Supporting Islands of Coherency for highly-parallel embedded architectures 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

As their complexity grows, the architectures of embedded systems are becoming increasingly parallel. However, the frameworks used to assist development on highly-parallel general-purpose systems (such as CORBA or MPI) are too heavyweight for use on the non-standard architectures of embedded systems. They introduce significant overheads due to the lack of architectural and structural information contained within most programming languages. Specifically, thread migration across irregular architectures can lead to very poor memory access times, and unconstrained cache coherency cannot scale to cope with large systems.

This paper introduces an approach to solving these problems in a scalable way with minimal run-time overhead by using the concept of 'Islands of Coherency'. Cooperating threads are grouped into clusters along with the data that they use. These clusters can then be efficiently mapped to the target architecture, utilising migration only in the areas where the programmer explicitly declares it.

This is supported through the use of an existing technique called Compile-Time Virtualisation (CTV). CTV does not support run-time dynamism, so it is extended to allow the implementation of Islands of Coherency. The presented system is evaluated experimentally through implementation on an FPGA platform. Simulation-based results are also presented that show the potential that this approach has for increasing the performance of future embedded systems.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; C.0 [**General**]: Modeling of computer architecture; D.3.4 [**Programming Languages**]: Processors—*Retargetable Compilers*

General Terms

Design, Languages, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SCOPES'10, June 29–30, 2010, St. Goar, Germany.
Copyright 2010 ACM ...$10.00.

1. INTRODUCTION

Embedded system architectures range from simple single CPU systems, to large architectures with dozens of heterogeneous interacting cores and non-standard memory layouts. Also, systems are becoming increasingly dynamic due to techniques to support user-configurable software, power scaling, fault tolerance, etc. However, the programming languages used to program these systems (e.g. C) assume an architecture with a single logical address space and single shared bus that can be far removed from the actual hardware architecture and behaviour. For multi-CPU heterogenous embedded systems, it is difficult for the programmer to map the application functionality to the target platform and to exploit its dynamic characteristics efficiently.

This paper considers a solution to these problems using an 'Islands of Coherency' model. In this model, the programmer is able to group related threads and data such that they maintain locality during migration, leading to more efficient systems. The model also lowers cache coherency requirements, leading to increased scalability.

In this paper we implement Islands of Coherency with an existing technique called Compile-Time Virtualisation (CTV). CTV has previously been shown to allow programmers greater control over the mapping of their applications to non-uniform embedded architectures. However, as a compile-time technique it does not support run-time dynamism. In this work, CTV's system model is extended to support the required dynamism and an implementation is created and evaluated.

Section 2 further explores the problem considered by this work while section 4 gives an overview of CTV and details the extensions added to support dynamic systems. Section 5 describes an implementation of CTV which supports these extensions, and sections 6 and 7 evaluate and conclude.

2. MOTIVATION

Embedded systems have tight power and cooling constraints which limit processing clock frequencies. To provide greater computational power therefore, system designers exploit on-chip parallelism and application-specific features. The increasing disparity between clock speeds and memory access times (the 'memory gap') forces the adoption of non-standard memory layouts that can provide more localised storage closer to the site of computation and greater effective bandwidth. Embedded architectures also tend to contain features that are implemented using dedicated hardware, reconfigurable logic, or custom CPUs. Embedded architectures may contain:

- Multiple heterogenous processing elements [12, 6].
- On-chip networks and buses, possibly spanning clock domains [8, 26].
- Non-standard memory hierarchies with shared memory and the integration of new memory technologies [11, 1].
- Unique features of the implementation fabric such as DSP cores, SIMD units, or other custom hardware [21, 23].

Whilst many embedded systems are entirely static (e.g. simple single-CPU systems), as embedded platforms become increasingly parallel the issue of dynamic migration of computation is introduced. For example, modern mobile phones contain multiple interacting processing cores which are exploited to meet the computational requirements of the broadband communications stack and multiple user applications. Important dynamic features that are being introduced in such embedded systems are:

- **Multiprocessor systems:** As with desktop machines, devices with multiple execution units will support a POSIX-style threading model that may allow threads to migrate and be scheduled across cores.
- **Power scaling:** Reducing power consumption is essential for many embedded devices. A device that is under low load may switch some cores off and migrate all their computation onto a smaller active set.
- **Fault tolerance:** As cores fail, if possible it is desirable to migrate any threads that were executing on them to other areas of the system.
- **Load balancing:** Load balancing systems allow for systems to better deal with infrequent bursts of high volumes of computation (e.g. in a network switch).
- **Dynamic reconfiguration:** FPGA technologies allow embedded architectures to change at run-time through a process called partial dynamic reconfiguration [24], leading to high amounts of dynamism.

A key issue is the efficient exploitation of the dynamic features of the target platform by the programmer, through the use of a standard programming language.

2.1 Programming multiprocessor embedded systems

With programming multiprocessor embedded systems there are two distinct issues to consider: program structure and memory coherency. The former considers the mapping of the program onto the underlying multiprocessor (and non-uniform memory) architecture. The latter considers the need to provide the illusion of a single address space from the program perspective to support existing languages (e.g. C). We now consider these issues in detail.

