Application-Defined Virtualisation for Embedded Real-Time Software on Complex Architectures

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

N. C. Audsley
University of York
neil@cs.york.ac.uk

Abstract

In this paper we present a novel approach to embedded system development based on compile-time virtualisation. Whilst the target architecture may include multiple heterogeneous CPUs with non-uniform memory, the programmer is presented with an idealised view that supports the abstractions required by high-level languages like C. Unlike standard virtualisation, the programmer can influence the virtualised mappings to better tailor their system towards a target application. An example implementation is presented along with preliminary results.

1 Introduction

In order to meet escalating consumer demands for processing power and device functionality, the architectures of embedded systems are becoming increasingly complex. Many systems adopt application-dependent architectures that are optimised to perform a specific purpose. Such systems often contain multiple heterogenous processing elements [7, 2]; on-chip networks [5]; non-standard memory hierarchies with shared memory and new memory technologies [6, 1]; and unique features of the implementation fabric such as DSP cores and SIMD units [9, 10].

These new designs can complicate software development considerably because programming languages were developed under the assumption of a standard uniprocessor architecture that contains a single block of contiguous memory. By their very design, the abstractions developed for programming languages hide what have become important implementation details. The programmer is required to break the abstraction model using a number of techniques that are external to the source language. For example, architectures with non-contiguous blocks of memory require the use of custom linker scripts and compiler pragmas. These techniques are architecture-specific, time-consuming and error-prone.

In an attempt to solve this, much recent work has considered the development of new languages or language extensions that use certain features of these more complex architectures. However, such an approach is not a long-term solution as new hardware techniques are constantly being developed, and it is not feasible to create a new language each time. Also, the complexity of language and compiler development is considerable and such an approach fragments the development community, rendering obsolete existing tools for verification and validation.

Some example of domain-specific extensions are OpenMP [4] and Chapel [3] which attempt to integrate parallelism into the language to simplify parallel programming; Sequoia [6], which tries to simplify NUMA programming; and Streams-C [8] and StreamIt [11] which add concepts to C for describing stream paths and computational kernels.

This paper argues that the overall approach taken by the above languages can be improved by adopting a system with more suitable abstraction models. For example, code executes on a CPU but it does not describe the system the CPU is situated in. A language like (standard) C does not have a suitable abstraction model to reason about its execution model or target architecture – the execution model is constant; the target architecture can only accessed via the rather limited abstraction of memory mapping.

Within this paper we explore relaxing these fundamental language assumptions – although the programmer does not want complete control they do want to be able to *influence* the compilation to better suit their design.

2 Compile-time virtualisation

This paper describes a different approach that attempts to insulate the programmer from the complexities of the underlying hardware by using virtualisation. The approach presents the programmer with a standard Von Neumann-style architecture with a uniform memory map that is suitable for standard languages, such as C or Ada. Both automatic and designer-guided translations are provided that map between programming language elements and architectural elements. Normal virtualisation has a fixed mapping between the virtual environment and the implementation environment. The proposed technique allows for this mapping to be influenced by the designer so that custom architectures can be effectively exploited without breaking the abstraction rules of the source language. In order to maintain efficiency and avoid run-time overheads the virtualisation is applied at compile-time.

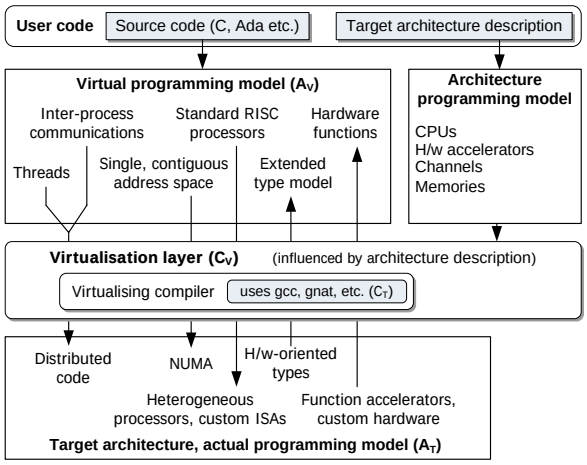

Figure 1. System overview

The proposed virtualisation layer (VL) is bidirectional and allows for more efficient use of custom hardware by exporting architectural constructs up to the software level. For example, hardware accelerators and other unique elements like on-clip multipliers or clock managers can be presented to the programmer as first-class parts of the source language, effectively extending the source language in a controlled way. Without this extension the system is still beneficial, but only features expressible by the source language can be mapped to the target hardware.

The main advantage of virtualisation is that it presents the designer with a much simpler programming model. It helps to maintain appropriate abstraction models by moving architectural concerns into the mappings of the VL and away from the source code. It also greatly facilitates code reuse and allows designers to explore different hardware architectures without affecting the software.

