Exposing Non-Standard Architectures to Embedded Software Using Compile-Time Virtualisation

Ian Gray
Department of Computer Science
University of York, York, U.K.
ian.gray@cs.york.ac.uk

Neil C. Audsley
Department of Computer Science
University of York, York, U.K.
neil.audsley@cs.york.ac.uk

ABSTRACT

The architectures of embedded systems are often application-specific, containing multiple heterogenous cores, non-uniform memory, on-chip networks and custom hardware elements (e.g. DSP cores). Standard programming languages do not use these many of these features natively because they assume a traditional single processor and a single logical address space abstraction that hides these architectural details. This paper describes Compile-Time Virtualisation, a technique which uses a virtualisation layer to map software onto the target architecture whilst allowing the programmer to control the virtualisation mappings in order to effectively exploit custom architectures.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.4 [**Programming Languages**]: Processors—*Retargetable Compilers*

General Terms

Design, Languages, Performance

1. INTRODUCTION

It is becoming increasingly common for the architectures of embedded systems to include application-specific hardware features such as function accelerators, non-standard memory layouts, or custom interconnect. Standard programming languages do not use these features natively, as they assume a traditional single processor and a single logical address space abstraction that hides these architectural details. Hence, custom hardware is accessed through error-prone, ad-hoc techniques that are external to the abstraction model of the source language (custom libraries, assembly programming, manual linking etc.). This paper proposes Compile-Time Virtualisation (CTV) as a solution. CTV uses a virtualisation layer to map high-level software onto the target architecture. It allows access to custom hardware by allowing the programmer to control how language constructs from the virtualised model are implemented on the actual target architecture.

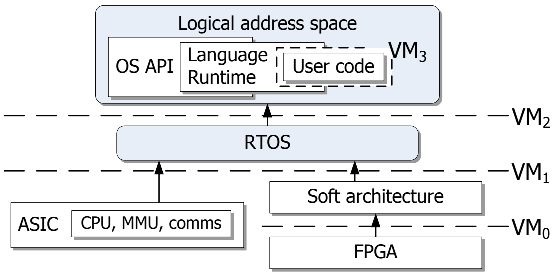

Figure 1: The stack of VMs in the standard general-purpose architecture.

A key motivation for CTV is the rise in complexity of embedded system platforms. Due to the general trends of Moore's Law, hardware designers have become able to integrate a larger number of elements into a single design. This has become known as the System-on-Chip (SoC) paradigm [13] and is a divergence from the common single-CPU single-memory space assumed by most programming languages. This, coupled with limits on clock speed and memory access times has led to an explosion in the variety of on-chip architectures being deployed. A major trend is towards more specialised, application-specific designs that may contain multiple heterogenous processing elements [2, 15]; Networks-on-Chip [10]; non-standard memory hierarchies [12]; DSP cores and SIMD units [17, 24]; and reprogrammable elements using FPGA-based technology [1].

Programming heterogeneous architectures presents a fundamental challenge for the development of embedded software. Figure 1 shows the standard general-purpose programming model expressed as a stack of virtual machines. Each virtual machine hides underlying details to simplify programming and insulate the programmer from implementation changes. VM_1 is built from the actual hardware and presents a single contiguous address space and in-order execution of opcodes. VM_2 is presented by the real-time operating system (RTOS) so that each process believes it has sole control over the CPU, access to atomic actions, and other RTOS features. VM_3 represents the language-level virtual machines of languages like Java and Smalltalk [23].

Unfortunately these VMs add inefficiency and because they do not allow easy access to underlying hardware it becomes difficult for code at higher layers to access custom hardware without the use of specialised libraries. When custom hardware elements are introduced they either cannot be exploited (as with function accelerators) or they cause architectural assumptions of the VMs to fail and user code no longer functions (as with non-contiguous memory layouts). This paper argues that although the programmer may not require complete control over the implementation of their code, they do want to be able to *influence* compilation to better suit their target design. To this end, *Compile-Time Virtualisation* (CTV) amalgamates VMs 0, 1 and 2 into a single virtualisation layer, termed the *Virtual Platform* (VP), the operation of which can be controlled by the application programmer via the compiler.

The paper summarises work related to this field in section 1.1. It then describes Compile-Time Virtualisation in section 2 and Anvil, an example CTV implementation, in section 3. Section 4 describes the ways that Anvil can be used to target custom architectures and presents an evaluation of its capabilities.

1.1 Related work

From observing modern architectural trends, it can be determined that embedded development languages must be able to exploit four main architectural features:

- **Parallelism:** multi-core, co-processors.
- **Memory:** memory spaces, shared memory, caching, coherency.
- **Unique hardware features:** co-processors, function accelerators, communication channels.
- **New paradigms:** streaming languages, data path computation, actor-oriented programming.

