing in a non-uniform memory architecture. The programmer can declare an item of shared as such:

```
int shared_data[4550];
```

but because the memory architecture is hidden from the programmer they have no influence over where the toolchain chooses to place the data at runtime. If the wrong memory space is used, other processors may not be able to access the data, or access may be very slow due to congestion. To communicate their desired mapping to the linker, the programmer is forced to break the abstraction layers of the language and do the following:

```
int shared_data[4500] __attribute__((section ("sharedmemory")));
```

and then declare the new `sharedmemory` section in a custom link script which is passed to `ld`. If 0x8C000000 is the address at which shared memory starts the declaration looks like this:

```
. = 0x8C000000;
sharedmemory : { }
```

This solution requires the programmer to manually link the data in their system. When using CTV, the placement decision is kept at a high abstraction level and the programmer can simply state to which memory space the data should be mapped. If the programmer does not mind where the variable is mapped, CTV will place it automatically. Other threads will be refactored so that they can access this data without the programmer having to pass pointers or hard-code addresses.

CTV is a *language-agnostic* technique, meaning that the actual development language does not change the way in which the technique is applied. A CTV *implementation* will choose to work with a given source language and compiler toolchain,

but the CTV technique itself operates on a language-agnostic system model, described later in section 3.

The CTV VP has the following three main features:

Compatibility with the chosen programming model The VP is a high-level view of the underlying hardware that presents the same programming model as the source language to simplify development. For example, it may present a single logical address space or uniform inter-thread communication. In essence, as with standard run-time virtualisation the layer is tasked with ensuring that the programmer's code operates correctly without low-level programmer intervention. Because the layer hides low-level implementation details it allows for code to be architecturally-neutral, as the developer does not need to break the abstraction models of the language.

Flexible mappings from the virtual architecture to actual hardware Every virtualisation-based system contains a set of virtualisation mappings. These mappings map elements of the software and virtual hardware onto the actual physical hardware. (For example, threads $\rightarrow$ CPUs, variables $\rightarrow$ memory spaces etc.) In a standard run-time virtual machine (like Java), these virtualisation mappings are implemented by a run-time system and are largely opaque to the programmer. In CTV, the mappings are directly exposed to the source code, allowing the programmer to use their application-specific knowledge to *influence* the implementation of the code and achieve a better mapping onto the target architecture. For example, the designer can choose to place threads that frequently communicate onto CPUs that are physically close to each other, or to place global data in memory spaces that are close to where its it going to be needed. This kind of architectural exploration cannot be performed easily in Java. 'Compatibility with the chosen programming model' ensures that however the programmer adjusts these mappings, the software will still operate correctly. Only its non-functional properties will be affected.

Visibility of custom hardware elements Existing run-time systems hide underlying hardware details, making it difficult to use custom hardware elements. In CTV these elements are exported up to the programmer through the VP at design-time and presented in a form that is consistent with the source language's programming model. This allows these elements to be effectively exploited without extra development effort and in a manner that is consistent with the current programming model. The virtualisation system has enough information to handle marshaling of data, synchronisation, data copying issues etc. and programmer intervention is not required.

An overview of the CTV system is shown in figure 3. The primary difference between a run-time VM like Java and CTV is that by moving the virtualisation to compile-time, run-time overheads are reduced to a minimum. If the hardware design later changes (for example, the addition of a CPU or a new memory layout) a new VP is generated and the same input code is automatically retargeted to use this new architecture.

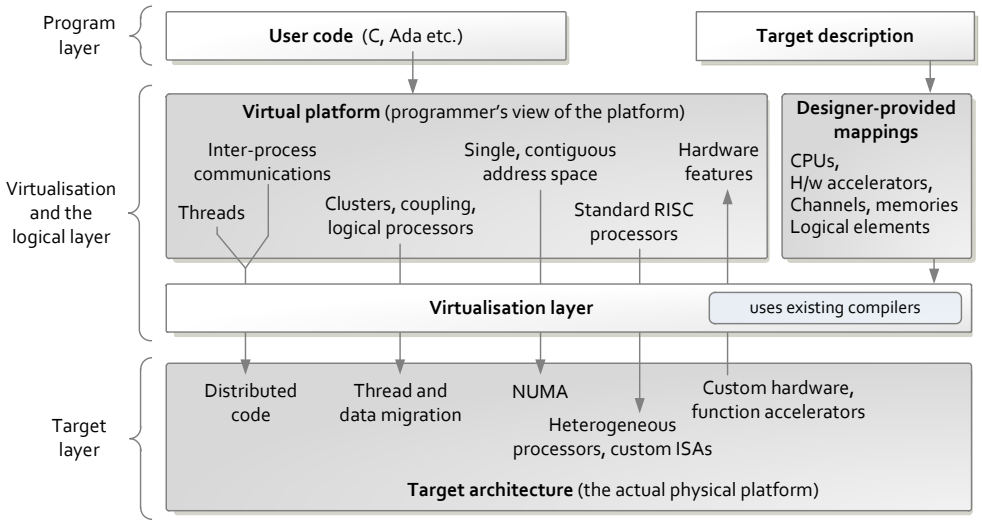

Figure 3: Tools-oriented overview of CTV.