2.1 System overview

In normal run-time virtualisation, a layer sits between executing code and the true implementation architecture, A_t. The VL presents to the running code an idealised view of the architecture, A_v. A_v has a different set of abstractions which allow for a simpler or more powerful programming model. The true complexity of the hardware is hidden. At run-time, the executing code (which is written assuming A_v) is translated into operations that are appropriate for A_t. This translation process incurs a considerable cost, so to avoid this the presented system instead uses compile-time virtualisation. Compile-time virtualisation sits between the application source code and its compiler rather than between executing code and the hardware. It is analogous to run-time virtualisation; it presents the code with a virtual compiler C_v, which translates source code into a form which is suitable for the true compiler C_t.

In this system, C_t is a pre-existing compiler such as `gcc` or `GNAT`. A compiler and its input language have a set of implicit assumptions about the hardware upon which their output will run. It is these assumptions which cause problems in embedded systems development because they are often incorrect, forcing programmers to use the techniques mentioned in section 1. In effect, C_t only targets one architecture whereas C_v can target a range of architectures by allowing the programmer to tweak the translations employed within the VL. It uses C_t to perform the bulk of the compilation, whilst ensuring the output code is suitable for A_t.

As well as removing runtime overheads this approach allows existing compilers and toolchains to be used, something which is not the case with most language extensions. It is also worth noting that this technique is language- and implementation-independent. Nothing is assumed about the starting language or the final implementation, only the mappings within the VL change, as described in the following section.

2.2 Mapping to the target architecture

In order to be able to map language elements to architectural elements correctly, the VL must be aware of the high-level layout of the target architecture. It needs to know the number of CPUs in the system, the memory layout of each CPU and the manner in which CPUs can communicate. This information can be provided by the programmer (as in our implementation described in section 3) or inferred from the source code itself.

It is through this information that the designer can influence the VL mappings and therefore the implementation of their design. A system with two threads may be implemented with two CPUs with a thread each, or one CPU that executes both. This sort of decision is outside the scope of normal programming languages so either is equally correct. However, the two solutions have very different non-functional properties that the designer must control. Previous approaches have failed to give the designer this ability.

2.3 Required abstractions

In order to develop the virtualisation layer concept we have developed an initial list of abstractions that must be supported by the virtualisation layer in order for software to execute correctly and efficiently.

Uniform memory architecture All memory is in a contiguous block with a single address space, accessible from all threads. Memory access times are independent of address and within the same order of magnitude.

Consistent concurrency controls Inter-process concurrency control is possible, i.e. mutexes or condition variables.

Standard ISAs The capabilities of each CPU in the system are equivalent, and executing on one CPU is the same as executing on another. No custom accelerators, except for the floating-point unit.

For many embedded architectures these do not hold. For example, a programmer can write C code (which assumes a contiguous memory space), but it will not work on an architecture with a non-uniform memory map. The virtualisation layer must therefore provide a translation between the assumptions of the input language and the actual implementation hardware. Given the list above we have developed an initial implementation of a virtualisation layer than allows the execution of multithreaded C code on multi-core platforms. This is described in the following section.

3 Anvil

To test the efficacy of the virtualisation layer concept, we are currently developing an example virtualisation system named *Anvil*. Anvil allows the targeting of C code to FPGA-based NUMA multicore architectures. C is not a concurrent language, so in order to provide multiple threads of control use of the `pthreads` library is assumed. `pthreads` was selected because it is commonly used, part of the POSIX standard, and has been extensively studied.

Anvil's C_t is `gcc`, and the C_v it provides accepts C code to distribute code over a custom FPGA-based system. Rather than just targeting the source code, it also leverages the architectural information that it has been given to generate VHDL for the entire system, ready for FPGA place and route. Therefore, Anvil also includes features of high-level synthesis languages. Anvil works by reading a system-level description of the target architecture and refactoring the programmers C source code so that it can be used on the described hardware. Threads are separated into separate source files, one for each target CPU. Inter-thread communications are recoded to use the communication media of the architecture. The single address space abstraction is layered over the code so that NUMAs can be transparently accessed by the programmer.

Unless specified otherwise, Anvil will default to the same implementation that C expects; a single CPU system with a single block of contiguous memory. However, the designer can provide a simple system description along with the C source code that influences the virtualised mapping and produces a different architecture. This system description contains processor, memory, channel, variable. Processors and memories declare CPUs and RAM respectively, channels provide inter-CPU communications and variables declare globally-shared data. Attribute assignments are used to assign memory and threads to processors. Example Anvil source to describe this is shown below. A full description of the Anvil language is outside the scope of this paper.

```
CPU1, CPU2 : processor Microblaze;
shared, mem1, mem2 : memory BlockRAM;
CPU1^memory = mem1; CPU2^memory = mem2;
CPU1^extramem = shared;
CPU2^extramem = shared;
globalvar : variable;
globalvar^memory = sharedmem;
```

This sets up two CPUs with memory and declares a block of shared memory that can be accessed by both. `globalvar` is a global variable in the C source code, and is assigned to the `sharedmem` memory. Type information is not required as it can be taken directly from the source.