In an attempt to utilise these features, much recent work has considered the development of new languages or language extensions. OpenMP [4] and Chapel [3] attempt to extract parallelism from source code to utilise high-performance computing architectures. Sequoia [12] is an extension to C which aims to ease programming for non-uniform memory architectures (NUMA). SystemC [18] and SpecC [14] are examples of system-level languages that can be used to help the programmer to better utilise features of the underlying hardware, such as hardware accelerators and reconfigurable logic. Some work attempts to give the programmer different programming paradigms that better suit novel architectures. Streams-C [16] and StreamIt [27] add semantics to C for describing data paths and computational kernels for stream programming whilst there is a recent interest in general purpose languages that execute on graphics processors, such as CUDA [21] and OpenCL [20].

Unfortunately, such languages tend to only concentrate on solving a single problem. For example Sequoia only targets non-standard memory layouts and does not include provisions for easy access to custom hardware. This paper contends that due to the rapid rate of hardware evolution it is neither feasible nor sustainable to continue developing new languages with associated compilers and toolchains for each new hardware element and paradigm that appears. There is also a fundamental semantic discontinuity encountered when attempting to extend a software language with architectural information, as these approaches have done. Programming languages are introspective by design, meaning they are only concerned with internal details such as data structures and statements. In general they cannot describe features external to themselves that define the system in which the code will execute, or how it should be implemented. Compile-Time Virtualisation avoids this problem by introducing a holistic abstraction model that gives the programmer greater control over the compiler.

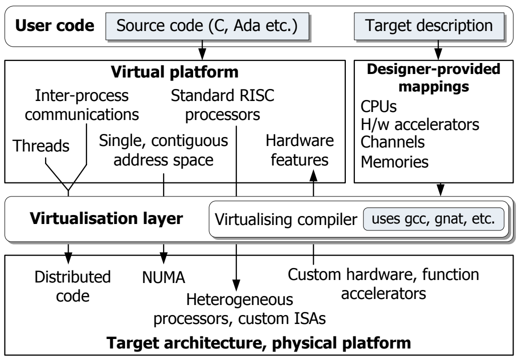
User code
Source code (C, Ada etc.)
Target description
Virtual platform
Inter-process communications
Standard RISC processors
Threads
Single, contiguous address space
Hardware features
Designer-provided mappings
CPUs
H/w accelerators
Channels
Memories
Virtualisation layer
Virtualising compiler
uses gcc, gnat, etc.
Distributed code
NUMA
Heterogeneous processors, custom ISAs
Custom hardware, function accelerators
Target architecture, physical platform

Figure 2: Tools-oriented view of CTV

2. COMPILE-TIME VIRTUALISATION

As described in section 1, the VMs present in standard software development hide important architectural information from the programmer, thereby preventing the efficient use of hardware features. *Compile-Time Virtualisation* (CTV) removes this problem by replacing the existing VMs with a single virtualisation layer across the entire architecture, termed the *Virtual Platform* (VP), which has three important features:

- 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. This allows the programmer's code to function irrespective of the target architecture.
- Unlike a standard run-time VM, the virtualisation mappings (thread → CPU, variable → memory space etc.) that are implemented by the VP are exposed to the programmer. This allows them to *influence* the implementation of the code and achieve a better mapping onto the architecture. For example, by placing communicating threads on CPUs that are physically close to each other, or locating global data in appropriate memory spaces to minimise copying.
- Also unlike a VM, custom hardware is 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, thereby allowing it to be effectively exploited.

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 processor or a new memory layout) a new VP is generated and the same input code is automatically retargeted to use this new architecture. This facilitates code reuse and allows designers to perform design space exploration rapidly.

The VL translates incoming source code, passes it though the language's existing compiler, and links the resultant object code. No new compilers need to be written, hence new platforms can be targeted easily. This is shown in

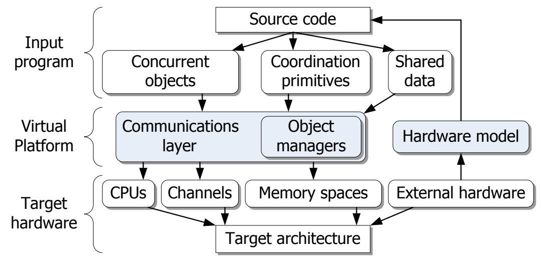

Figure 3: CTV system model

figure 2. This technique is language- and implementation-independent; nothing is assumed about the source language or the final implementation.

CTV does not duplicate the work of system design languages such as SystemC or SpecC. The design method presented by these languages requires the designer to begin at a high level of abstraction and iteratively refine the design until it is of a sufficiently low level to be implementable, and tends to target both hardware and software. CTV instead focusses on software and gives the programmer influence over implementation choices that are normally made by the compiler to obtain a better implementation on a wide range of architectures. Similarly, the CTV VP should not be confused with virtualisation environments like the CoWare Virtual Platform [8]. The CoWare VP is a software simulation of the target hardware that provides introspection and observability to aid software design. This technology is important to industry and could be integrated into CTV, but addressing it is an orthogonal problem.

Section 2.2 gives an overview of the abstract requirements of CTV, and discusses the system model as shown in figure 3. The components of the VP are then described in more detail in sections 2.3-2.5.

