Targeting Complex Embedded Architectures by Combining the Multicore Communications API (MCAPI) with Compile-Time Virtualisation

Ian Gray
Department of Computer Science
ian.gray@cs.york.ac.uk

Neil C. Audsley
Department of Computer Science
neil.audsley@cs.york.ac.uk

Abstract

Within the domain of embedded systems, hardware architectures are commonly characterised by application-specific heterogeneity. Systems may contain multiple dissimilar processing elements, non-standard memory architectures, and custom hardware elements. The programming of such systems is a considerable challenge, not only because of the need to exploit large degrees of parallelism but also because hardware architectures change from system to system. To solve this problem, this paper proposes the novel combination of a new industry standard for communication across multicore architectures (MCAPI), with a minimal-overhead technique for targeting complex architectures with standard programming languages (Compile-Time Virtualisation).

The Multicore Association have proposed MCAPI as an industry standard for on-chip communications. MCAPI abstracts the on-chip physical communication to provide the application with logical point-to-point unidirectional channels between nodes (software thread, hardware core, etc.). Compile-Time Virtualisation is used to provide an extremely lightweight implementation of MCAPI, that supports a much wider range of architectures than its specification normally considers. Overall, this unique combination enhances programmability by abstracting on-chip communication whilst also exposing critical parts of the target architecture to the programming language.

Categories and Subject Descriptors C.0 [*General*]: Modeling of computer architecture; C.2.4 [*General*]: Computer-Communications Networks—Distributed systems; 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

When programming complex embedded heterogenous multicore platforms, there is an inevitable trade-off between programmability and efficiency. High-level programming languages typically used in embedded systems development (e.g. C) assume the presence of a common address space to facilitate the use of shared data. They provide few mechanisms or abstractions for direct programming of the target hardware for efficient device programming. Whilst these assumptions can be supported efficiently on SMP-style architectures with cache-coherence algorithms and similar architectural techniques, it is a significant challenge to provide support on heterogeneous platforms with asymmetric memory architectures (containing many CPUs, function accelerators, memories and communications channels).

In this paper we propose an approach to support the common assumption of a single logical address space on complex, heterogeneous architectures through the use of compile-time refactoring of the source code. The technique can be applied to legacy code written with no assumptions regarding target platform. However, this can lead to inefficient systems, particularly if the characteristics of the underlying non-uniform memory are not considered by the source code. To support direct exploitation of the architecture, we extend the refactoring approach to expose pertinent platform features to the source code with a simple architectural description language. This enables the programmer to improve efficiency without sacrificing the programmability of common languages. For example, facilities are provided to constrain computation (eg. threads) to particular processors, enabling them to be placed close to the shared data which they access. The data itself can be constrained to a particular area of memory.

The compile-time refactoring presents to the source code a Virtual Platform, which is an idealised view of the underlying hardware that supports the single address space abstraction and similar architectural simplifications whilst hiding low-level implementation details. The Virtual Platform supports the programming model of the source code, and can be manipulated to allow architectural exploration and mapping.

A key facet of the approach proposed in this paper is to incorporate proposed standard abstractions of on-chip, inter-core communication. The Multicore Association has developed the Multi-Core API (MCAPI) standard [25] which abstracts physical on-chip communication into virtual channels between nodes in the system. Nodes can be software entities (eg. OS, application thread) or hardware entities (eg. processor, hardware accelerator). The result is a uniform lightweight communications interface between constituent parts of the system.

As an API, MCAPI is bound by the programming models of existing languages. Whilst a programmer is able (via MCAPI) to achieve abstract communications with constituent nodes in the system, other parts of the architecture are hidden by the programming language. For example, it is becoming increasingly common for the architectures of embedded systems to include large amounts of application-specific hardware, such as function accelerators, non-

LCTES'11, April 11–14, 2011, Chicago, Illinois, USA.
Copyright © 2011 ACM 978-1-4503-0555-6/11/04. . . $10.00

standard memory layouts, heterogenous CPUs, or custom interconnect. Standard programming languages do not use these features natively as their abstraction models hide key architectural details. This makes it impossible for an MCAPI implementation to use these features, and relies on the presence of a distributed OS or middleware layer which is not always practical in a resource-constrained embedded system. This problem is solved by the approach in this paper.

The remainder of the paper analyses the problems encountered when targeting complex architectures in section 3, and describes the approach taken by this work in section 4. The approach involves two technologies, MCAPI (described in section 4.1) and Compile-Time Virtualisation (described in section 6). Section 7 describes the implementation detailed in this paper followed by an evaluation in section 8.

2. Contributions

This paper combines MCAPI, an API for multicore communications with Compile-Time Virtualisation (CTV), a technique that allows normal architecturally-neutral code to be mapped to complex architectures. The resulting system demonstrates the following contributions:

- Support for multicore programming models on complex architectures that MCAPI would not normally be able to support due to its reliance on its host language (for example, non-uniform memory architectures with non-standard interconnect). MCAPI can be used on more targets.
- Low-level and error-prone hardware-specific code is automatically generated.
- Integration of custom hardware items (function accelerators etc.) into the programming model.
- Decoupling of the target architecture with CTV's Virtual Platform greatly aids the portability and reuse of MCAPI code.

3. Problem Analysis

The abstraction models of existing programming languages were not developed to cope with the variety and variability of modern embedded systems. Early computer architectures were largely uniform and entirely static, consisting of a single processor with access to one contiguous block of memory. As a result, many architectural details were hidden by the abstraction layers of programming languages. This approach has been inherited by modern languages, which increasingly rely on the presence of middleware or a distributed operating system to provide program distribution and architectural mapping. Access to features such as complex memory or custom hardware can only be achieved though the use of abstraction-breaking techniques (link scripts, inline assembly, raw pointers etc.). These techniques are error-prone, difficult to port to new architectures, and hard to maintain.