Program structure

Programs are generally expressed in units such as threads and data objects. However, the relationship between threads and data (i.e. which data objects are required by a thread) is not typically present in programming languages. When a thread migrates from one CPU to another, the language runtime lacks the necessary structure that would allow it to also migrate any related threads and the data they use. This unguided thread migration can lead to increased inter-thread distances and therefore higher communications latency and poor memory access times, as shown in figure 1. This problem affects embedded systems particularly, because they are rarely homogenous grid architectures of the kind found in supercomputing environments.

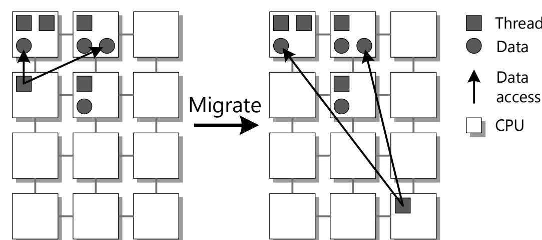

Figure 1: Unguided thread migration can lead to increased inter-thread distances and poor performance.

In an attempt to mitigate this problem, some languages allow the programmer to bind threads to only execute on a subset of available CPUs. Thread affinities in Java [9] are one such example. Affinities describe which CPU a thread may be scheduled upon at runtime, but there is no way to bind items of shared data to that thread or to state that threads should be grouped and moved together. It may sometimes be possible to infer this data to a limited extent with static analysis, but this does not help in the general case.

Chapel [7] uses the concept of *locales* to group threads and data. A locale can then be bound to specific nodes of the architecture, much like affinities. We note that locales are low-level, forcing all threads in a locale to execute on the same processing node and limiting the sharing of data between locales. Also, Chapel relies on a shared-memory model and a regular grid architecture making it rather heavyweight for many embedded systems. A related language, Unified Parallel C (UPC) [5] allows threads to have private and shared data, and also introduces the concept of data affinity which states that a particular thread 'owns' a particular data item. Unfortunately it does not allow clustering of threads as the model is flat rather than hierarchical, and like Chapel it is not an embedded language and was designed for heavyweight supercomputing architectures.

We note that solutions used in general multiprocessor (and distributed) architectures employ middleware like MPI [16], CORBA [20] or PVM [13]. These are not appropriate for embedded systems due to their inefficiency and need for complex Operating System and communications support. Also, such systems are frequently multi-program, whereas this work is interested in distributing a single application across the target system.

Coherency

Whilst embedded architectures employ complex, hierarchical, heterogenous models, almost all existing languages assume a contiguous global address space. The programmer cannot effectively map the data of their program onto the

memory hierarchy, and so therefore has little control over data transfers between CPUs and the memory spaces of the architecture. The result of this single address space assumption is that many software languages place heavy demands on cache coherency algorithms, usually that the entire system is kept coherent. This is very expensive (in terms of execution time and required hardware) and it does not scale to support large numbers of caches. [2] On embedded platforms it is even worse because their application-specific nature means that frequently the programmer actually only needs to keep a few small areas of the system coherent, as determined by their application.

Lightweight schemes that limit coherency into islands are more scalable than complete coherency solutions [19], but they do require significant support from the programming environment which is typically not available. C assumes that the entire program is within a single contiguous address space, and its non-analysable pointer arithmetic requires perfect coherence across all memory. Furthermore, because threads are not a first-class part of the language they cannot be reasoned about in terms of the data that they use. Ada's Distributed Systems Annex (DSA) [3] allows for the notion of separate memory spaces, but it produces very heavyweight partitions that cannot communicate or share data without the use of explicit communication. It uses no coherency at all between partitions, no task migration, and inside a partition the memory model is similar to C. Java largely ignores different memory spaces as it is designed to operate on top of the JVM, but support has been added to the Real-Time Specification for Java [14] to allow the programmer to place objects in different memory spaces using the concepts of scoped and immortal memory. This allows placement throughout the architecture but the resulting code is very hardware-specific. There is no concept of threads and data being related and coherency is still presumed to be across all memory. As mentioned previously, UPC allows threads to claim an item of data as its own, but the model requires all shared data to be accessible to all threads.

Some recent work has considered ways to limit this coherency problem. Huang et. al. [17] augments the programmer's source code at points where locks are requested and released with library calls to determine what shared data is required by which parts of the design. Similarly, Virtual Tree Coherence (VTC) [10] uses a tree-based coherency system to limit coherence actions to the necessary subset of the nodes of an on-chip network.

Summary

Modern languages are built upon a set of architectural assumptions and abstraction layers that simplify development but do not allow the programmer to reason about or exploit features of the underlying hardware. For general-purpose systems this is preferable because the hardware is largely uniform. However, for dynamic embedded systems the lack of precise architectural mappings causes difficulties mapping the application to the platform, and can lead to unnecessary overhead. This causes particular problems in systems that use dynamic migration, as locality between threads and their data is lost. Also, existing programming models tend to place infeasible demands on the cache coherency system by assuming system-wide coherency.

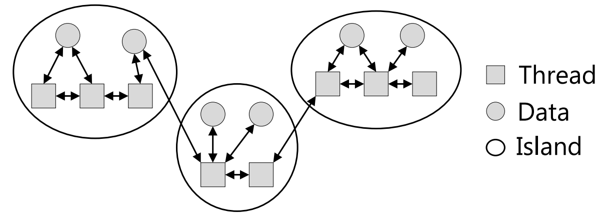

Figure 2: Islands of coherency in a large software system.