2.1 CTV compared to run-time virtualisation

CTV does not perform the same function as run-time virtualisation. Consider Java, an existing run-time virtualisation system. Java's VM extends the *run-time capabilities* of the system to make it appear to the programmer that the target CPU can execute Java bytecodes, when in fact it cannot. CTV, in contrast, does not exist at run-time and instead it extends the *compile-time capabilities* of the language and its compiler, to make it appear to the programmer that their language and toolchain can target more complex architectures than it normally can.

2.2 System model

The CTV system model is derived from the common characteristics of concurrent, imperative programming languages. These languages all have some notion of a concurrent object (threads or processes) and a set of coordination primitives that allow the concurrent objects to work together. For a language with a POSIX-based threading model these are mutexes and condition variables, for Ada [26] these are task entries and the rendezvous. Every concurrent language must also handle shared data, which may range from simple variables to complex elements like Ada's protected objects.

Given this, the CTV system model is shown in figure 3. In the diagram, arrows represent a programmer-controllable mapping from one element set to another. CTV models source code as a 3-tuple of sets (T, P, D) where T is a set of concurrent objects, P is a set of coordination primitives and D is a set of language-level shared data objects. (Compound types like structures and arrays are represented by only one set member.) Similarly, target architectures are represented by a 4-tuple, (C, L, M, H) where C is the set of CPUs in the target architecture, L represents communication channels (buses and on-chip networks), M is the set of memory spaces, and H is the set of external hardware elements (e.g. function accelerators, I/O channels) that are to be exposed to the source language. The VP is a mapping of source language-level elements to architectural elements that can be controlled by the programmer to best exploit the features of the target system:

$$(T, P, D) \rightarrow (C, L, M, H)$$

This model allows for many extra capabilities of the source language to be carried into the VP. For example, in the Real-Time Specification for Java thread affinity is modelled as a mapping $T \rightarrow C$ and memory spaces are modelled as $D \rightarrow M$.

Threads and shared data may be mapped to any target CPU and memory space respectively, so the VP must present a programming model which hides the implementation details of this complexity. Three main features are required.

- **Shared memory:** The implementation of a distributed shared memory system is required to allow transparent access to shared data from all threads regardless of its storage location, whilst still allowing control over where shared data is placed in the system. This creates the abstraction of a single logical address space over the entire architecture, which is assumed by almost all standard languages. For example, when two threads access a global variable CTV must account for the possibility that the two threads are mapped onto separate CPUs, neither of which share a memory space with the variable itself.
- **Concurrency:** Concurrency primitives that operate across the whole target architecture must be provided to support the execution model of the source language. Most concurrency features are provided by the operating system (i.e. POSIX mutexes) or the language run-time (i.e. Java synchronised methods) and so are localised to a single CPU and cannot operate over the entire system.
- **Communications:** The custom communications resources of the target architecture must be used transparently by all inter-thread communications without manual programmer intervention. For example, if a thread requests a mutex lock,it should communicate the request to the rest of the system appropriately.

To provide these three features, the VP introduces the concept of Object Managers, specifies a run-time communications layer, and specifies a model for the integration of external hardware elements. These are described in the following sections.

2.3 Object Managers

A key requirement of CTV is that it must be able to support future architectures with potentially thousands of interacting cores. Any form of centralised control must be avoided or it will result in a loss of scalability. Therefore, CTV uses a set of *Object Managers* (OMs) to manage operations on the members of the T, P and D sets (threads, coordination primitives, and shared data). Different OMs are entirely independent and operate in parallel, hardware permitting.

An OM is a passive object, mapped to a CPU, that is assigned to oversee all operations concerning a subset of the members of the T, P and D sets. For example, the OM of a mutex primitive stores the mutex's state and handles lock and release requests atomically. The OM of a shared data object handles read and write requests. Any number of objects can be assigned to an OM, however shared data objects must be stored in the same memory space as their OMs. Assigning two OMs to the same CPU is equivalent to assigning a single OM that manages both sets of objects.

$$s(O_3) = s(O_1) \cup s(O_2) \Rightarrow c = \{O_1, O_2\} \equiv c = \{O_3\}$$

Where c is a CPU from set C and $s(O_x)$ is the set of objects O_x is managing. Therefore the system model defines the set of OMs (O) as a subset of the CPUs in the system. Most OMs will share their CPU with other threads, so the performance of these threads will be reduced. The number of OMs, the items they manage, and the CPUs onto which they are allocated can be influenced by the programmer through the VP mappings in terms of the system model:

$$\wp^{T \cup P \cup D} \rightarrow O \text{ where } O \subseteq C$$

The set of OMs is static, which does slightly limit the run-time behaviour of the source code. However, for embedded systems this is not overly restrictive because dynamic behaviour is generally reduced in embedded development to increase code predictability and to ease proof of correctness.

The actual functionality of the OMs varies depending on the capabilities of the source language. Most concurrent languages (like Java and Ada) allow a thread to start other threads and then wait for them to finish, so for these languages the thread OMs must provide this functionality. Shared data OMs are the same for all languages and they allow a remote thread to read and write the contents of the object (or part of it).