In a resource-limited embedded system, the absence of a full OS causes the architectural assumptions of the language to fail, resulting in the following problems:

Non-uniform memory and caching:

Languages cannot assume that all variables in scope are accessible to all threads of the system. If a thread is located on a processor that is not connected to some of the memory spaces of the system then this is not the case, and some data items may be inaccessible.

Universal communications:

Programming models also cannot assume that threads can communicate with all other threads. In an SMP-style architecture this assumption is reasonable, but as implementation architectures grow the presence of a single shared communications bus is less likely. Communications must be routed using complex routes that may involve a number of hops. Routing must be performed manually by the programmer, or with a middleware layer such as CORBA [22]. Existing middleware solutions target networked systems and are too large for resource-constrained embedded systems.

Universal coherence:

When caches are present in the implementation architecture coherency must be considered. Language models tend to assume that items of shared data are kept coherent across the entire system, regardless of where each item is actually used. Cache coherency does not scale to large numbers of processing cores [2], yet the models do not allow the programmer to limit coherency to useful subsets of the system.

Distributed services:

Language models provide a range of services or libraries to express threading, migration, coordination primitives etc. However, these services are often not implemented over complex target architectures, restricting their use to individual cores of the system. For example, pthreads based mutexes will not operate correctly between the cores of an embedded architecture without the use of a full distributed operating system to act as a middleware layer. When language runtime systems are distributed (for example the Distributed Ada Runtime System [16]) they tend to target large-scale homogeneous networked systems rather than heterogeneous embedded systems.

The architectural features listed above will cause the programmer's code to fail because the programming model of the input code is not supported by the underlying architecture. There are also a range of issues which do not cause implementations to fail, but can lead to very inefficient systems:

- **Flexible mappings:** The transparent placement of code and data throughout the target architecture is essential in a heterogeneous system. Threads need to be mapped to the processor which can most efficiently execute them, and data should be stored close to the threads which use it.
- **Communications abstractions:** As systems move further from SMP-style architectures with a single shared bus, explicit communication abstractions are required to make efficient use of the target system.
- **Exploitation of unique hardware elements:** Unique hardware features (such as function accelerators, I/O devices, DSP cores etc.) must be accessible to the programmer.

Embedded languages such as C, Ada or Real-Time Java do not provide the programmer with the expressive power to reason about these issues from within the source language. C in particular is heavily dependent on its assumed target architecture of a standard Von Neumann architecture with a single processor and single logical memory space.

Other languages do allow some architectural mapping detail, but none provide enough to efficiently map to modern embedded architectures unassisted. For example, Java thread affinities [9] allow the programmer to reason about the location of computation in the target system, but its memory model is similar to that of C and cannot easily map to NUMA systems. The Real-Time Specification for Java [12] is currently attempting to extend this model to consider more complex memory hierarchies. Ada's Distributed Systems Annex (DSA) [3] is designed for distribution between workstations and as a result its partitions are too coarse for use within a small embedded system. (Partitions cannot directly share memory or peripherals.)

There has also been a recent push to develop new languages that contain greater amounts of architectural detail. For example to target non-uniform memory [10] or data streaming architectures [11, 26]. Unfortunately, such languages tend to only concentrate on solving a single problem each, and cannot adapt easily to support new architectural paradigms in the future.

Partitioned Global Address Space (PGAS) languages are another recent development. Rather than assuming a single logical memory space, a PGAS programming model targets a partitioned memory space. Intra-partition communication is assumed to be cheaper and faster than inter-partition communications, and threads can be placed inside partitions to ensure they retain locality with their working data set. Common PGAS languages are X10 [7], UPC [4] and Chapel [5].

The development of PGAS languages demonstrates that a move towards a greater amount of architectural information is important for the efficient exploitation of complex architectures. However, PGAS languages focus on high-performance computing environments, and do not support the kinds of application-specific memory hierarchies or custom hardware elements commonly seen in embedded systems.

The approach taken in this paper shares similarities with run-time virtualisation systems (such as Java) and the architectural information of PGAS languages, but it focussed on supporting highly variable architectures.

4. Approach

This paper introduces a system which solves the problems identified in section 3. The approach has the following characteristics:

- Rather than attempt to define another new language, standard C is used because of the large amount of programmer experience, industrial-level tools, and legacy code that already exists for it.
- In order to support efficient use of complex communication hierarchies, a proposed standard for on-chip communications from the Multicore Association (called MCAPI [25]) is used. MCAPI adds explicit inter-thread communications to the programming model.
- Mapping to complex architectures is achieved through the use of Compile-Time Virtualisation (CTV) [13], a technique for programming embedded architectures that imposes minimal run-time overheads.
- CTV provides a flexible Virtual Platform that hides the underlying complexities of the target hardware whilst affording the programmer control over high-level mapping decisions to maintain efficiency.

MCAPI is described first in section 4.1. CTV, and the way in which it can be combined with MCAPI is then described in section 6.

4.1 MCAPI

The Multicore Association (MCA) was founded in 2005 as an industry forum for establishing a set of application programming interfaces (APIs) to be supported by industry on multicore technologies. The MCA has a number of key industrial members and is widely seen as an important contributor to multicore programming practice.

The APIs developed are in two main areas: communications and resource management. Communications, via the MCAPI API, is intended to support communication within a closed distributed system – e.g. a multicore system-on-chip. Resource management, via the MRAPI API, is intended to support and coordinate shared memory management and synchronisation (eg. mutexes, semaphores). The MCAPI API has now been released as a de-facto industry standard whilst MRAPI remains under discussion. MCAPI, MRAPI, and the efforts in virtualisation and application programming are shown in figure 1. It is important to note that the two APIs are independent of each other, and can be implemented separately.