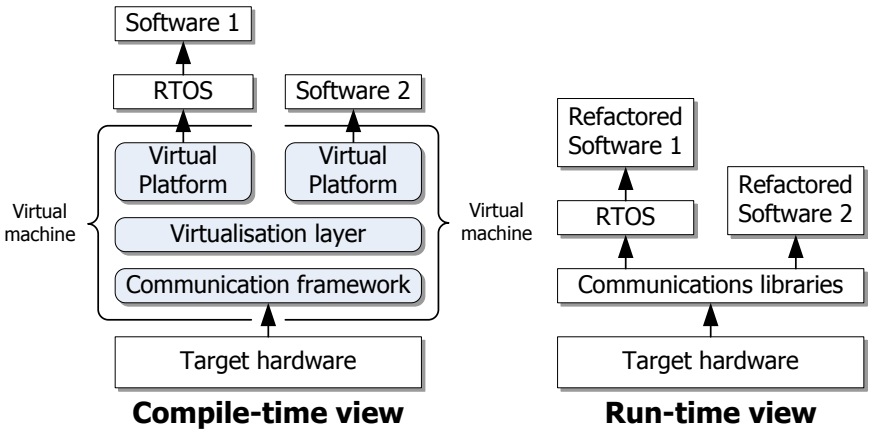

Figure 4: How the compile-time view of CTV differs from the run-time view.

2.2 Moving from Run-Time to Compile-Time

As its name implies, Compile-Time Virtualisation is differentiated from run-time virtualisation systems by the fact that its virtualisation layer only exists during compilation. The lack of run-time virtualisation means that overheads are reduced to a minimum, which is essential when developing for resource-constrained embedded systems. Specifically, the system does not have to store code for an interpreter or run-time support system, and each instruction of the compiled program executes natively on the target hardware. This is illustrated in figure 4.

To understand the semantic difference between run-time and compile-time virtualisation it is helpful to first consider Java, an archetypal example of run-time virtualisation. Java's Virtual Machine (the JVM) interprets the compiled bytecodes of the user's program. The JVM conceptually sits as a layer between the output of the compiler and the capabilities of the target CPU, interpreting the compiler output so that it can be understood by the CPU and executed. The JVM has the effect of making it appear to the programmer as if the CPU is capable of executing Java bytecodes, where in real-

ity it cannot. In other words, because the virtualisation exists at run-time, it extends the *run-time* capabilities of the system.

In contrast, CTV's virtualisation exists at compile-time which allows it to instead extend the *compile-time* capabilities of the system. By sitting as a layer between the user's uncompiled code and a unmodified target compiler, CTV makes it appear to the programmer as if a given compiler can take its standard input language but efficiently target a range of architectures that it normally cannot. CTV uses unextended, unmodified compilers yet it allows the programmer to effectively target complex hardware from the source language level that would normally require the use of hand-coded assembly, custom link scripts, and other abstraction-breaking techniques.

CTV is a way of extending the capabilities of a pre-existing language and compiler in a controlled way to meet changing future demands. This vastly helps code reuse, because existing code is not made obsolete when a new architectural paradigm is developed that would previously have required a new language or language extension.

3 CTV System Model

This section defines the system model used by CTV to represent high-level features of embedded software, the typical embedded architectures that such software runs on, and the mappings between the two. The programming language that the developer is using must be cast in terms of this system model so that it can be used with CTV. Currently C, Java and Ada have been cast into this model, and a CTV implementation exists for C. It is likely that the MADES project will develop a CTV implementation for another language.