If an OM is called by a thread that is hosted on the same CPU the request is handled internally, but if the thread or OM are remote then the communications layer (section 2.4) is used to send messages across the system.

2.4 The communications layer

The primary goal of the communications layer is to allow the VP to present a single logical address space to the source language. This requires the implementation of an object-based distributed shared memory system [25], such as Teamster [5] or Rthreads [11]. The specific advantages and disadvantages of different systems do not affect the overall approach. To implement this, the layer uses the functionality of the OMs on behalf of the source threads. Remote variable accesses are passed to the correct OM over a suitable communications channel (set L), whilst managing cache coherency and efficiency. The actual read or write is performed by the OM, but the operation as a whole is coordinated by the layer. Similarly, the layer manages the operation of coordination primitives. For example, when a thread requests a mutex lock the layer composes and transmits the request to the mutex's OM. Then, the OM performs the actual atomic test-and-set and uses the layer to send the reply.

The layer is applied at compile-time and leaves minimal run-time overhead. Once the mapping of OMs to the target architecture is known, the virtualisation layer can determine which threads need to communicate with which OMs. For example, if thread i creates thread j and reads variable s, i needs to communicate with the OMs of j and s. These communications are mapped onto set L and from this information the layer is built, linking in driver code to utilise the communication channels as required. Traditionally run-time issues like routing are moved to compile-time.

The source code is refactored to inject calls to the layer. Accesses to remote data objects are detected before compilation and calls inserted that implement the access. Like most distributed shared-memory systems, this assumes that remote accesses do not use arbitrary pointer manipulation. Pointers can be used, but only if they can be determined at compile-time to only point to the shared object.

The main penalty of moving the virtualisation from run-time to compile-time is that threads and OMs must remain static and cannot migrate throughout the system at run-time. Given the target domain, however, this is not an unreasonable restriction. Commonly code can not be migrated throughout a complex embedded architecture without re-compilation because of heterogeneous CPU instruction sets or differing memory layouts. Even when they can be, thread migration is complex and can make it difficult to analyse the worst-case execution time of the system to meet real-time deadlines. Finally, the frameworks to support such migration introduce undesirable run-time overhead.

Hardware type	**Port configuration**
Std. function accelerator	1 sync. port
Mailbox	1 sync. for sending, 1 async. for receiving
CAN Network	n async. ports
CSP channel	2 sync. ports

Figure 4: Examples of how external hardware is modelled by CTV

2.5 Hardware model

CTV defines a model for access to external hardware from the H set of the system model. Each element of custom hardware has at least one *port*, which can be used to access it from a CPU or co-processor. Each port is either synchronous or asynchronous and the ports of a single hardware element can be of different types. Synchronous ports provide blocking read and write semantics. The thread of control accessing the port must wait for the operation to complete before it continues with its execution. In contrast, asynchronous ports provide non-blocking semantics and allow the CPU to trigger an action which will complete at a later date. It is up to the implementation of the source language to determine a suitable notification method, although callback functions are often appropriate. Examples of how external hardware is represented in this model are shown in figure 4.

Each hardware port is assigned a set of parameters. All ports have an address, which is the base memory address that the attached CPU accesses it at. Asynchronous ports also have an associated interrupt vector and the name of a callback routine from user-level code that will be called when the interrupt fires. Note that synchronous ports may use interrupts internally for efficiency, but they will still present a synchronous interface to the programmer.

```
program ::= {decl | assign}
decl ::= id_list ":" type id [params]
id_list ::= id | id "," id_list
type ::= "processor"|"memory"|"hardware"|"channel"
params ::= "(" p_list ")"
p_list ::= param |  param "," p_list
param ::= number | string | list
assign ::= attrid "=" (param | id [params])
attrid ::= id "^" id
```

Figure 5: Simplified EBNF of the Anvil ADL

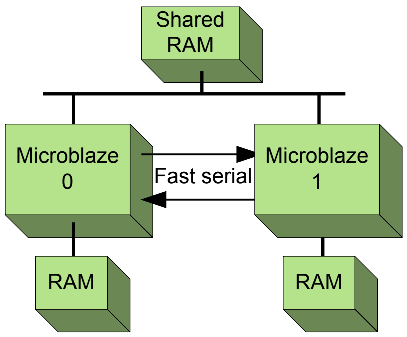

Figure 6: Simple dual-core system

3. ANVIL: AN IMPLEMENTATION OF CTV

Anvil is an implementation of CTV that maps applications written in C to complex multi-core architectures with non-uniform memory, custom on-chip interconnect, and non-standard hardware features such as function accelerators. Use of the POSIX standard pthreads library is assumed to provide multiple threads of control. Anvil uses an unmodified version of the `gcc` compiler internally for its object code generation. Input code is extensively refactored according to the mappings in its VP description (section 3.1) to present `gcc` with source code that will operate correctly on the target architecture. This refactoring is described in section 3.2.