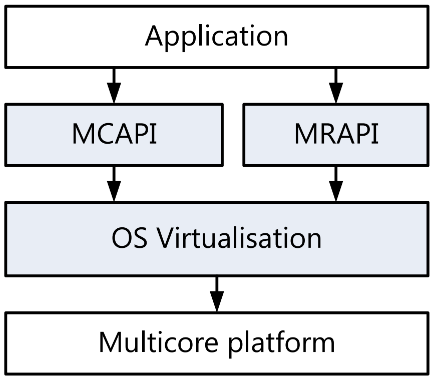

Figure 1. Multicore Association APIs Overview

MCAPI [25] defines a communication API for closely coupled systems where there are multiple cores connected in an arbitrary topology with no requirement for symmetry. MCAPI enables point-to-point communication between *endpoints* that are associated with *nodes*. An MCAPI node is a logical notion that can be a process, a thread, an OS instance, or hardware component (accelerator, CPU, DSP core etc.). A node can contain several MCAPI endpoints and endpoint identifiers are unique within the system, named with a tuple `<node_id, port_id>`. Endpoints can have a set of attributes describing QoS features, buffer capabilities and timeouts.

After MCAPI initialisation, a connection is created and established between two endpoints. Then both sender and receiver can open a channel between those endpoints. Data to be sent is stored in an application buffer and passed to an MCAPI send endpoint. When there is space at the receiver, the data is sent to the receiving endpoint where it is stored in a FIFO buffer for subsequent application use.

An example use of MCAPI to communicate between two nodes is given in figures 2 (`node_1`) and 3 (`node_2`). In this example two channels are used: a packet channel `channel_1` is used to communicate data from `node_1` and to `node_2`; and a scalar channel `channel_2` is used for the reply.

Related Work

Approaches such as OpenMP [6] and MPI [15] are aimed at widely distributed systems and are intended to address issues in parallel programming rather than fundamental communications abstractions. Existing communication frameworks such as CORBA [22], TIPC [20] and MPI tend to be unsuitable for resource-constrained embedded systems because are they are all either designed for communications between larger entities (workstations, or clusters) or result in large libraries that consume a lot of system memory. MCAPI is designed to place as small a set of requirements on communicating nodes as possible, leading to a very small implementation footprint.

Approaches such as TBB [23] and OpenCl [21] have been proposed as programming approaches for closely coupled systems – essentially symmetric multiprocessor architectures typified by desktop multicore processors and graphics processing units (GPUs) respectively. However, these approaches do not provide coverage of general heterogeneous architectures.

```
//MCAPI Communications Example - node_1.c
char *buffer[1024];
mcapi_uint8_t reply;
size_t size;
mcapi_node_t node_id_local = NODE_LOCAL;
mcapi_node_t node_id_remote = NODE_REMOTE;
mcapi_port_t pid_ch1_loc = PORT_CH1_LOCAL;
mcapi_port_t pid_ch1_rem = PORT_CH1_REMOTE;
mcapi_port_t pid_ch2_loc = PORT_CH2_LOCAL;
mcapi_endpoint_t e1_loc, e1_rem, e2_loc;
mcapi_pktchan_send_hndl_t ch1, ch2;
mcapi_request_t request;
mcapi_status_t mcapi_status;

//Create local endpoints on channels 1 and 2
e1_loc = mcapi_create_endpoint(
   pid_ch1_loc, &status);
e2_loc = mcapi_create_endpoint{
   pid_ch2_loc, &status);

//Get and connect the remote endpoint on channel 1
e1_rem = mcapi_get_endpoint(
   node_id_remote, pid_ch1_rem, &status);
mcapi_connect_pktchan_i(
   e1_loc, e1_rem, &request, &status);

//Open a Packet channel and a Scalar channel
mcapi_open_pktchan_send_i(
   &ch1, e1_loc, &request, &status);
mcapi_open_sclchan_recv_i(
   &ch2, e2_loc, &request, &status);

//Blocking send on channel 1
mcapi_pktchan_send(
   ch1, buffer, 16, &status);

//Blocking receive on channel 2
reply = mcapi_sclchan_recv_uint8(
   ch2, &status);

//Non-blocking send on channel 1
mcapi_pktchan_send(
   ch1, buffer, 1024, &request, &status);

//Continue with application whilst send occurs...

//Wait on channel 1 send complete
mcapi_wait(
   &request, &size, &status, MCAPI_INFINITE);
```

Figure 2. MCAPI send/recive example: `node_1`

Implementations of MCAPI are currently limited to an outline reference implementation from the MCA. Little other work has considered MCAPI, only a model checking approach to check the use of MCAPI API calls from an application [24].

4.2 MRAPI Overview

The MCAPI API does not consider the provision of shared resources. The Multicore Resource Management API (MRAPI) [17] is a companion API that has been developed to address this issue by defining a set of lightweight primitives that are designed to be used by MCAPI programs. MRAPI is still in development, but the primitives currently supported are:

```
//MCAPI Communications Example - node_2.c
//Declarations identical to node_1.c

//Create local endpoints on channels 1 and 2
e1_loc = mcapi_create_endpoint(
   pid_ch1_loc, &status);
e2_loc = mcapi_create_endpoint(
   pid_ch2_loc, &status);

//Open a Packet channel and Scalar channel
mcapi_open_pktchan_recv_i(
   &ch1, e1_loc, &request, &status);
mcapi_open_sclchan_send_i(
   &ch2, e2_loc, &request, &status);

//Get and connect the remote endpoint on channel 2
e2_rem = mcapi_get_endpoint(
   node_id_remote, pid_ch2_rem, &status);
mcapi_connect_pktchan_i(
   e2_loc, e2_rem, &request, &status);

//Blocking receive on channel 1
mcapi_pktchan_recv(
   ch1, &buffer, &size, &status);

//Blocking send on channel 2
mcapi_sclchan_send_uint8(
   ch2, data, &status);

//Non-blocking receive on channel 1
mcapi_pktchan_recv_i(
   ch1, &buffer, &request, &status);

//Continue with application whilst send occurs...

//Now wait on channel 1 receive complete
mcapi_wait(
   &request, &size, &status, MCAPI_INFINITE);
```

Figure 3. MCAPI send/recive example: `node_2`

- Mutexes
- Semaphores
- Shared and remote memory
- Metadata

Whilst MRAPI is independent from MCAPI (as shown in figure 1), it is intended that MRAPI implementations utilise the communications primitives provided by MCAPI. This does not prevent MRAPI implementations using a different communications infrastructure.