3.1 Implementation details

The first stage of implementation maps threads and variables onto CPUs and memory. Anvil performs a static analysis of the source code to determine the number of pthreads in the system. This places a restriction on the threading models used that they must be statically analysable, which is a common restriction in embedded systems development. If non-analysable behaviour is required, it must be confined to operate within a single CPU. Once this information is obtained, Anvil generates a C source file for each CPU in the system. The CPU that is assigned the initial main thread gets a copy of the original `main()` function. All other source files get a generated `main()` function which simply waits until its thread is started and then calls the thread body function. Each source file also receives a copy of all data structures, typedefs, and library functions that the thread requires.

Global variables must then be assigned. The scope rules of C require that all variables shared between threads are in global scope. Anvil uses this knowledge and can safely ignore local variables in the mapping phase, assuming that they will be placed in memory local to the thread's CPU. The declarations for global variables are placed in the source file of the thread they are assigned to. This assignment is currently provided by the designer but analysis could assist with this in the future.

The second stage of architectural mapping involves applying the SAS and concurrency control abstractions. We have developed and implemented a technique based on *object managers* which assigns the control of shared data and concurrency primitives to threads throughout the system in a scalable and efficient manner. Full details are outside the scope of a short paper and will be presented soon. Once the object managers are assigned, library functions for inter-thread communication written. Then, all pthreads calls are modified to use a specially developed 'Anvil pthreads' library which implements the object manager system. Finally, global variable accesses are wrapped by calls to fetch and update the values of the variable from its manager. The result of this phase is that the code of each thread now calls specially-written communication routines for access to all external data and for concurrency control.

The final implementation stage involves generating support files for `gcc` and the `ld` linker to ensure that code and data sections are correctly placed in the memory map of each CPU. Once this stage is complete, each source file

can be compiled and executable object code is produced for each CPU in the system.

3.2 Preliminary results

The following section describes the results of preliminary testing of Anvil. For this test, we created a standard C implementation of Triple-DES encryption that pseudo-randomly generates fifty blocks of data and encrypts them. We also created a two-threaded pthreads version in which one thread encrypts even-numbered blocks and the other encrypts odd-numbered blocks. In the pthreads version, mutexes are used to ensure data is fetched and saved under mutual exclusion and the master thread joins with the worker at the end to complete execution. The single-threaded version was implemented on a Microblaze soft-core processor on a Xilinx Spartan 3e using a single block of RAM for code and data storage. The multithreaded version was implemented on two Microblazes, each with its own block of memory, and a simple UART for inter-CPU communications. The two-thread Anvil architecture definition was as follows:

```
CPU1, CPU2 : processor Microblaze(size);
mem1, mem2 : memory BlockRAM(1024);
CPU1^memory = mem1; CPU2^memory = mem2;
comms : channel uart(921600, 8, 0);
CPU1^channels = [comms];
CPU2^channels = [comms];
CPU1^threads = ["main"];
CPU2^threads = ["thread1"];
```

This sets up the hardware and allocates each thread of the system to a CPU. From this information the hardware can be built, although Anvil does not yet do this automatically. The execution speed of our DES implementation is data-invariant and the single-threaded version completes in 34,363,789 clock cycles. The two-threaded version completes in 17,442,119 clock cycles, 50.76% of the execution time. Therefore, this program displays only 0.76% overhead compared to the theoretical optimal. However, this program was very easy to parallelise and the time spent holding mutexes was kept artificially small, leading to such impressive execution speed improvements. In actual programs such improvements are less likely, but the low system overheads will remain, and it is this point which is important. Anvil cannot make programs more parallelisable, but it does not impose a large operational overhead, allowing the designer to get the best from their code. Further tests are currently underway to ensure such results are repeatable.

4 Conclusion

In conclusion, we have presented a novel approach to embedded systems development that uses compile-time virtualisation to insulate the programmer from the details of the underlying hardware. The virtualisation layer provides mappings between language constructs and architectural constructs that can be influenced by the programmer to tailor their code to a range of implementation architectures. The approach is entirely language-independent and due to the use of compile-time virtualisation overheads are very small. A large advantage of this approach is that existing compilers and toolchains can be used. Expert knowledge is not required but it is assumed that the designer wants control of the mapping process. Therefore some knowledge of embedded systems will help produce a more efficient system. Our work does not automatically map software to hardware, perform parallelisation, or find optimal scheduling schemes because these problems are orthogonal and being studied elsewhere. The results from such work can be easily combined with our findings.

We have created an example implementation of the virtualisation layer concept called Anvil. Anvil operates from C source code that uses the pthreads library and targets FPGA-based heterogenous multi-core systems. Initial tests have shown the Anvil system produces correctly distributed systems with a very small amount of operational overhead. We expect a complete prototype of Anvil soon which will allow us to perform more in-depth evaluation.

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] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore SoCs. *DAC*, 00:789, 2002.

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

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[5] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. *DAC*, 00:684–689, 2001.

[6] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06*, page 83, 2006.

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

[8] 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.

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

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

[11] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Ho, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications, December 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.