3.1 Virtual platform description

Anvil uses a simple architecture description language (ADL) to define its VP and the mappings from source language to target hardware. This language does not have the power of a full ADL [7] because it only needs to provide a very high-level view of the architecture. Not all mappings need be defined (the system can perform many allocations automatically) but it is through this mechanism that the designer can influence the implementation of the system. The language syntax is described in figure 5.

The ADL allows the definition of target architectures in terms of the CTV 4-tuple model. The keywords `processor`, `memory`, `channel` and `hardware` declare sets C, L, M, and H respectively. Each declared item can be assigned *attributes* which add extra information about its behaviour and describe the mappings of the 3-tuple (concurrency objects, coordination primitives, shared data) down to the target platform. The simple dual-CPU system shown in figure 6 can be described as follows:

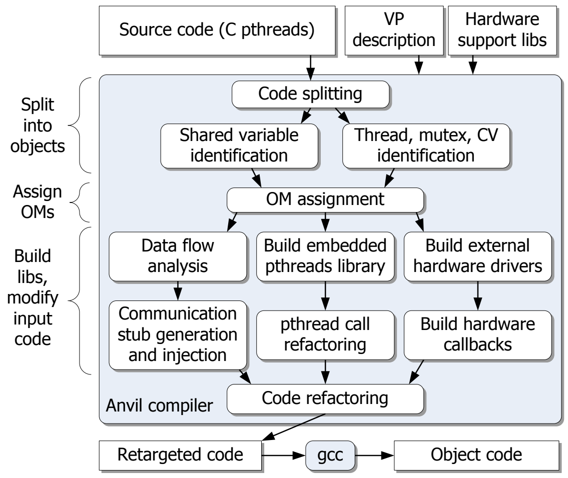

Figure 7: The main processes inside the Anvil compiler

```
CPU1, CPU2 : processor Microblaze;
mem1, mem2, mems : memory BlockRAM;
sc : channel UARTSerial;
CPU1^memory=mem1; CPU2^memory=mem2;
CPU1^extramemory = mems; CPU2^extramemory = mems;
sc^connectedto =
   [CPU1(0x84000000, 1), CPU2(0x84000000, 0)];
```

Note the use of the `connectedto` attribute to provide the serial channel's port parameters as required by the CTV hardware model (see section 2.5). Once the architecture has been described, the programmer maps system elements onto it. The following code segment uses attributes to map shared data into memory spaces (line 1), threads to execute on CPUs (line 2), and assign system objects to OMs (lines 3-4).

```
mems^variables = ["sourceimage"];
CPU1^threads = ["mythread", "main"];
CPU1^manages = ["sourceimage"];
CPU2^manages = ["mythread", "main"];
```

In this code, `sourceimage`, `mythread` and `main` are source language-level elements - items in the user's source code.

3.2 Refactoring

The Anvil refactoring process is shown in figure 7. 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. (The (T, P, D) input tuple.) Due to the fact that this is performed at compile-time, this information must be compile-time static. This is the main restriction on run-time behaviour that is imposed by CTV, although it is believed that given the target domain this is not an onerous restriction. Non-analysable behaviour is still supported, but it must be constrained to a single CPU. This is similar to many other systems, such as Rthreads [11].

The OM model is then applied. Items from the (T, P, D) tuple are assigned to OMs on the system's CPUs according to input from the programmer's ADL, as described in section 3.1. Unassigned system objects are automatically assigned, although only a simple allocation algorithm is used currently because optimal OM assignment is not the focus of this work. Existing work [19] has concentrated on the problems of evaluating task assignments in complex embedded systems and their results are useful here.

After OM assignment, the input program must be split into a set of programs, one for each CPU of the system. A new `main()` function is created for each of these CPU-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 CPUs 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.

Now Anvil has generated a program for each CPU, but they will not correctly execute due to a lack of shared memory and that the POSIX pthreads library doesn't support complex architectures. Three support libraries are required: an object manager-aware shared memory system, an embedded pthreads library, and driver code for custom hardware elements. These libraries are optimised for each CPU to use the memory, communications and hardware available, therefore one must be constructed for each core in the system.

As described in section 2.4, the shared memory system implements an object-based shared memory system. Where two threads share data that is not in shared memory, remote reads and writes must be conveyed over the system's communication channels. The implemented system uses migration semantics. When a remote variable is accessed it is semantically migrated over the to the reader who can manipulate it freely before writing its final value and migrating it back to its manager. This fits well with the mutex-based coordination of pthreads-based programs. The refactoring engine uses data-flow analysis over the abstract syntax tree to determine the points in the program where shared variables (that are not in shared memory) are used as L- or R-values and calls that remotely read and write the data are injected before and after these accesses. Consider the following code:

```
void main(void) {
   printf("%d\n", x);
   x = x + 1;
}
```