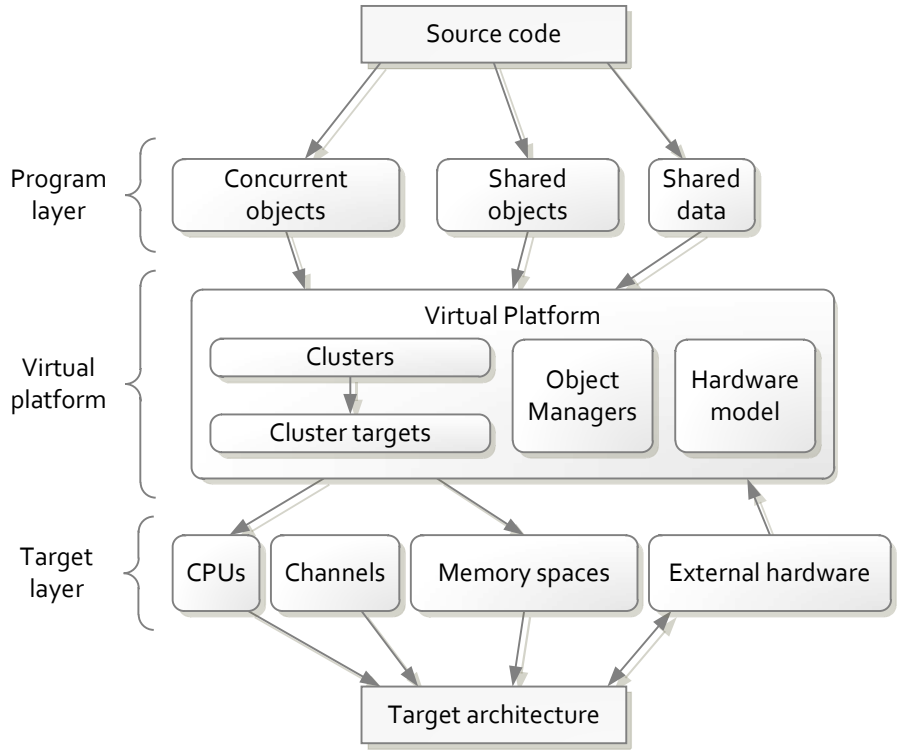

Figure 5: CTV's system model shows the introduction of the Virtual Platform

Figure 5 shows the CTV system model. This model is composed of three layers:

- Program layer (section 3.1): Describes the input program in terms of the threads and data items that it is composed of.
- Virtual Platform (section 3.2): Maps elements of the program layer to elements of the source layer.
- Target layer (section 3.3): Describes the target architecture of the system, onto which the clusters of the logical layer are mapped.

These layers are described in the following sections.

3.1 Program Layer

The program layer (depicted as the top layer in figure 5) expresses the input program as three sets of objects:

- Concurrent objects: A set of the units of concurrency of the source language. Concurrency is an essential modelling primitive, motivated by the trend towards highly-parallel architectures. Examples of concurrent objects are Ada tasks, Java threads and C/C++ pthreads.
- Shared objects: Passive constructs that implement remote procedure call (RPC) semantics that are called by the concurrent objects of the system. Shared objects allow multiple concurrent elements to synchronise with each other and coordinate their execution to avoid race conditions and the corruption of shared data. Examples are mutexes and condition variables from pthreads, protected objects from Ada, and synchronised objects in Java.
- Shared data: Data items that are read and written by the concurrent objects of the system.

The program layer does not explicitly model the communications between concurrent objects (threads). Communications channels are modelled by the target layer as an implementation target rather than a system specification and CTV makes use of the available communications channels as appropriate.

3.2 Virtual Platform

The VP sits between the program layer and the target layer and is responsible for mapping elements of the source code to the target architecture and for expressing controlled dynamic behaviour in the form of thread and data migration.

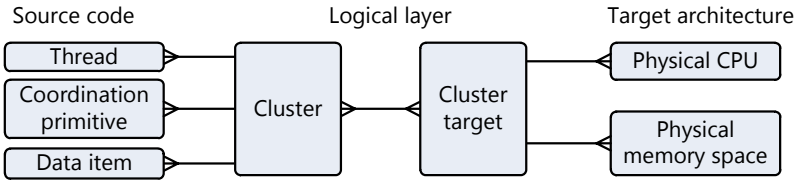

Figure 6: The mappings of the VP

The VP defines two sets:

- Clusters: Clusters are used to inform the toolchain and runtime that a group of concurrent objects, shared objects and data items are tightly-coupled and should therefore execute closely together. Typically, a cluster represents a sub-problem of a larger application. Intra-cluster communications requirements are high and time-critical whereas inter-cluster communications are less common and not as critical to the overall throughput of the system.
- Cluster targets: A cluster target (CT) is an abstraction of the processing elements and memory spaces of the target layer. Informally, CTs are used to