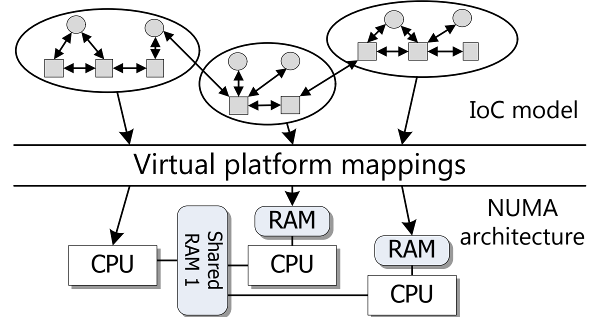

Figure 3: Supporting IoC in a standard programming environment

Solutions for programming multiprocessor embedded systems with existing languages (like C) must provide:

- the ability for the programmer to control the mapping of elements of the program onto the underlying multiprocessor (and non-uniform memory) architecture;
- efficient mechanisms to migrate groups of related threads and data.

3. ISLANDS OF COHERENCY (IOC)

As a potential solution to the problems noted in the previous section, this paper uses the concept of *Islands of Coherency* (IoC). In this model, rather than considering the whole system at a single level of granularity, the threads of the system are grouped into cooperating clusters. The threads in these clusters typically coordinate on a particular subproblem, sharing the same input and temporary data. Communication between clusters is less frequent, higher-level, and not as data-driven. These clusters are referred to as IoC because caches within an island are kept tightly coherent with each other (e.g. with hardware support), but not with those outside the island which may rely on simple message passing or no coherence at all. This layout is shown in figure 2. Due to its hierarchical nature this model is very scalable. The model is not new as aspects of it can be seen in many high-performance parallel frameworks (like CORBA, MPI), but this paper is interested in an efficient embedded implementation.

In the IoC model, when a thread from a cluster is migrated the system can choose to also migrate the other threads and data of its cluster. Whilst this carries a larger initial cost, it ensures that clusters retain their locality and as a result subsequent calculations can be much faster.

The models of existing programming languages do not

provide this clustering information and so migrations cannot be guaranteed to preserve locality. However, it is possible to support IoC in existing programming languages through the use of a mapping layer, as shown in figure 3. This mapping can be performed at either run-time or compile-time. This paper attempts to minimise overheads to produce a system suitable for high-performance embedded systems, so a compile-time solution is presented. A technique called Compile-Time Virtualisation (CTV) is used, which is detailed in the following section. CTV does not support dynamic behaviour, so this paper extends its model to allow the implementation of IoC and these extensions are detailed also.

4. COMPILE-TIME VIRTUALISATION

Compile-Time Virtualisation (CTV) [15] 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), which 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, it may present a single logical address space or uniform inter-thread communication. In essence, as with standard run-time virtualisation the layer is tasked with ensuring that the programmer's code operates correctly without low-level programmer intervention. Because the layer hides low-level implementation details it allows for code to be architecturally-neutral, as the developer does not need to break the abstraction models of the language.

Flexible mappings from the virtual architecture to actual hardware:

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

Visibility of custom hardware elements:

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

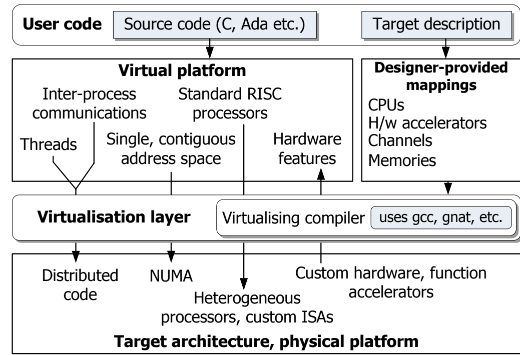

Figure 4: Tools-oriented overview of CTV.

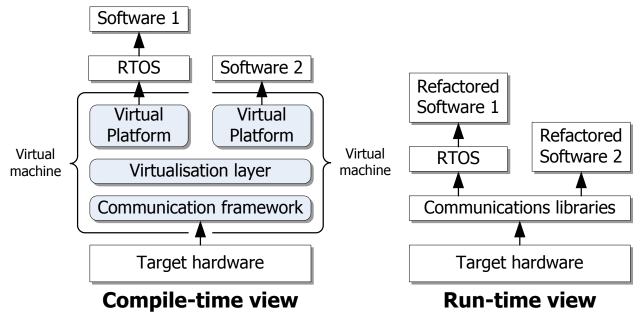

Figure 5: How the compile-time view of CTV differs from the run-time view.

An overview of the CTV system is shown in figure 4. The primary difference between a run-time VM like Java and CTV is that by moving the virtualisation to compile-time, run-time overheads are reduced to a minimum. If the hardware design later changes (for example, the addition of a CPU or a new memory layout) a new VP is generated and the same input code is automatically retargeted to use this new architecture. As its name implies, Compile-Time Virtualisation is differentiated from run-time virtualisation systems by the fact that its virtualisation layer only exists during compilation, as illustrated in figure 5.

Moving the virtualisation to compile-time results in lower run-time overheads and greatly facilitates code-reuse because it allows the use of unmodified languages and compilers. However, the disadvantage is that the run-time dynamism of the system is limited, as will be discussed in the following section.