5. Using MCAPI for complex architectures

Due to the fact that MCAPI is an API, it is restricted to use the same programming model of its host language. Its implementation can be written to support complex interconnect and memory topologies, but this is not sufficient if the base language does not support these features. For example, MCAPI's C implementation (the only implementation currently available) is limited by the fact that standard C does not allow the programmer to specify multiple threads of control. Therefore, it is frequently used alongside a threading library, such as pthreads [18]. However, pthreads requires the use of an OS kernel, which can create a bottleneck in the system, requires a SMP-style memory layout, and does not map to application-specific architectures.

Consequentially, programmers of complex embedded systems are forced to manually split their single program into a set of cooperating programs, frequently one for each target CPU. These split programs must then be manually mapped to the target architecture with the use of driver code, low-level assembly, custom linker scripts, and other techniques that are outside of the source language's programming model.

This multi-program approach allows fine-grained control over architecture mapping, thereby reducing inefficiency and allowing non-standard hardware features to be exploited (function accelerators, DMA engines, scratchpad memories etc.). However it introduces a number of problems:

- The code is entirely architecture-specific and cannot be easily ported to another system. Equally, off-the-shelf components from other systems must be rewritten before they can be used.
- Such low-level code is error-prone and requires extensive hardware knowledge to produce.
- DSE is poor, as architectural changes can require large amounts of the code to be changed. This leads to a development model in which the hardware is designed before software is produced, and then once software development has started hardware development ceases. Iterative hardware development is not possible.
- A single-program model is more natural and fits better with the way in which programmers describe and prototype their systems.

The next section describes CTV, which allows the programmer to map their single program without having to split it into subprograms.

6. Compile-Time Virtualisation

The issues identified in section 3 and 5 mean that when developing software for complex systems, MCAPI alone is not sufficient. This paper combines MCAPI with a technique called Compile-Time Virtualisation to provide better architectural mapping and therefore allow MCAPI applications to efficiently exploit non-standard application-specific architectures. Compile-Time Virtualisation is described in the rest of this section, and the way in which it is integrated with MCAPI is detailed in sections 6.1 and 7.

Compile-Time Virtualisation (CTV) [13] is a virtualisation-based technique that attempts to give the programmer a more suitable abstraction model for developing software for complex, application-specific architectures. CTV replaces the existing layers of virtualisation and abstraction that are present in standard software development with a single virtualisation layer across the entire architecture, termed the *Virtual Platform* (VP). The VP has 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, for languages such as C it presents a single logical address space, hiding data migration, updating and caching issues. The layer ensures that the programmer's code operates correctly on a complex architecture without low-level programmer intervention. The VP allows for code to be architecturally-neutral.

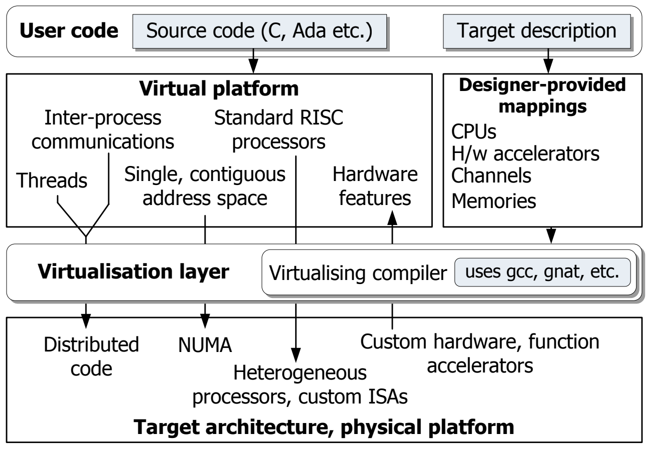
User code
Source code (C, Ada etc.)
Target description
Virtual platform
Inter-process communications
Standard RISC processors
Threads
Single, contiguous address space
Hardware features
Designer-provided mappings
CPUs
H/w accelerators
Channels
Memories
Virtualisation layer
Virtualising compiler
uses gcc,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

Figure 4. Tools-oriented overview of CTV.

Flexible mappings from the virtual architecture to actual hardware:

Every virtualisation-based system contains a set of virtualisation mappings which map elements of the software and virtual hardware onto the actual physical hardware. In run-time virtual machines these mappings are implemented by a run-time system and are largely opaque to the programmer. When using 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 control the placement of threads and data. The VP ensures that the software will still operate correctly however these are mapped; only the code's non-functional properties will be affected.

Visibility of custom hardware elements:

Custom hardware 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.

An overview of the CTV system is shown in figure 4. 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) the programmer simply needs to create a new set of VP mappings and the same input code is automatically retargeted to use this new architecture. The main disadvantage of CTV is it limits the run-time dynamism of the system as the topology of the input code must be known at compile-time.

CTV operates at compile-time by refactoring the programmer's input code according to the VP mappings and automatically generating communication libraries and drivers. The refactored code then runs directly on the target hardware. The abstraction and virtualisation layers aid development, but they are not present at run-time and so do not add inefficiency. Also, because the layers are applied by refactoring, CTV does not require new compilers and the source language's existing compiler and toolchain can still be used.