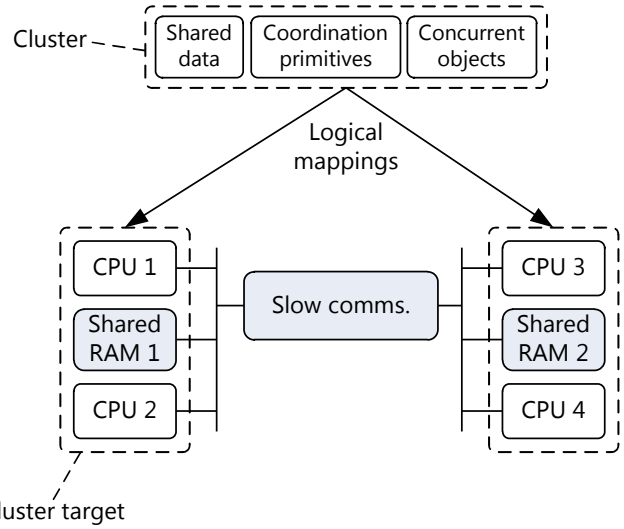

Figure 7: Example showing the use of clusters and cluster targets.

state (from knowledge of the implementation architecture) that a given group of processors and memory spaces are a good location for executing a cluster. CTs are logical concepts distributed across multiple physical processors. Code executing on a CT may actually execute on any of its physical processors. Similarly, data stored in a CT may be actually stored in any of its physical memory spaces.

The two-layered approach of the VP has a number of advantages, as illustrated in figure 7 which gives an example of their use. In the depicted architecture, the programmer wants to express that due to the slow link in the centre of the architecture, tightly-coupled threads should execute as a group on either CPUs 1 and 2 *or* CPUs 3 and 4, but not a mixture of the two as this will require large amounts of data to be transferred over the centre link. Equally, the data items used by these threads should be stored in the appropriate memory bank. This is a common problem that has not yet been addressed by existing approaches.

The programmer can partition their program into clusters of interacting threads and data, and their target architecture into clusters of processors and memory spaces that can execute those clusters. Then, the VP mappings allow precise control of computation and data placement where it is required, and powerful cluster-based control for more dynamic areas of systems.

The second benefit to this approach is that it provides a clean separation of concerns between the application programmer and the hardware designer. The source layer and the clusters into which it is split are completely architecture-neutral. They are used to express the coupling of the software alone, and can be determined by the programmer without any knowledge of the final target architecture. Similarly, CTs are defined by the hardware engineer without any knowledge of the software that will be executed on the target layer. It is only the mapping between clusters and CTs that requires knowledge of the entire system. This is in contrast to locale and affinity -based approaches where hardware-specific mapping information is prematurely included in the software specification.

Note that the clusters in this model are not the same as system partitions. The memory model exposed is still a shared memory model, clusters are used by the programmer to influence the behaviour of the toolchain and to provide more high-level mapping information.

3.3 Target Layer

The target layer provides a high-level view of the target architecture as four sets of architectural elements:

- Processing elements: The processors of the target architecture.
- Communication channels: Hardware features that transfer data between processing elements.
- Memory spaces: The distinct memory spaces in the system.
- Other hardware elements: Custom hardware elements of the target architecture (such as function accelerators) and I/O devices.

4 VP Requirements

The purpose of the VP is to allow the architecturally-neutral software of the source layer to execute correctly on highly non-standard target architectures of the target layer. The requirements of the VP therefore can be determined by enumerating the ways in which the architecturally-neutral code can fail and remedying them. In all cases, code will fail to execute correctly when the underlying architecture does not meet the implicit assumptions made by the language's programming model. (Recall that CTV is language-agnostic, so different source languages can be used.) Languages like C, Ada and Java therefore require the following features of the VP:

Provision of a single logical address space over a NUMA Programming languages assume the presence of a single logical memory space in which the data of their program is stored. In embedded architectures this is frequently not the case and a distributed shared memory system must be provided to support the language. Also, the presence of shared memory and caches requires the implementation of cache coherency algorithms.

Universal communications between all system elements Related to the assumption of a single shared memory space, languages assume the availability of universal communications between all computing nodes of the architecture. The implicit architecture is that of either a totally-connected grid of processors and memory spaces, or a single shared bus. The VP must provide universal communications that can route messages between all nodes of the system, as required by the source application.

Thread and data migration The VP also provides facilities for thread and data migration to support more dynamic systems.

5 Restrictions on Input Code

CTV does not support completely unrestricted input code due to its use of a single-program model to describe the operation of the entire system. The primary restriction is that the items modelled by the program layer (concurrent objects, shared objects, and items of shared data) have to be declared compile-time static in order for them to be mapped at by the CTV system to elements of the hardware. The reason for this restriction is that the CTV mapping process executes at compile-time, taking threads and passing them to various compilers depending on the processor that thread is mapped to. If the threads are not declared statically then CTV cannot do this. Dynamic run-time-created elements are still supported, but they are restricted to execute only on the processor that created them (which is exactly the same as in the multi-program model).