4.1 System model

The CTV system model shown in figure 6 represents source code as a 3-tuple of sets called the *source tuple* (T, P, D) where:

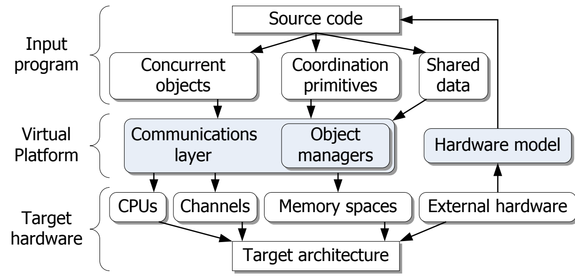

Figure 6: CTV system model

- T = **concurrent objects (threads):** A set of the units of concurrency of the source language. For example, threads, tasks or processes.
- P = **coordination primitives:** Language constructs used to coordinate the items of set T, such as mutexes, monitors, mailboxes etc.
- D = **shared data:** Language-level items of shared data. Compound types like structures and arrays are represented by only one set member.

Similarly, target architectures are represented by a 4-tuple called the target tuple (C, L, M, H) where:

- C = **CPUs:** Processing elements of the target architecture.
- L = **communication channels:** Hardware features that transfer data between elements of set C, such as buses and on-chip networks.
- M = **memory spaces:** All distinct memory spaces in the system.
- H = **hardware elements:** Unique hardware features, like function accelerators and I/O channels, that are to be exported up to the source language.

The VP is a mapping between these two tuples:

$$(T, P, D) \xrightarrow{VP} (C, L, M, H)$$

This mapping distributes the source program over the target architecture. Mapping a thread to a CPU informs the compilation process upon which CPU the programmer wants the chosen thread to execute. Data is placed into memory in the same way. Due to the compile-time nature of CTV these mappings must be compile-time static. This system model, therefore, does not allow threads or data to migrate at runtime as is required by the IoC model. This work introduces extensions to CTV to allow the use of the IoC model, which are detailed in section 4.3.

The CTV system model requires the VP to implement complete compile-time mobility of code and data - the programmer must be able to place elements throughout the system and the VP must ensure that the functional properties of the code are not affected. To do this, the VP implements three systems - a distributed shared memory system; concurrency primitives that operate across the target architecture; and communications libraries that make transparent use of the underlying communications media of the architecture. These are implemented using a scalable communications and coordination model called the Object Manager model.

4.2 The Object Manager model

CTV is designed to support architectures with potentially thousands of interacting cores. As a result, any form of centralised control must be avoided or it will result in a loss of scalability. Therefore, shared memory access, concurrency control, and inter-thread communications are implemented with a model known as the Object Manager (OM) model [15].

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 source tuple (threads, concurrency primitives and shared data items). 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 and is responsible for implementing cache coherency. 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 slightly. 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:

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

4.3 Extending CTV with logical objects

The system model shown in sections 4.1 and 4.2 is entirely static - the VP mappings cannot be changed at run-time. Therefore, previous iterations of CTV could not support any migration of threads or data. In this paper we add a *logical layer* to the model which is an abstraction of the target tuple that notionally behaves in the same way as its physical counterpart, but may be distributed across multiple members of the target tuple. The layer is composed of a set of *cluster* objects (u) and introduces the concept of logical CPUs (c) and logical memory spaces (m). The logical layer exists as part of the VP, and therefore sits between the source and target tuples.

$$(T, P, D) \xrightarrow{logical} (u, c, m) \xrightarrow{physical} (C, L, M, H)$$

A cluster is an abstract concept that describes a group of tightly-related threads and the data that they use, and can be seen as similar to the idea of a Chapel locale. Every thread and data item from the source tuple must be mapped to exactly one cluster. Every cluster must be mapped to at least one logical CPU and at least one logical memory space, but it can be mapped to more. Threads assigned to a given cluster may migrate between all of the logical CPUs to which the cluster is assigned. Similarly, data items assigned to a cluster may be located in any of the logical memory spaces to which the cluster is mapped. Logical CPUs and memory spaces are abstractions of respective members of the target tuple. Threads that a cluster places into a logical CPU may

be scheduled on any of the target CPUs that comprise it.

The two levels of grouping in this model are required to allow the expression of realistic dynamism. Cluster objects are used by the application programmer to inform the compile-time system how their application and its data are linked. Logical objects are used by the hardware designer to inform CTV about areas of the architecture that are suitable for containing computation clusters. At compile-time, CTV creates a system that ensures clusters retain locality and only migrate between the logical elements that they are assigned to. This is done by extending the OM model to support migrating threads and data and is discussed in section 5.1. It is up to the implementation as to whether the system uses a true load balancing system, standard multiprocessor scheduling, or a redundant fault tolerance mechanism.

This model allows the programmer to inform the compiler where clusters of threads and data may be situated in order to ensure that tightly-coupled threads remain tightly-coupled. Clusters also tell the compiler about the amount of cache coherency required in the system. CPUs that are within the same logical memory space are kept coherent, and those from different memory spaces are not. The actual algorithm used is not specified by CTV, and is implementation-dependent. Any existing system can be used.

Note that it is possible to map more than one cluster to the same logical items, thereby allowing IoC to overlap. Also if the logical layer is not required then the programmer can use a 1-1 identity mapping, which creates a single logical element for each target element, a single cluster for each logical element, and maps them one-to-one. The identity layer has no effect and introduces no dynamism. Finally, if required, threads can still also be directly mapped to target CPUs, to allow the programmer to 'hint' to the compiler where the thread should execute.