Here, `x` is a shared variable which is read and written by the code in `main`. 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, 1); //id, offset, len, bytes
10     printf("%d\n", x);
11     x = x + 1;
12     _anvil_write_sv(0, 0, 0, 1);
13  }
```

Lines 1 and 2 declare data structures internal to the shared memory library that are responsible for tracking and managing shared data. Line 7 assigns an address that can be used as local storage for when the remote variable is migrated over, and finally lines 9 and 12 are the injected read and write calls. To extend this, shared data can be associated with a mutex so that all writes are cached and the final value not updated until the associated mutex is released. Pointer access to shared data is also supported through the use of offsets if the pointer is initialised to point to the shared data. Anvil has enough compile-time information to exploit burst-mode and DMA transfers to speed data transmission.

Cache coherency must be considered when two CPUs are accessing shared data from shared memory. However, this is a well-studied problem and the results from existing work [9, 6] can be directly implemented in CTV. As coherency is not the focus of this work, Anvil currently uses a simple algorithm. Upon a write to a shared variable, that variable's OM instructs the other threads that access this data to flush their cache next time they attempt a read. More complex algorithms would increase efficiency but are not yet implemented. Some architectures support cache coherency natively (e.g. the ARM Cortex) so when targeting these architectures Anvil does not need to consider coherency and assumes it will be provided automatically.

The embedded pthreads library - the Anvil implementation of the CTV communications layer - contains stub functions for all architecture-specific functionality (such as sending a message to another CPU or handling interrupts). These are completed at compile-time with routines read in from an external support library to create a CPU-specific library that can support all of pthread's thread, mutex and condition variable operations. It uses the OM assignments to determine the CPU that each request should be sent to. This is also done at compile-time, no run-time overhead is required.

Finally, driver code for custom hardware is retrieved from Anvil's external libraries and added to the output code. All Anvil drivers use a C struct to store the port parameters that are required by the CTV hardware model (section 2.5). This struct is injected by the refactoring process from information provided in the ADL so that the programmer does not need to worry about hardware addresses or interrupt vectors. An example of this is shown later in section 4.5.

Once the support libraries have been built, the final stage involves refactoring the original pthread calls to point to the new embedded pthreads library and the code will compile under standard `gcc` but execute on the target architecture.

Due to the fact that the refactoring stage adds another layer of transformation to the programmer's source code, CTV's impact on traceability must be considered. However, because the refactoring is entirely predictable, debugging information can be carried through in the same way that it is through standard compilation optimisations and into ancillary libraries.

4. TARGETING ARCHITECTURES WITH ANVIL

This section details some of the ways that Anvil uses CTV to map software onto complex embedded architectures. All systems are implemented on a Xilinx Virtex 4 FPGA (the XC4VLX25-FF668-10) on the Xilinx ML401 prototyping board and use the Microblaze soft-processor. Other FPGAs and cores are supported however.

Benchmark	With CTV	Without CTV
quicksort-fpu	26,222	26,222
fast-dct	7,716	7,716
binarysearch	114	114

Figure 8: Evaluation time (clock cycles)

4.1 Overhead introduced by CTV

The first architecture presented is a single-CPU system with a single block of memory - the standard Von Neumann-style architecture assumed by C. This is considered because it helps to illustrate the main advantage of CTV, that the only run-time overheads introduced are those actually *required* by the system (such as copying shared data, inter-thread communications etc.). When the VP and the actual platform are the same, the virtualisation reduces to nothing.

Figure 8 shows the execution times of three benchmark programs when run with and without the aid of CTV. It shows that because CTV does not need to alter the code to help it execute correctly on the target architecture the resulting overhead is 0%. This is different to run-time systems such as Java or CORBA [22] which always introduce overheads.

It can be determined that there are only two possible sources of (non-required) overhead in the system. First, Anvil is forced to use a general-purpose communications protocol for implementing its architecture support libraries. With application-specific knowledge it is possible to trim the required number of messages and therefore reduce time spent communicating. However this is also architecture-specific and error-prone, and can only offer a small benefit for most applications.

The second source of potential inefficiency concerns the shared memory and cache coherency systems implemented by Anvil. These systems are applied at compile-time, and it is not always possible for the compiler to determine exactly which elements of shared data are read in a given code block. As a result, the compiler may have to transfer more data than is actually required at run-time. Similar problems are faced by all shared memory systems and as better techniques are discovered they can be easily implemented in Anvil to reduce this. One interesting advantage that CTV-based systems have in solving this problem is that because CTV has access to the source code (which run-time systems do not) more expressive source languages will help the compiler to perform better.