The hardware mappings (thread $\rightarrow$ processor, variable $\rightarrow$ memory, etc.) do not have to be compile-time static, but the more static they are the more efficient the resulting system is. The logical layer (section 3.2) is used to describe dynamic mappings that allow threads or data to migrate throughout the system at runtime. The more static that the mappings are, the more specialised the VP can be. For example, if thread a needs to communicate with thread b and thread b is statically-mapped, then a always knows which communication channel to use to contact b. This will therefore be hard-coded into a's object code when it is run through the VP at compile-time. However if b is dynamically-mapped then the VP is forced to implement a discovery protocol to locate b before any communication can take place. Static mappings (which are common for embedded systems) allow the VP to provide a highly-targeted implementation that does not suffer from the kind of excessive run-time overheads that would be seen in a middleware-based system.

Other restrictions are more pragmatic and relate to the specific programming language being used. For example, Anvil (an implementation of CTV for the C programming language, described in the following section) requires that the input code only use pointers that can be statically analysed. Arbitrary pointers are restricted because the compiler cannot tell whether they are being used to access shared variables etc.

Similar to this, there are many languages that support low-level I/O constructs, for example Ada can access external devices through the use of representation clauses. Such constructs are not required when using CTV because the VP presents a higher-level interface to I/O and custom devices, but they are not restricted either. If such low-level code is used then the programmer is responsible for ensuring that it operates correctly. For example, if the memory map is changed so that devices appear at different addresses, accesses through the VP will automatically update to reflect this change whereas native constructs will not and must be updated manually.

6 Anvil: an Implementation of CTV

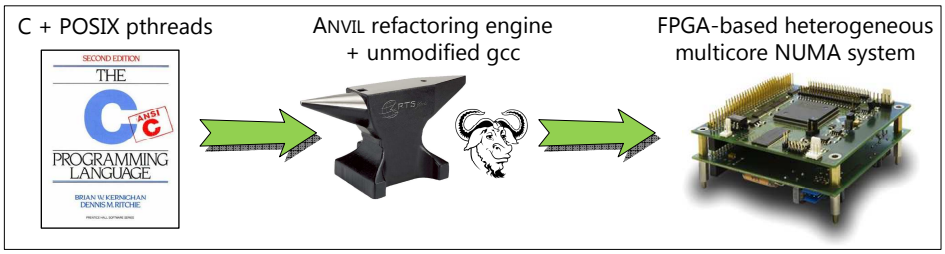

Figure 8: Overview of Anvil, an implementation of CTV.

This section describes *Anvil*, an implementation of CTV that takes applications written in ANSI C as its input. The programmer makes use of the POSIX pthreads library to describe multiple threads of control and their interactions (because C does not have native threads and cannot otherwise describe concurrency). Anvil maps its input programs over FPGA-based, complex, multi-core architectures with non-uniform memory, custom on-chip interconnect, and non-standard hardware features such as function accelerators. Due to the fact that its input code is standard ANSI C, Anvil uses an unmodified version of the `gcc` compiler internally for object code generation. Input code is extensively refactored so that `gcc` is presented with source code that will operate correctly on the target architecture.

Anvil is discussed in this section to illustrate a way in which CTV can be implemented. Anvil is not likely to be the implementation used in the MADES project as it was developed purely as a research implementation to explore the development of CTV.

6.1 Anvil Toolchain

As a compile-time system, Anvil is centred around a source-to-source refactoring engine which takes the programmer's architecturally-neutral input C code and refactors it for execution on the target architecture. Anvil is comprised of the following parts (shown in figure 9):

- AnvilADL. A simple architecture description language that is used by the programmer to describe their code in terms of the CTV system model (section 3). AnvilADL allows the definition of all the elements of the source, logical and target layers, and to describe the mappings between them. This is presented along with the input code to the rest of the Anvil toolchain. AnvilADL is briefly described in section 6.2.
- Refactoring engine. Anvil's refactoring engine is a full C parser and static analysis tool that understands the semantics of the pthreads library. It is used to analyse the input source code and apply source-to-source transformations to turn the input single-program into code suitable for execution on the target architecture. The result of this stage is a set of programs (one for each target processor) which implement the same functionality as the input program. The refactoring is guided by the programmer though the mappings in the AnvilADL description of the system. The refactoring engine is described in section 6.3.