4.4 From dynamic-default to static-default

As previously mentioned, unless logical CPUs and memory spaces are used in the CTV system it is assumed that the input and output tuples are static and the mappings between them will not change at runtime. In other words, unless stated otherwise, dynamism is assumed to be virtually non-existent. Programs use a fixed number of threads that do not migrate between CPUs, items of shared data cannot be created or freed, synchronisation features are fixed etc. This results in a system which is very similar to the Ravenscar subsets of Ada [4] or Java [18] that are designed to make code easier to analyse for correctness in safety-critical contexts.

The use of logical elements allows the programmer to define specific areas of their system in which dynamism may occur, limiting its effect on the entire system. Dynamic behaviour is restricted and must be specifically enumerated, and the provision for dynamism in one part of the system does not directly impact the correctness or execution time analysis of another (static) part of the system. It could indirectly affect the analysis if the two sections communicate or compete for shared resources, but techniques such as sporadic servers [22] can be used to mitigate this interference. This represents a major conceptual change from the way that systems are normally designed, resulting in designs that are more predictable and more amenable for use as real-time systems.

In standard development, if dynamic features such as arbitrary thread creation and migration are supported by the language then they may happen at any time and so run-time systems must always be built to support these mechanisms. The overhead associated with these features must be paid, even when they are not actually required by the code. The only time this overhead can be avoided is if the compiler can perform sufficient static analysis to determine that support for the unused features can be safely removed. Unfortunately this is often equivalent to the halting problem in the general case, meaning the compiler is forced to assume the worst case and include it. CTV's approach does not suffer from this problem. With the assumption of no dynamic behaviour the 'default' run-time support required is minimal. Any extra dynamic behaviour must be specifically justified and enumerated by the programmer resulting in a system with the minimum required overhead.

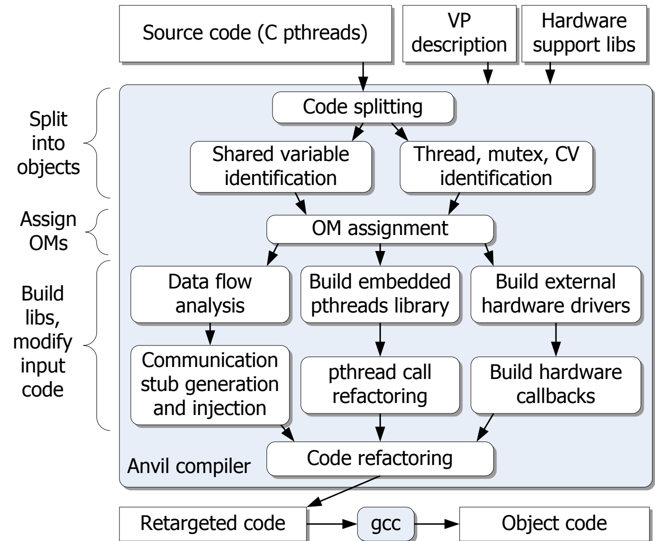

Figure 7: The main stages of the Anvil refactoring engine.

5. IMPLEMENTATION

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. The POSIX standard pthreads library is used to provide multiple threads of control. Anvil uses an unmodified version of the `gcc` compiler internally for its object code generation.

The programmer uses a simple architecture description language (ADL) to describe the target architecture in terms of the target tuple (C, L, M, H) and map to it features from the source language (T, P, D). Examples can be found in section 6. The majority of Anvil's work involves refactoring the programmer's input code according to the mappings in its VP and producing standard ANSI C programs that it compiles with `gcc`. The refactoring process is illustrated in figure 7 and discussed in more detail in [15].

```
1  extern _anvil_a_var_t _anvil_accessedvars[];
2  int x; //Storage for remote data, (could be malloced)
3  void main(void) {
4     _anvil_accessedvars[0].data = (unsigned char *) &x;
5     _anvil_read_sv(0, 0, 4, 1); //id, offset, bytes, OM id
6     printf("%d\n", x);
7     x = x + 1;
8     _anvil_write_sv(0, 0, 4, 1);
9  }
```

Figure 8: Simple example of Anvil's refactoring to implement shared memory. Lines 1, 2, 4, 5 and 8 are added by Anvil.

5.1 Extensions to support logical objects

Previously, Anvil could be certain where every object was in a given system because they did not move at run-time. CPU-specific communications libraries were generated during compilation that implement these communications directly with no lookups or other inefficiencies. To implement dynamic objects however it is clear that a level of indirection has to be implemented as the sender does not know where the receiver is currently located.

Coordination primitives (mutexes etc.) do not migrate under the logical layer so their implementation does not need to change. However, threads and data items do. When communicating with one of these dynamic objects the communications must be routed through its object manager, which maintains a record of the item's current location so it knows where to send the message. This means that the location of OMs must not change. In practice this turns out not to be much of a restriction because the majority of embedded systems are static, and only proportionally small sections are dynamic.

This extra resolution hop is unavoidable and all dynamic systems have a similar mechanism, which usually involves querying an OS running on a given core or making use of communication broadcast to ask where an item is. There are a number of potential optimisations that can reduce its impact. For example, the sender can cache receiver locations so that it does not need to make a request for each message. Each object manager must maintain a list of the items that have requested location information so that if its managed item migrates it can inform all clients to keep their cache up-to-date.