4.2 Shared memory architectures

The first non-standard architecture presented is a dual-processor system containing dedicated memory for each CPU and a third block of shared memory accessible to both. The CPUs communicate using an Anvil-supported serial communications channel. This architecture is shown in figure 6. Three applications are presented running on this system, a Sobel edge filter, triple-DES encryption and a forward discrete cosine transform (FDCT). All of the programs access shared data and have not been rewritten for use in Anvil. In these tests, Anvil successfully mapped the software threads across the two CPUs and handled communications and cache coherency correctly. Figure 9 lists the execution time (in clock cycles) for these three applications, demonstrating the expected increase in throughput as the cache is introduced.

Program	Cached	No cache
Sobel	13,078,036	28,430,514
3-DES	16,234,470	19,364,574
FDCT	31,643	36,688

Figure 9: Evaluation time (clock cycles)

Program	Execution time (cycles)
Single-threaded	34,363,789
Multi-threaded (hand-coded)	19,217,404
Multi-threaded (Anvil)	19,364,574

Figure 10: 3-DES on a 2-core NUMA system

To examine the amount of unnecessary overhead being introduced by Anvil, a hand-written version of the triple-DES program was created and compared with the 3-DES results from the previous experiment. This is shown in figure 10. As can be seen, the Anvil version demonstrates only 1% overhead. The hand-coded version employed an application-specific communications protocol, an option which is not available to any automatic tool. This is the first source of overhead identified previously in section 4.1. This indicates that the performance of Anvil can be very close to that of hand-written code.

4.3 Distinct memory architectures

Systems with distinct memory spaces require the OMs to actively pass data between threads. This is clearly slower, but is often required as the number of cores and memory spaces in an architecture increases. To demonstrate Anvil's support for this, the shared memory was removed from the above architecture and the Sobel application reimplemented. The application still completed correctly but it is heavily bottlenecked by the slow serial connection between the two CPUs. The application has to transfer the entire 200×200 pixel source image from CPU1 to CPU2, and then copy the filtered image back, which took over 9 million cycles and increased the run-time of the application by 32%, or 69% when compared with the cached version.

If present on the architecture it is possible to use DMA engines to automate this copying and reduce wasted cycles. This is not yet implemented in Anvil but it is supported by the CTV object manager model.

4.4 Many-core systems

To show the scalability of CTV and the OM model, a five-core system inspired by IBM's Cell processor was also developed, shown in figure 11. In this architecture, cores 1-4

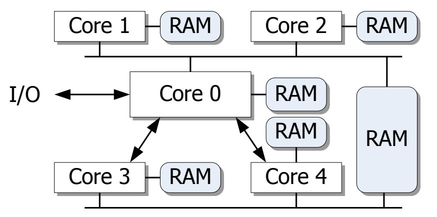

Figure 11: 5-core NUMA system with custom interconnect

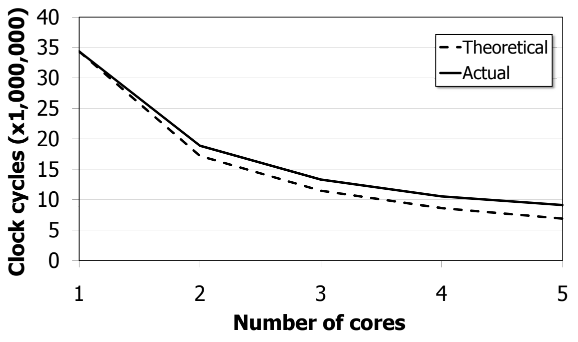

Figure 12: Evaluation times for 3-DES compared against the theoretical best-case

are small Microblaze cores with a 3-stage pipeline whilst core 0 has an extended instruction set due to it having a floating point unit, 5-stage pipeline, and instruction and data caches. The architecture contains a complex memory layout and multiple communication channels. It consumes 91% of the target FPGA's logic resources and has a throughput of 2160 Dhrystone MIPS at 360MHz. Porting the application to this architecture simply requires preparing an Anvil ADL file to describe the VP and running the refactoring process. The input code remains the same. When a thread is mapped onto core 0, because this is known at compile-time the compiler is able to create a binary which uses its extra instructions. Figure 12 shows the execution times for varying numbers of active cores when compared against the theoretical best-case speed up (2x for 2 cores, 3x for 3 cores etc.). As expected, the graph shows that the actual implementation is slower as the advantages of parallelisation are reduced by task interference. However, this example shows the communications layer operating correctly over a complex architecture and the interthread communications scaling to the highest number of cores that can be fit onto the target FPGA. Whilst this result is encouraging, further experiments will demonstrate architectures with even greater numbers of cores.

4.5 Function accelerators - Synchronous

This example demonstrates how Anvil can use function accelerators from the implementation architecture. The target system is a single-core system containing hardware to evaluate the quadratic polynomial $y = ax^2 + bx + c$ in floating point arithmetic. The coefficients a, b, and c are presented to the hardware and on the next clock cycle y is available. The Microblaze core used in this system does not have a floating point unit, so all its floating point operations have to be emulated in software.

The accelerator is first cast into the CTV hardware model. It is accessed through a single synchronous port and therefore the only parameters it requires is the memory address that it is located at (0x86000000 in this case). It is described in Anvil ADL as:

```
quadeval : hardware Quadratic(CPU1, 0x86000000);
```

The Anvil hardware libraries define the interface to this accelerator through driver functions:

```
typedef struct {int *addr;} quad_t;
```

Software	With h/w acceleration
3808 - 3910 cycles	48 cycles

Figure 13: Quadratic evaluation times

```
int evaluate_quadratic
      (quad_t quad, char a, char b, char c){
   *(quad.addr) = (a << 16)|(b << 8)|c;
   return (*(quad.addr + 1));
}
```

`quad_t` is a structure type which holds the port's parameters, the declaration of which is inserted into the source code during refactoring.

```
quad_t quadeval = {(int *)0x86000000};
```

The user can write code to use the `evaluate_quadratic` function normally and the refactoring will ensure that it is always compiled to access the correct memory locations. Through the VP, custom hardware is accessed using *language-level* constructs in a way that fits C's programming model. The problematic and error-prone integration is handled at compile-time by Anvil without requiring the user to deal with assembly code, custom linking or any techniques external to the source language.

Figure 13 shows the number of clock cycles taken to evaluate a single equation using software emulation and the hardware accelerator. A range of input variables for the coefficients were tested and the results show that the hardware version is considerably faster, as expected.

4.6 Function accelerators - Asynchronous

Asynchronous accelerators are integrated in much the same way as synchronous ones. This section describes the integration of a timer that can be requested to interrupt the CPU a given number of clock cycles in the future. This hardware is described in the CTV hardware model as having a single asynchronous port, so it therefore requires an address, an interrupt vector, and the name of a function to trigger when it interrupts. In this example the following ADL is used:

```
mycall : hardware Callback(CPU1, 0x84000000, 3, trigger);
```

Again the libraries define the hardware interface, in this case a trivial function which is used to request a callback from the hardware, however they must also integrate the interrupt handling. This is done during refactoring when the communications layer is built. The main interrupt handler of the layer is extended so that every time an interrupt arrives on vector 3 it calls a function named `trigger` which is a void function that takes no arguments and is provided by the user to perform whatever actions they require. As with the synchronous example above, the integration process is now entirely expressed in terms of the source language's programming model. Both examples scale to include an arbitrary number of accelerators and the technique is only limited by the addressing capabilities of the target CPU.

Using the techniques from this section and section 4.5, external communication channels such as UARTs and networks have been brought into the Anvil model. The model also works for I/O devices like buttons and LEDs. Anvil's hardware library currently provides support for a wide range of such devices.

5. CONCLUSION

This paper has shown how Compile-Time Virtualisation (CTV) can be used to extend the programming model of a high-level language to help the programmer 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's Object Manager system 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 presented which is shown to be able to effectively target multi-threaded C code to complex multi-core architectures with custom memory layouts and non-standard hardware elements such as function accelerators. Anvil is shown to introduce only a very small run-time overhead when compared with hand-optimised code.

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.

6. REFERENCES

[1] D. Andrews, D. Niehaus, and P. Ashenden. Programming models for hybrid cpu/fpga chips. *Computer*, 37(1):118–120, 2004.

[2] W. Cesario et al. Component-based design approach for multicore SoCs. *DAC*, 00:789, 2002.

[3] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[4] R. Chandra et al. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[5] J.-B. Chang, C.-K. Shieh, and T.-Y. Liang. A transparent distributed shared memory for clustered symmetric multiprocessors. *The Journal of Supercomputing*, 37(2):145–160, 2006.

[6] L. Cheng, J. Carter, and D. Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 328–339, Feb. 2007.

[7] P. C. Clements. A survey of architecture description languages. In *IWSSD '96*. IEEE Computer Society, 1996.

[8] CoWare, Inc. CoWare Virtual Platform - hardware/software integration and testing...without hardware. http://www.coware.com/products/virtualplatform.php (Accessed Aug 09).

[9] A. Cox and R. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Computer Architecture, 1993., Proceedings of the 20th Annual International Symposium on*, pages 98–108, May 1993.

[10] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. *DAC*, 2001.

[11] B. Dreier, M. Zahn, and T. Ungerer. The Rthreads distributed shared memory system. In *3rd Int. Conf. on Massively Parallel Computing Systems*, 1998.

[12] K. Fatahalian et al. Sequoia: programming the memory hierarchy. In *SC '06*, page 83, 2006.

[13] S. Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[14] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[15] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[16] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *FCCM '00*, 2000.

[17] J. Gunn, K. Barron, and W. Ruczczyk. A low-power DSP core-based software radio architecture. *IEEE, Selected Areas in Communications*, 17(4):574–590, Apr 1999.

[18] Institute of Electrical and Electronics Engineers. SystemC language reference manual (IEEE std 1666-2005). 2005.

[19] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout. A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 876–881, Washington, DC, USA, 2005. IEEE Computer Society.

[20] A. Munshi, editor. *The OpenCL Specification*. Khronos OpenCL Working Group, 2008.

[21] NVIDIA Corporation. *CUDA Programming Guide ver 1.1*. 2007.

[22] A. L. Pope. *The CORBA reference guide: understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[23] F. Rivard. Smalltalk: A reflective language. In *International Conference on Metalevel Architectures and Reflection*, 1996.

[24] J. Robelly, G. Cichon, H. Seidel, and G. Fettweis. A HW/SW design methodology for embedded SIMD vector signal processors, 2005.

[25] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *Computer*, 23(5):54–64, May 1990.

[26] S. T. Taft and R. A. Duff. *Ada 95 Reference Manual: Language and Standard Libraries (ISO/IEC 8652:1995(E))*. LNCS 1246, Springer Verlag, 1997.

[27] W. Thies et al. StreamIt: A compiler for streaming applications, December 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.