Virtual Platforms have seen some industrial use in simulation and validation. The CoWare system [8] creates an executable simulation of the target hardware and environment. Virtualisation helps

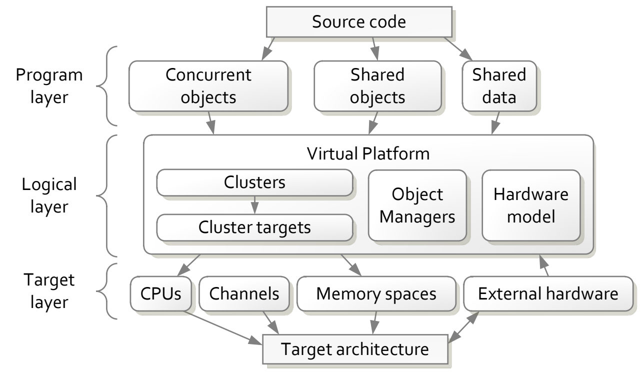

Figure 5. The CTV system model.

the designer to inspect and debug the system, but is not used to assist the programmability of the hardware as is the case with CTV.

6.1 System model

CTV's system model (shown in figure 5) is composed of three layers - the *program layer*, the *logical layer* and the *target layer*. The program layer represents the input source code as three sets:

- **Concurrent objects:** Constructs from the source language with an independent thread of control, such as threads, tasks or processes.
- **Shared objects:** Passive objects with a local state that export a set of functions and procedures that can be remotely invoked by the concurrent objects of the system. Shared objects are used to coordinate the execution of concurrent objects, and to pass data between then in a controlled and thread-safe manner. For example, mutexes, monitors, mailboxes, Ada protected objects etc.
- **Shared data:** Language-level items of shared data. Compound types like structures and arrays are represented by a single shared data object.

The logical layer is used to implement dynamic systems with thread and data migration. This is explored further in [14].

The implementation described in this paper is based on the MCAPI example implementation [25] which is written in C. Consequentially, the input language to the described system is unmodified C which, due to the fact that C does not include primitives for describing multiple threads of control, makes use of the POSIX pthreads library to express parallelism and concurrency control. Both the MCAPI and pthreads APIs are implemented on top of the CTV communications layer, as shown in figure 6. Therefore, threads are concurrent objects, mutexes and condition variables are shared objects, and program variables are shared data objects.

The target layer represents architectures as:

- **Processors:** Processing elements of the target architecture.
- **Communication channels:** Hardware features that transfer data between elements of set C, such as buses and on-chip networks.
- **Memory spaces:** All distinct memory spaces in the system, except caches, which are a feature of the processor $\rightarrow$ memory space connection.
- **Custom hardware elements:** Unique hardware features, like function accelerators and I/O channels, that are to be exported up to the source language.

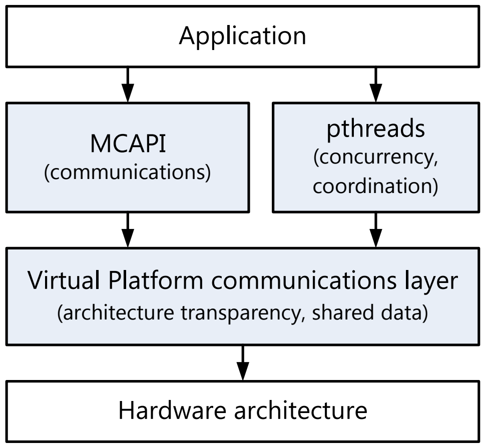
Application
MCAPI
(communications)
pthreads
(concurrency, coordination)
Virtual Platform communications layer
(architecture transparency, shared data)
Hardware architecture

Figure 6. MCAPI is used alongside pthreads. Both are implemented on the CTV communications layer.

In this implementation, each MCAPI node is mapped to a single processor. Threads execute as part of any single node in the system and must execute on the same processor as their host node. It is a simple extension to also allow multiple nodes to exist on the same physical processor. Nodes cannot be split across multiple processors. This is so that the communication topology of the system can be fixed and known at compile-time. With nodes constrained to a single processor it is clear at compile-time to which processor a given message must be delivered, leading to lower run-time overheads and no 'discovery phase' to determine the location of a node when sending messages. Endpoints are therefore effectively created between processors, allowing for much easier mapping to the communications fabric of the target architecture. Due to the fact that CTV provides the compiler with architectural information, it can be used to automatically apply MCAPI zero-copy communications on systems that have shared memory.

MCAPI channels are not directly mapped to the channels of the target hardware. Instead, the VP's communications layer makes use of the underlying hardware channels to implement MCAPI's communications. The mapping of higher-level communications over hardware channels is performed inside the VP and described in more detail in section 7.

6.2 API

The MCAPI API does not have to be modified when combined with CTV. It is the responsibility of the VP to ensure that compatibility is assured. However, as the implementation described in this paper is only preliminary, only a subset of the full API is currently supported. Currently non-blocking operations, timeouts, and zero copy operations are not supported, but these will be added as the implementation is completed.

7. Implementation

The implementation described in this section distributes standard C programs that use the POSIX pthreads library and the MCAPI communications API over complex, multicore systems with heterogenous processors, non-standard memory layouts, arbitrary on-chip communication topologies and application-specific hardware features. This system relies heavily on CTV-based techniques. Full details and worked examples of the CTV analysis and refactoring processes are outside the scope of this paper and can be found in existing CTV literature. This paper focuses on describing the additions required to implement MCAPI on top of CTV's communica-

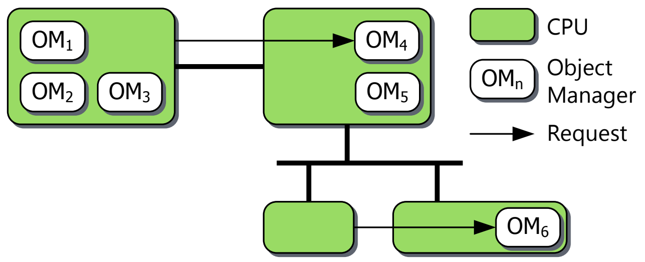

Figure 7. The OM model allows simultaneous requests to execute in parallel without centralised control.

tions fabric and the advantages that can be gained by doing so, but a brief overview is given of all necessary topics.