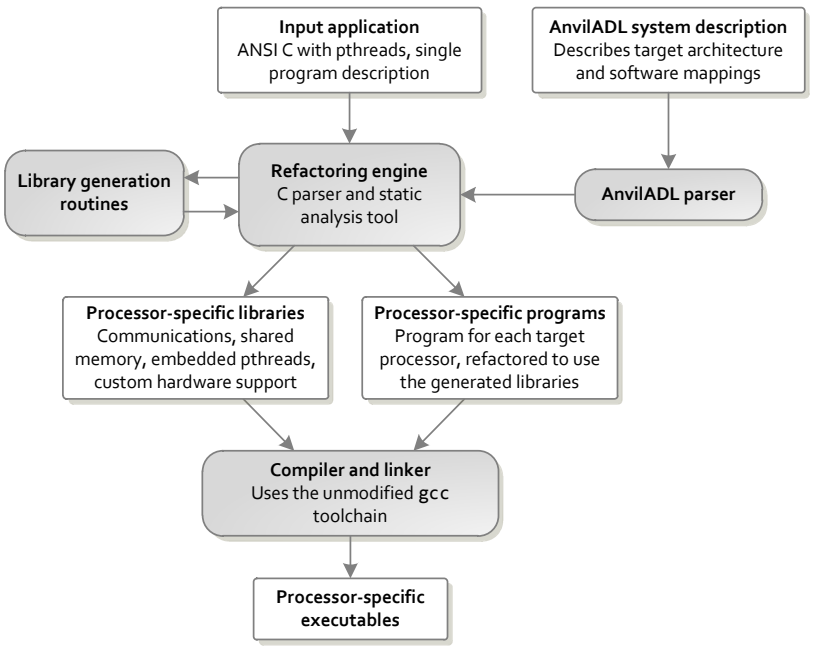

Figure 9: Overview of the main processes in the Anvil system

- Architecture-support libraries. To support the CTV system model Anvil's VP is required to implement a range of OS services, including cache coherency and shared memory systems. The code to do this is provided in a range of libraries that are included as necessary at compile-time by the refactoring engine. This also includes support for elements of custom hardware.

6.2 AnvilADL

AnvilADL is a simplified architectural description language that is used by the programmer to represent their system in terms of the CTV system model. It is used to define the following features:

- Features of the source application (how it is represented in terms of concurrent objects, shared objects and shared data)
- Features of the target architecture (processors, channels, memory spaces, custom hardware)
- Logical elements (clusters, cluster targets)
- Mapping information (source layer objects $\rightarrow$ clusters, clusters $\rightarrow$ cluster targets, cluster targets $\rightarrow$ target layer objects)
- Lower-level detail of target layer objects (for example a processor has attributes that define its type and how it is connected to memory)
- The connectivity and topology of the target architecture (the way in which memory spaces and channels are connected to processors)

```
(* AnvilADL syntax definition *)

(* Top-level non-terminal *)
description ::= {declaration | assignment}

(* Declarations *)
declaration ::= identifier_list ":" feature_category [iptype];
identifier_list ::= identifier | identifier "," identifier_list
feature_category ::= "processor" | "memory" | "hardware" | "channel" |
    "cluster" | "target" | "manager"

(* Assignments *)
assignment ::= lvalue "=" rvalue ;
lvalue ::= identifier "^" attributename
rvalue ::= literal | parameterised_instance | lvalue

(* Complex types *)
parameters ::= "(" list ")"
array ::= "[" list "]"
list ::= list_item |  list_item "," list
list_item ::= parameterised_instance | literal
parameterised_instance ::= identifier [parameters]

(* Base types *)
iptype ::= string
literal ::= number | string | array | boolean | "None"
boolean ::= "True" | "False"

(* Identifiers are case insensitive, start with a letter, and contain
letters, digits or _. Strings are delimited with double quotes
numbers can be in decimal (687), hex (0x2AF) or binary (0b1010101111) *)
```

Figure 10: EBNF of AnvilADL's syntax

- Internal features of CTV's communications and coordination systems that are outside the scope of this document.

The ADL is responsible for defining the VP that is used in an Anvil system. The programmer uses AnvilADL to express the way in which their code should be mapped to a given architecture. This is then used to influence the operation of the rest of the Anvil system. The same input code can be mapped differently, or to a different architecture, by providing different AnvilADL. A full discussion of AnvilADL is outside the scope of this document. Instead, examples are provided that show the way that it can be used by the programmer.