An important point is that due to the compile-time nature of CTV any inefficiency is *only* introduced for dynamic objects (where it is unavoidable). Static communications are totally unaffected. This is an example of the way in which CTV's 'static by default' approach reduces overheads to the minimum required by the system.

Migrating data items introduces a number of other considerations. For systems that are implementing migration for fault tolerance, the system must reserve enough space in each target memory space to guarantee that there is room for the item to migrate into. In systems with less stringent requirements, a small run-time system must be implemented that can attempt to allocate enough memory to perform the move. If this allocation fails then the migration cannot take place. It is up to the implementation to decide on the appropriate choice. Anvil uses the reserved space mechanism.

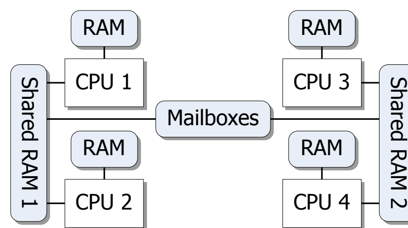

Figure 9: The experimental architecture

```
myapp : island;
leftpair : virtual processor;
rightpair : virtual processor;
myapp^cpus = [leftpair, rightpair]
myapp^manages = ["shareddataitem",
    "mythread1", "mythread2"];
leftpair^cpus = [cpu1, cpu2];
rightpair^cpus = [cpu3, cpu4];
leftpair^memory = sh_mem1;
rightpair^memory = sh_mem2;
```

Figure 10: Anvil ADL fragment showing application mapping to islands and virtual objects.

6. EVALUATION

This section first presents experimental results that show Anvil's implementation of IoC in action in section 6.1. Then, section 6.3 presents results from simulation to show the potential that the technique has in improving the efficiency of larger systems.

6.1 Experimental results

The experimental architecture (figure 9) is a four-core NUMA system implemented upon a Xilinx XC4VLX25 Virtex 4 FPGA [25] on the Xilinx ML401 prototyping board and uses the Microblaze soft-processor. The cores are arranged as two pairs, and each pair has its own bank of shared memory. The pairs can communicate using a dedicated mailbox.

Threads can migrate between all cores of the system to allow cores to be shut down to save power or for fault tolerance, but clearly when this happens it is preferable for data to move with their threads. If threads and data move to different pairs, access times are very slow. Without IoC and the architectural knowledge provided by CTV this requirement cannot be easily expressed in existing programming systems. Figure 10 shows an Anvil ADL fragment illustrating how this mapping is provided by the programmer.

A range of single and multi-threaded benchmarks were run on the target system. Each program was executed in three different modes and the resulting execution times are shown in figure 11. In this figure, all results are normalised against the *local* execution time, which is the time taken when running with its data in local shared memory (i.e. it has not migrated at all). *Migrated* shows the same program, but when one of its threads has been migrated to the other CPU pair and must fetch its data remotely. *Islands* shows the cost of executing the application under CTV's IoC. This includes the initial cost of migrating the application's data and then its subsequent execution time.

As can be seen from these results, our extended CTV allows threads to migrate throughout a heterogeneous architecture without risking incurring sizable performance penal-

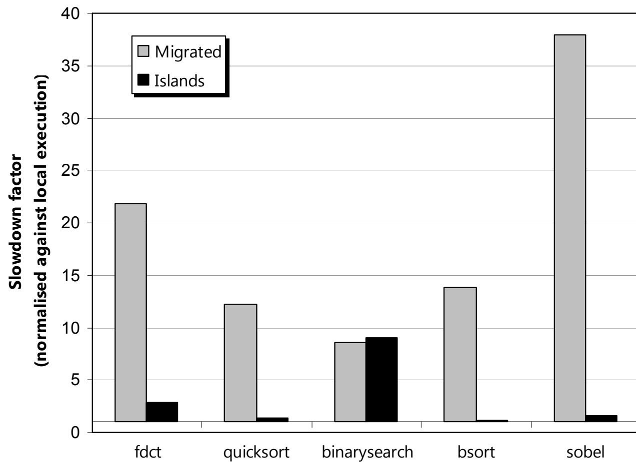

Figure 11: Resultant slowdowns experienced when threads execute with non-local data.

ties. Most programs complete in between 1.1 and 2.8 times the execution time, as compared to between 8.5 and 37.9 for systems that do not use IoC. Overhead in the IoC system is due to migrating the thread's data and migrating it back at a later date, which is largely unavoidable and must be paid in any such system.

One outlier is the binary search benchmark. This system performs slightly worse when using IoC than not. The reason for this is that the search only reads on average $\log n$ data items (where n is the input array length) yet migrating the entire array requires n accesses. Whilst individual random accesses are much slower than the bulk DMA transfer used in the IoC-based system, it is slightly faster overall. This shows that whilst island migration is better for most programs, care should be taken if the program only accesses a small number of random elements from a large data source. The converse of this is the Sobel filtering example. Because the Sobel algorithm frequents the same pixel values multiple times, the IoC-based system is vastly more efficient.

6.2 Cache coherency

Anvil's VP exposes to the programmer a single logical address space across the architecture which is designed to be compatible with C's memory model. CTV trades run-time variability for predictability and efficiency. By requiring that the number and location of shared data OMs is compile-time static, only the cores that actually use an item of data need to be considered for coherency. This is an improvement over more general systems, which have to assume the worst-case and keep all cores coherent.