7.1 CTV's Object Manager model

The MCAPI system implements its channel- and datagram-based communication with CTV's existing communications layer, which is called the Object Manager model. An overview of this model is provided here.

The Object Manager model is a decentralised communication and coordination model that allows CTV implementations to support future architectures of potentially thousands of cores without a loss of scalability. The model is built around the concept of the Object Manager (OM). An OM is a passive object mapped to a CPU that provides services for the threads, shared objects and shared data items which it manages. For example, the OM of a shared data object handles read and write requests. By manipulating the VP mappings the programmer defines the set of OMs, what they manage, and maps the OMs to suitable locations in the target architecture.

During compilation, the threads of the system are refactored so that when they need to interact with a managed object (mutex, shared data item, etc.) they send their request over an appropriate hardware communications channel to the CPU that hosts that object's OM. Drivers for communications hardware are automatically brought in from a provided hardware support library and assembled into a platform library which is linked into the final executable. OMs are implemented as interrupt handlers in the CPUs of the system and can be integrated with a kernel or OS if present. Systems that provide coordination and shared memory using a single OS model [1] may lead to bottlenecks; this is avoided by the OM model, thereby leading to greater parallelism and scalability. Hardware allowing, unrelated OM requests (i.e. sending MCAPI messages between two different nodes or locking two different mutexes) happen in parallel (figure 7).

7.2 VP mappings

VP mappings are provided by the programmer through the use of a simple architectural description language (ADL). This ADL is used to describe the target architecture in terms of the system model's target layer (section 6.1). The ADL is also used to map source language-level constructs over the target architecture. Figure 8 shows a simple ADL snippet that describes a multi-CPU setup connected by mailbox communication channels and non-uniform shared memory. The ADL uses an attribute notation to provide required implementation details (e.g. memory-mapped hardware addresses, interrupt vectors).

When used to implement MCAPI this ADL does not require any extensions because, as described in the following section, mapping information for MCAPI nodes can be inferred from the source code. Through the use of these mappings the programmer can arrange the threads and nodes of their MCAPI program over the

```
CPU0, CPU1, CPU2 : processor Microblaze;
shared01, mem0, mem1, mem2 : memory BlockRAM;
bus01 : channel Mailbox;
bus02 : channel Mailbox;

CPU0^memory = [mem0(0x00000000),
  shared(0x40000000)];
CPU1^memory = [mem1(0x00000000),
  shared(0x40000000)];
CPU2^memory = mem3;

bus01^endpoints = [CPU0(0x80000000),
  CPU1(0x80000000)];
bus02^endpoints = [CPU0(0x82000000),
  CPU2(0x80000000)];
```

Figure 8. ADL describing a complex three-core system.

architecture with a level of control that is not normally possible. The topology of both the software and hardware is known at compile time in CTV, so the underlying communications drivers and libraries can be automatically-generated. This allows very fast design-space exploration as the input source code does not need to be rewritten for different mappings.

For example, consider the C declaration:

```
int sharedarray[50];
```

The programmer can place this in shared memory in the ADL as follows:

```
sharedarray : variable;
shared12^variables = [sharedarray];
```

On CPU0, the code is refactored by Anvil to the following:

```
int sharedarray[50]
    __attribute__((section ("shared12")));
```

and the following section added to CPU1's link script:

```
SECTIONS
{
  . = 0x20000000;
  shared12 : {}
}
```

Accesses to `sharedarray` are then refactored to use Anvil's shared memory system which ensures coherency etc., discussed in existing CTV literature.

7.3 Static analysis and refactoring

As with existing CTV implementations, the presented system uses static analysis to determine offline the topology of the communications and nodes in the input program. The threads of the system must be declared global and static so that they can be extracted from the source code and mapped to processors of the target architecture. This places restrictions on the run-time variability of the C code (no dynamic thread creation), but given the target domain of high-integrity embedded systems these restrictions are reasonable. A simple extension can allow threads to be dynamically created within a node if this behaviour is required, but the overall system topology must remain static.

The input code is parsed to an AST and the symbol table generated. Objects of type `pthread_t` are identified and data flow analysis is used to associate them with calls to `pthread_create`. This allows the refactoring engine to determine which C functions are

```
void _anvil_send_to_cpu(int cpuid,
        int *pkt, int len) {

  int x;
  for (x = 0; x < len; x++) {
    switch(cpuid) {
      case 1:
        mbox_write(bus01, pkt[x]);
        break;
          case 2:
        mbox_write(bus02, pkt[x]);
        break;
      default:
        break;
    }
  }
}
```

Figure 9. Autogenerated part of the Anvil comms. layer.

used as thread bodies. If the programmer's code is not statically analysable then they must recode or manually annotate to provide this data. From this information, the single input program can be split into a set of output programs, one for each target processor according to the VP mappings provided by the programmer. This requires the creation of a new `main` function for each split program that waits until it is woken by a `pthread_create` call from elsewhere in the system before calling the thread body. Recall that threads are mapped to the processors of the target architecture by the programmer (section 7.2). Reachability analysis is used to determine the code that is required in each split program.

The analysis system then determines which MCAPI nodes are located upon which processors. Each node has an ID which is set by the programmer in the input code. It is necessary to determine how these node IDs relate to the processors of the target architecture. This is done by looking for calls to `mcapi_initialize` and analysing the call's parameters. The analysis has already determined which functions correspond to thread bodies, and the programmer has mapped threads to processors, so from this it can determine which node will be initialised on which processor. For example, if thread t is mapped to processor p and t makes the following call:

```
mcapi_initialize(200, &version, &status);
```

the analysis can determine that processor t is assigned node ID 200 by the software. Again, if this information cannot be statically determined then the programmer must reexpress their program to make it so. This information is used to build a small private function `nodeid_to_cpuid` that is used by the MCAPI library at runtime to determine which processor to send a given MCAPI message to. The CTV communication layer has already dynamically generated architecture-specific functions for each processor that are used to communicate with each other processor in the system, using on-chip communications as appropriate. The MCAPI implementation can make use of this. In the three-core example system in figure 8, at compile-time the section in figure 9 is generated as part of the communications layer.