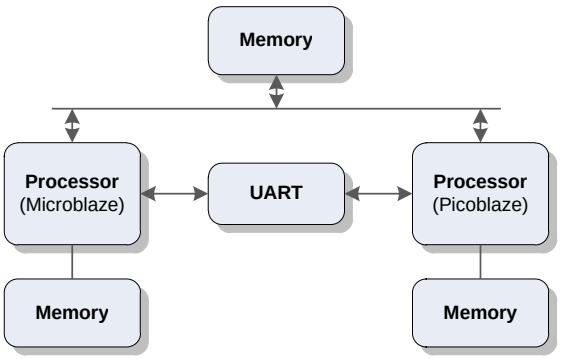

Figure 11: Example heterogeneous NUMA architecture

The first step in creating an AnvilADL description is to declare the processors and memory spaces of the target architecture. For example, the architecture shown in figure 11 can be described as follows:

```
cpu1 : processor Microblaze;
cpu2 : processor Picoblaze;
cpu1^barrelshifter = True;
mem1, mem2, sharedmem : memory BlockRAM;

mem1^size = 8388608;
mem2^size = 8388608;
mem1^width = 32;
mem2^width = 32;
sharedmem^size = 256;

cpu1^memory = [mem1(0x00000000)];
cpu2^memory = [mem2(0x00000000)];
cpu1^extramemory = [sharedmem(0x02000000)];
cpu2^extramemory = [sharedmem(0x02000000)];

myuart : channel UART;
myuart^baud = 9600;
myuart^uartsettings = "8N1";
myuart^endpoints = [cpu1(0x80000000, "mem", 0),
  cpu2(0x80001000, "mem", 0)];
```

Code can then be mapped over this architecture as follows. Consider the following program:

```
int shared_data[50];
```

```
void threadfunc(void) {
   ...
   //Access to shared_data
   ...
}

int main(void) {
   pthread_t thread1;
   pthread_t thread2;
   pthread_create(&thread1, 0, threadfunc, 0);
   pthread_create(&thread2, 0, threadfunc, 0);
   ...
}
```

This program creates two identical threads (out of the `threadfunc` function) called `thread1` and `thread2`. These threads access a shared data item called `shared_data`. This can be mapped over the target architecture above using a description such as the following:

```
cpu1^threads = ["main", "main.mythread"];
cpu2^threads = ["main.thread2"];

sharedmem^variables = ["shared_data"];
```

This description places the main thread (the program's entry point) and `thread1` on processor `cpu1`, and places thread `thread2` on processor `cpu2`. The shared data is placed in the shared memory space of the two processors. All issues of coherency and inter-thread communication are handled by the Anvil refactoring engine. This starts to highlight the power of CTV, because if it is determined that the system is running too slowly the description can be changed and an entirely new mapping evaluated immediately.

6.3 Anvil Refactoring

The Anvil refactoring process is shown in figure 12. The first phase involves reading the input program in terms of the CTV system model. Anvil parses the incoming source code, generates the abstract syntax tree and symbol tables, and from this information determines the threads, mutexes, condition variables and shared data that are present in the source code. These are the elements of the program layer (section 3.1) that are to be mapped over the target architecture.

According to the mappings in the programmer's AnvilADL description, the input program is split into a set of programs, one for each processor of the system. A new `main()` function is created for each of these processor-specific programs which sleeps until it is woken by the appropriate `pthread_create` call elsewhere in the system. The original `main()` function (the program's initial thread) remains unchanged. Most of the processors in the system will only need a small subset of the input code, usually the body of its thread and whichever library functions are called. This is determined using reachability analysis and call-graph generation, although `gcc` and `ld` can also do this automatically.