Anvil implements a directory-based coherency scheme that makes use of the OM model. When a core writes to an item of shared data it also informs that item's OM that it has done so. The OM is aware of which cores access that data item from the IoC model and so it only needs to inform that subset about the change. The informed cores can then invalidate their cache lines accordingly. On the example system, this means that keeping a pair of CPUs coherent only requires 23 cycles after a write, as opposed to over 150 if the OM had to inform all CPUs of the system.

The directory size problem is solved, because OMs only need to store information about the shared data that is accessed by their island. Other items of shared data elsewhere in the system have no impact, and this can be verified offline using static analysis due to the static nature of CTV. The system allows the programmer to write as if the entire program is within a single address space, but does not place coherency demands on the hardware and run-time system that are not explicitly required by the application, ensuring a minimum amount of overhead.

6.3 Simulation results

In order to explore the IoC concept further, a simulator was developed that allows experimentation with large grid-based systems. The architecture simulated is a regular Manhattan grid of processing nodes, where each node has its own local memory. Each node is connected to its North, South, East and West neighbour via a dedicated mailbox for message passing. There is no global shared memory or caches. The simulation framework creates a random set of tasks and assigns them to a number of IoC in the system. It also creates a number of items of shared data and these are also distributed though the system. Each created task has an execution time and a memory access pattern that determines which items of shared data it needs and how frequently. Finally, each processing node of the system has a likelihood of being shut down, thereby triggering a migration of any tasks and data items that are on that node. The simulation runs in two modes. It can respect the IoC and migrate the entire island and all of its accessed data when one of its nodes is shut down, or it can not respect the islands and simply move the tasks that it has to. Memory access times and inter-node communication times are taken from real Microblaze-based FPGA systems. Undirected migrations are considered because standard programming languages do not provide grouping or mapping information that is required to use a more intelligent algorithm (without extensions or pragmas).

In this simulation the cost of the allocation algorithm that determines the migration target location is not considered. This is because we are interested in showing the benefit of IoC when used with future architectures with potentially hundreds of cores. For such architectures, the cost of migrating a single thread will be very similar to the cost of migrating an island of threads.

The purpose of the following experiments is to determine whether or not migrating the entire island (which has a higher initial cost than migrating a single task) results in shorter average memory access times over the execution of the program, and if so under what circumstances.

Varying grid size

Figures 12 and 13 show the effect of varying task execution time on grids of different sizes. In these experiments the probability of migrating tasks from a CPU is fixed one migration every 100,000 clock cycles. This is deliberately rather frequent - real systems are likely to migrate much less. As can be seen, on the smallest architecture (2x2) migrating the entire island appears on balance less efficient as the costs of migrations are considerably higher and do not give any real benefit due to the small size of the system and rapid migrations. However, on all larger architectures it does not take long before migrating whole island results in more efficient systems. The tests for the 4x4 architecture demonstrate clearly that for tasks with short execution times

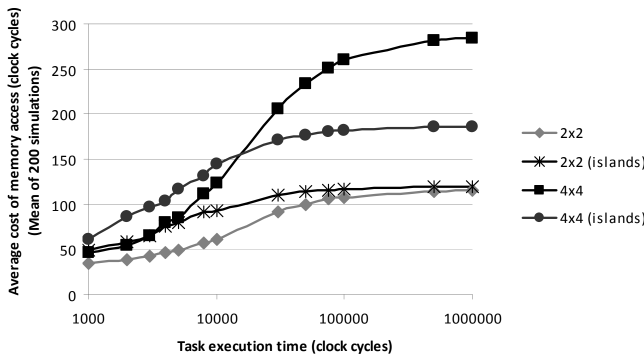

Figure 12: Varying task execution time with and without island migration on smaller architectures

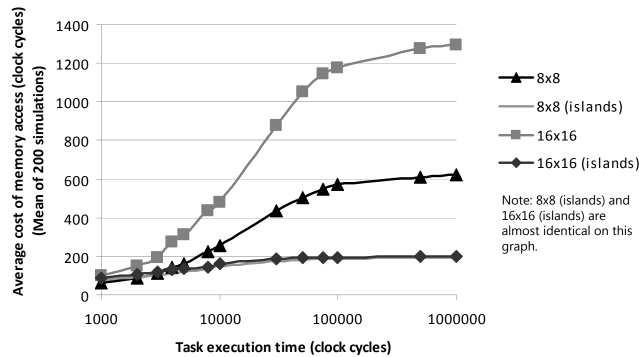

Figure 13: Varying task execution time with and without island migration on larger architectures

the tasks are only likely to be migrated a small number of times so the large transfer cost of migrating an entire island is not beneficial. Longer tasks however rapidly become less and less efficient as they migrate further from their data, as demonstrated most noticeably by the results for the 8x8 and 16x16 architectures. Also, when using IoC, the results for the 8x8 architecture are almost identical to the results for the 16x16 architecture due IoC's preservation of locality. This demonstrates the scalability of the IoC approach.

Varying task migration chance

Figure 14 shows the effect of varying the task migration chance. When migrations are incredibly frequent (around every 200 cycles) the cost of island migration is much higher than migrating single tasks, but for more realistic systems island migration results in systems that have a lower average memory access time by a factor of 3 in some cases.