Similar code is generated to manage interrupts and other hardware. `_anvil_send_to_cpu` is used directly by the MCAPI implementation to transport data between processors, assuring low-overhead operation on varying hardware.

In the current implementation, each node stores its endpoints in an array of fixed maximum size that is manipulated by the `mcapi_create_endpoint` and `mcapi_delete_endpoint` functions. The endpoint also maintains a set of transmit and receive message buffers that use the underlying Anvil communications layer as above.

Blocking (for example on message receipt) is implemented by using a stub function of the Anvil layer. In systems with no kernel, Anvil spin locks waiting to be interrupted. In systems with microkernels the blocking features of that kernel are called so that other threads may execute.

7.4 Messaging protocol

The VP implementation already defines an internal set of messages that are sent between OMs to implement its various features. These messages perform the following actions:

- Create, end and join threads.
- Wait for a mutex lock, query a mutex state, reply when lock is obtained.
- Wait for a condition variable, signal a condition, signal all (broadcast) a condition.
- Read and write shared variable data.

Whilst it is possible to use these messages solely to implement other features, initial experiments confirmed that this results in high latency as a large number of low-level messages need to be processed for a single higher-level event. Instead, extra messages are introduced into the communications layer to implement specific MCAPI operations. The following messages are required:

- Request an endpoint from a remote node (and the reply message).
- Send a datagram (connectionless message).
- Establish and close channels between two endpoints and send data along a created channel.
- Request status and capability information from another node.

These messages increase the code footprint of the communications layer slightly, but result in lower latency communications. Note that large amounts of the MCAPI API do not require messages to be defined (for example mcapi_create_endpoint) because they operate only on the local node.

8. Evaluation

This section presents an initial evaluation of the combined MCAPI and CTV implementation. As MCAPI is a new standard, there is a lack of comparison points available to evaluate against. (MCAPI only currently has one official implementation, and it cannot be used on the complex architectures which this paper discusses.) Therefore, this section cannot aim to demonstrate specific speed or latency targets or speed-ups, but instead it aims to show the benefits of programmer-directed architectural mapping that are afforded by the approach in this paper. Section 8.1 shows how application knowledge can be used by the programmer to efficiently exploit complex architectures better than would be possible with an MCAPI system that does not use CTV. The same source code is reimplemented over a complex architecture by changing the VP mappings, resulting in different temporal properties but the same functional properties. Section 8.2 discusses the overheads in the implementation.

8.1 Architectural mapping

As established in section 5, the MCAPI specification and the programming languages to which it is bound do not allow the programmer to reason about the mapping of their program to the tar-

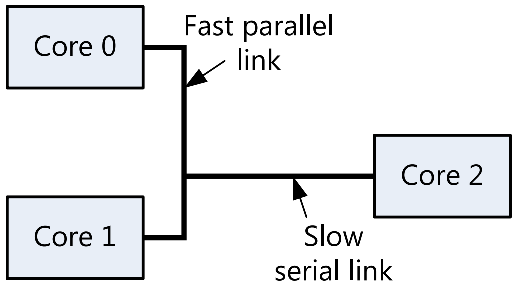

Figure 10. The implementation architecture

Application	Before moving	After moving	Speed up
FIR (1 pass)	918,549	815,922	1.13
FIR (2 pass)	1,159,384	1,056,500	1.1
Collatz (1k evals)	4,415,843	3,462,921	1.28
Collatz (2k evals)	10,204,554	7,004,586	1.46
3DES	1,729,640	1,544,322	1.12

Figure 11. Evaluation times for test applications (clock cycles)

get architecture. Consider the architecture in figure 10. The figure shows a three-core architecture with no shared memory and heterogeneous inter-processor communication channels. Cores 0 and 1 are linked by a fast mailbox-style FIFO that can transfer data at over 1MB/sec, whereas communications with core 2 requires the use of a much slower serial link (approx. 300KB/sec) that also has a higher transport latency. The architecture is implemented on a Xilinx Spartan 3 4000 FPGA using the Microblaze soft processor core [28].

Clearly the placement of threads throughout this system is important to maximise application throughput. The programmer should place tasks with lower communication requirements on core 2, so that the slower link does not affect the overall system as much. MCAPI's node and endpoint model of communications targets this architecture well, whilst the VP of CTV allows the programmer to perform deliberate and fast architectural mapping from a single program. A number of example applications were built, all using at least three threads of control so that all cores are active. For this test, the threads with the highest communication requirements of each application were identified by profiling and from application-specific knowledge. The results in figure 11 show the effect of moving the thread with highest computation requirements from core 2 to core 0. As expected, this reduces overall execution time in all cases, but by differing degrees depending on the specific benchmark. It must be observed that this move cannot be specified in a normal C-based, embedded MCAPI implementation. In the case of the FIR filter or 3DES programs, the controller core has to pass large volumes of data so moving the dispatch thread to a core where more outgoing links are faster has a positive effect. The improvement is even more pronounced in the case of the Collatz engine because it requires less computation per unit of dispatch than FIR, so improving latency has a higher percentage effect (up to 146%).

The important fact about this experiment is that in all cases the application's code does not need to be rewritten, as it would need to be without the use of CTV. The programmer simply changes the VP mappings to move the target thread and the refactoring engine (section 7.3) inserts appropriate calls to the CTV communications layer, which is automatically built according to the target architecture. This shows the potential for rapid design-space exploration enabled by the combination of MCAPI and CTV. CTV also allows the seamless use of differing communication channels, and can target changing architectures without the need for code refactoring. These topics are covered in existing CTV literature and augments this work.