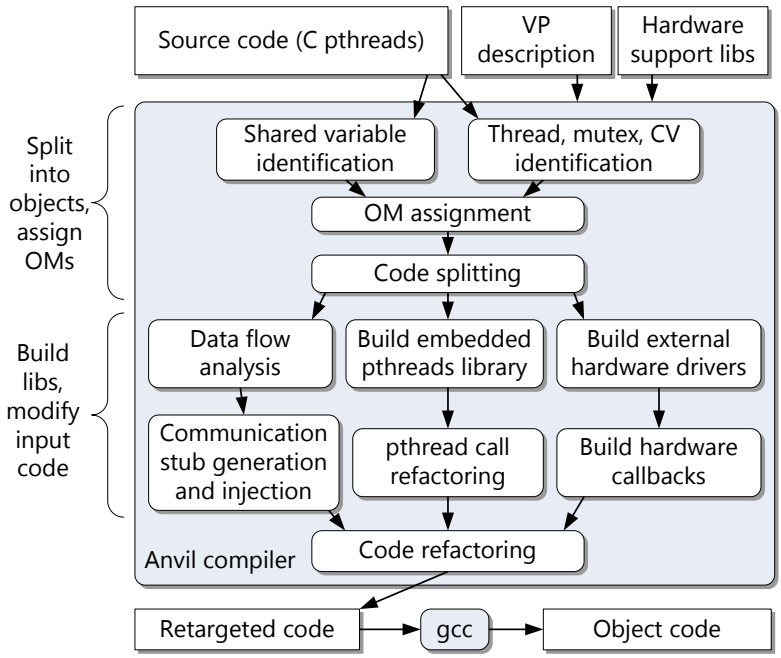

Figure 12: The main processes inside the Anvil compiler

Now Anvil has generated a program for each processor, but they will not correctly execute because they were written for the programming model that is exposed by the Anvil VP. That is, they assume a shared global memory space, universal communications and coherent caches. None of these assumptions are likely to be the case in a complex, non-standard, embedded architecture. To solve this problem, three support libraries are generated by the refactoring engine according to the target architecture and AnvilADL mappings to make efficient use of the underlying hardware. The details for this are outside the scope of this document, but further details can be provided if required. The generated libraries are a shared memory system, an embedded pthreads library, and driver code for custom hardware elements. A version of the libraries is constructed for each processor in the system. These processor-specific libraries are optimised to use the memory, communications and hardware available to the processor.

The following code fragment gives an example of how this works. The code is a function from Anvil's communication library which is used to send messages to other processors in the system. It is generated dynamically by the refactoring process according to the target architecture description (the AnvilADL). In this system, processor 0 could communicate with four other processors over a range of communication channels, so the function is built to reflect this. If the communications topology changes then functions such as this one are regenerated and the rest of the system can still operate correctly. Note that no run-time checks are needed so overheads are minimal.

```
void _anvil_send_to_cpu(int targetcpuid, int *packet, int len) {
 int x;
 for (x = 0; x < len; x++) {
  switch(targetcpuid) {
   case 0:
```

```
     break;
    case 1:
     mbox_write(0x80000000, packet[x]);
     break;
    case 2:
     mbox_write(0x80001000, packet[x]);
     break;
    case 3:
     can_write(0x80002000, 1, 2, packet[x]);
     break;
    case 4:
     uart_write(0x80004000, packet[x]);
     break;
    default:
     break;
   }
  }
}
```

Once the processor-specific support libraries have been built, the final refactoring stage involves refactoring the code of the processor-specific programs to use to them. The original pthreads calls are replaced with calls to the embedded pthreads library, and accesses to shared memory are surrounded by calls to the shared memory system which fetch and update shared data. Cache coherency is handled automatically. The shared memory system is illustrated with the following simple example. Consider this code:

```
void main(void) {
  x = x + 1;
}
```

Here, if `x` is a shared variable then Anvil transforms the code into the following:

```
1  extern _anvil_a_var_t _anvil_accessedvars[];
2  extern _anvil_var_t _anvil_managedvars[];
3
4  int x; //Local space for remote data
5
6  void main(void) {
7    _anvil_accessedvars[0].data = (unsigned char *) x;
8
9    _anvil_read_sv(0, 0, 0, 4); //id, offset, len, bytes
10   x = x + 1;
11   _anvil_write_sv(0, 0, 0, 4);
12 }
```

Lines 1 and 2 declare external references internal data structures of the shared memory system. Lines 4 and 7 show how these structures are populated with suitable pointers. Lines 9 and 11 are the injected read and write calls that fetch and update shared data items from the remote memory space which stores them. The read and write functions make use of functions such as `_anvil_send_to_cpu` from the previous example to transmit the actual data. Many more complex scenarios exist, but the above example serves to illustrate the basic concepts of the refactoring process.

7 Conclusions

This document has shown how Compile-Time Virtualisation (CTV) helps to extend the programming model of a high-level language to more effectively target changing, application-specific implementation architectures. When using CTV, the programmer writes code for the *virtual platform*, a virtualised view of the underlying hardware with a simpler programming model than the true platform. Code written for execution on this VP is refactored and then compiled by the source language's existing compiler, which does not need to be modified. The power of the technique comes from the fact that the mappings in this virtualisation layer can be *influenced* by the programmer to guide compilation and create a more efficient system. CTV's virtualisation layer exists at compile-time only, thereby reducing run-time overhead to a minimum. CTV does not require any form of central control and is naturally parallel, resulting in a system that can theoretically scale to any size supported by the implementation fabric. An example CTV implementation called Anvil is shown which can target multi-threaded C code to complex multi-core architectures with custom memory layouts and non-standard hardware elements such as function accelerators.

CTV copes very well with the changing architectures of modern embedded systems. Virtual Platforms can be defined that expose and utilise these new architectures without needing to develop new languages and toolchains or break compatibility with legacy code.