Varying amount of shared data

Finally, figure 15 shows the effect of varying the frequency with which tasks access their shared data items. When island migration is not being used, this has no effect on overall system performance because the migration mechanism does not consider which data items are accessed by which tasks. As can be seen, if tasks only access their shared data infrequently (of a similar order of magnitude to the chance of migration) then migrating islands has no benefits and the average costs are higher. However, more frequent access results in much faster access times on average.

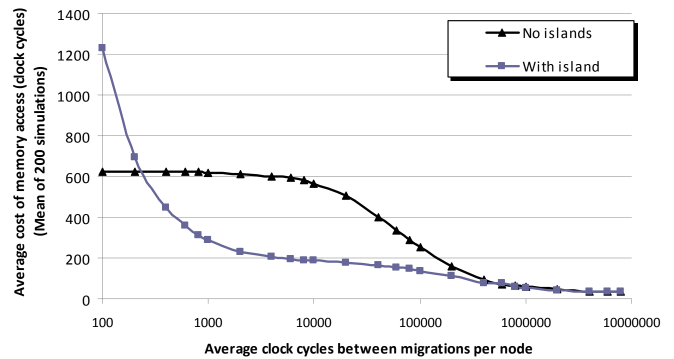

Figure 14: Varying task migration chance with and without island migration (8x8 grid)

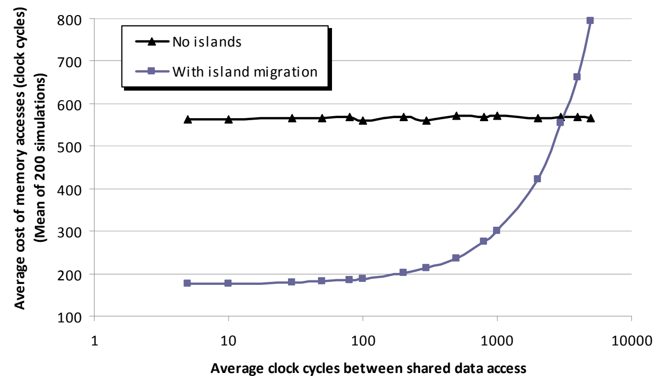

Figure 15: Varying how frequently tasks access shared data (8x8 grid)

Summary

These simulation results indicate that the IoC technique has the potential to greatly increase the efficiency of embedded systems that make use of migration. IoC becomes a particularly effective technique when:

- task execution time increases (as compared to the migration frequency),
- the architecture grows in size, or
- shared data is accessed more frequently.

If shared data is accessed very infrequently, migrations are incredibly common, or the architecture is very small the costs can outweigh any benefits, but it appears that on the vast majority of medium to large embedded systems with dynamism the technique can be used to deliver sizable performance benefits.

7. CONCLUSION

In this paper we have argued for the adoption of an Islands of Coherency model for modern embedded architectures that can allow for more efficient thread and data migration and reduce the pressure placed on cache coherency algorithms. To do this, we have taken a recent technique called Compile-Time Virtualisation (CTV) and extended its system model to allow it to work with dynamic systems.

The paper argues that the use of Islands of Coherency with CTV represents a paradigm shift from systems that are dynamic by default to systems that are presumed static, in which areas of dynamism must be explicitly enumerated. This leads to much greater predictability and aids analysis of non-functional properties.

The hierarchical IoC model is argued to be highly-scalable, and suitable for future embedded architectures which will contain greater numbers of CPUs. The model alleviates the need for system-wide cache coherency in favour of a programmer-directed approach that can better support multicore architectures. The compile-time nature of CTV reduces overheads to a minimum, in contrast to other run-time systems which are in general too heavyweight for use in embedded systems.

Anvil, an implementation of CTV, is augmented to support our extensions to CTV. Through experimental results we have shown that the Islands of Coherency model can have a large positive impact on the efficiency of complex embedded systems with dynamic thread and data migration.

8. REFERENCES

[1] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02*, pages 73–78, 2002.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 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] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In *Ada-Europe '98*, pages 263–275. Springer-Verlag, 1998.

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

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

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

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

[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] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 35–46, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

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

[15] 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.

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

[17] H. Huang, N. Yuan, W. Lin, G. Long, F. Song, L. Yu, Y. Liu, L. Liu, Y. Zhou, X. Ye, J. Zhang, D. Fan, and Z. Tang. Architecture supported synchronization-based cache coherence protocol for many-core processors. *Chinese Journal of Computers*, 8:1618–1630, 2009.

[18] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: A high integrity profile for Real-Time Java. In *In Joint ACM Java Grande/ISCOPE Conference*, pages 131–140. ACM Press, 2002.

[19] M. Loghi, M. Poncino, and L. Benini. Cache coherence tradeoffs in shared-memory mpsocs. *ACM Trans. Embed. Comput. Syst.*, 5(2):383–407, 2006.

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

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

[22] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, Volume 1:27–60, 1989.

[23] D. Stranneby. *Digital Signal Processing, DSP & Applications*. Newnes, Oxford, 2001.

[24] M. Ullmann, M. Huebner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 135–, April 2004.

[25] Xilinx Corporation. Virtex-4 user guide. *Xilinx User Guides*, UG070, 2007.

[26] C. A. Zeferino and A. A. Susin. SoCIN: A parametric and scalable network-on-chip. In *SBCCI '03*, page 169, Washington, DC, USA, 2003. IEEE Computer Society.