8.2 Overheads

The largely static nature of CTV allows for the MCAPI implementation to be very lightweight. As discussed in section 7.3, MCAPI nodes remain static throughout the execution of the program and their locations are fixed to individual CPUs. This information can be used by the MCAPI implementation to perform offline routing of many packets, and therefore implement low-latency datagram communications. CTV does allow for dynamic behaviour but this overhead is only included when it becomes necessary, and communications between static nodes is unaffected. This also allows for the code footprint to be kept small, as dynamic behaviour is only linked in when required.

The CTV communications layer and pthreads implementation are around 8.6kB of compiled Microblaze code, with the MCAPI implementation adding another 3.6kB. This varies for different target architectures as support for channels and peripherals are linked in only as they are required. For comparison, Xilkernel [27], a widely-used microkernel for FPGA-based architectures requires 22kB.

The scale of these sizes is highlighted by considering the memory footprint of run-time middleware solutions to perform communication, such as CORBA ORBs. Although not directly comparable due to their differing feature sets, Real-Time CORBA is aimed at deployment in real-time and embedded systems despite the TAO ORB measuring 2075kB in size, and the ZEN Real-Time ORB measuring approximately 2539kB [19]. This shows that the approach in this paper can be implemented in a very small amount of code, making it suitable for use on highly resource-constrained systems.

The MCAPI layer can also be shown to display good performance characteristics. The total latency and throughput for a given message are primarily determined by the communication channel in use, but it is possible to measure the latency through the MCAPI and CTV communications layers. Between two Microblaze cores, messages take up to 994 clock cycles to establish a connection and then an average of 48 clock cycles (plus physical transfer time) per byte to transfer from one core to the other. The layer therefore adds on average 24 cycles on the sender side and 24 on the receiver size once the channel is established. Clearly these numbers will vary on different processors with different memory architectures and compilers, but this shows that overheads in this implementation can be kept to only small fractions of the actual physical transfer time (which is outside the control of the software).

9. Conclusion

This paper has discussed the novel combination of an industry standard communications API and Compile-Time Virtualisation (CTV). The work argues that this technique allows for better exploitation of complex embedded architectures.

The Multicore Communication API (MCAPI) has been proposed by the Multicore Association to provide a standardised API for communication across multicore embedded architectures. It provides lightweight, efficient communications and is used to allow standard languages to communicate efficiently in a heterogeneous, distributed architecture. As an API, MCAPI is required to use the same programming model as the language to which it is bound. However, existing programming languages are lacking in their support for complex non-standard architectures. Specifically, the programmer cannot express from their source code the mappings between elements of their code (i.e. threads and data structures) and the hardware features to which they should be mapped. To solve this, the MCAPI implementation is combined with CTV. CTV is a technique which introduces a Virtual Platform to hide architectural complexities whilst still allowing programmer control over program mappings. The virtualisation only exists at compile-time, leading to very low overheads.

The MCAPI implementation is shown to be useful for targeting standard MCAPI programs that use the POSIX pthreads library onto complex FPGA-based architectures. The programmer can perform design space exploration quickly without recoding. Due to CTV's compile-time nature, the layer is shown to perform well with low message latency.

References

[1] J. Agron and D. Andrews. Building heterogeneous reconfigurable systems with a hardware microkernel. In *Proceedings of CODES+ISSS '09*, pages 393–402, New York, NY, USA, 2009. ACM.

[2] Baumann et al. The Multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of SOSP '09*, pages 29–44, New York, NY, USA, 2009. ACM.

[3] R. Brukardt. The Ada95 language reference manual - Appendix E, Distributed Systems (International Standard ISO/IEC 8652:1995). http://www.adaic.org/standards/95lrm/html/RM-E.html.

[4] W. W. Carlson, D. E. Culler, and E. Brooks. Introduction to UPC and language specification. *CCS-TR-99-157*, 1999.

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

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

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.

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

[9] P. Dibble and A. Wellings. JSR-282 status report. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM International Conference Proceeding Series, pages 179–182, New York, NY, USA, 2009. ACM.

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

[11] 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.

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

[13] I. Gray and N. Audsley. Exposing non-standard architectures to embedded software using Compile-Time Virtualisation. *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '09)*, 2009.

[14] I. Gray and N. Audsley. Supporting islands of coherency for highly-parallel embedded architectures using Compile-Time Virtualisation. In *13th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2010.

[15] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994.

[16] M. C. Gthe, D. Wengelin, and L. Asplund. The distributed Ada run-time system DARTS. *Software: Practice and Experience*, 21:1249–1263, 1991.

[17] J. Holt. Designing an industry standard api to manage multicore system resources. http://www.multicore-association.org/webinar/090811_MRAPI.pdf, August 2009.

[18] Institute of Electrical and Electronics Engineers. POSIX.1c, threads extensions (IEEE Std 1003.1c-1995), 1995.

[19] R. Klefstad, M. Deshpande, C. ORyan, A. Corsaro, A. S. Krishna, S. Rao, and K. Raman. The performance of ZEN: A real time CORBA ORB using real time java. In *Proceedings of Real-time and Embedded Distributed Object Computing Workshop*. OMG, September 2002.

[20] J. Maloy. TIPC: Providing communication for linux clusters. In *Proceedings of the Linux Symposium - Volume 2*, pages 347–356, 2004.

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

[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] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[24] S. Sharma, G. Gopalakrishnan, E. Mercer, and J. Holt. Mcc - a runtime verification tool for mcapi user applications. In *Proceedings of Formal Methods in Computer Aided Design 2009 (FMCAD09)*, 2009.

[25] The Multicore Association. Multicore communications API specification V1.063 (MCAPI). http://www.multicore-association.org/workgroup/mcapi.php, March 2008.

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

[27] Xilinx Corporation. Xilkernel. http://www.xilinx.com/ise/embedded/edk91i_docs/ xilkernel_v3_00_a.pdf, December 2006.

[28] Xilinx Corporation. Microblaze processor reference guide. UG081 v9.0, 2008.