The University of York
Department of Computer Science

Architectural and scalability issues in hardware synthesis of high-level languages

Qualifying Dissertation

Ian Gray
iang@cs.york.ac.uk

5th September 2007

Abstract

The proliferation of embedded systems over the last decade has increased the demands on system designers to create devices that are powerful and feature-rich whilst remaining reliable, cheap and energy-efficient. New technologies such as FPGAs and high-density ASICs provide suitable implementation fabrics, but designer productivity has not increased accordingly and almost all designs are still created using low-level hardware description languages. To mitigate this problem, hardware synthesis systems have been developed that attempt to translate high-level software languages like C or Ada into an implementable hardware description. Whilst the resulting increase in abstraction allows for more rapid development, current synthesis techniques do not scale to allow the efficient creation of large designs. The designer lacks control over the implementation architecture and cannot influence the synthesis process. As a result, the use of high-level synthesis is currently limited to prototyping or to the specification of small subcomponents.

This document examines the use of FPGAs as an implementation platform and how they are related to the system-on-chip and network-on-chip design methodologies. It then discusses the current state of hardware synthesis and examines the problems that are encountered when using it. Techniques from the software domain are then detailed that may help to resolve some of these issues. Reflection, aspect-oriented programming, and metaobject protocols are considered.

Contents

Chapter 1

Introduction

Embedded systems are application-specific computer systems that are deployed as part of a larger machine. Unlike general-purpose computers they typically perform only a small set of tasks that cannot be changed. Also, they are often transparently integrated into their host systems so that the user is not aware of their presence. Embedded systems are used in many different domains. Consumer electronics is one of the commonest markets and they can found in common household items such as mobile phones, microwaves, televisions and digital watches. The automotive industry also makes extensive use of these systems to create engine management or system diagnostic modules. They are even used in high-integrity systems, such as aeroplanes, factories, medical devices and power stations. In general, almost all digital devices contain some kind of embedded computer system.

Due to their intended applications, embedded systems often have a larger set of design requirements than general-purpose computer systems. Their physical size limits many factors such as processing power or the amount of available memory, meaning that efficiency is very important. Similarly, embedded applications are often battery-powered so they must limit the amount of energy they consume. As they must interact with the outside world they are commonly subject to strict timing requirements, leading to many embedded systems being classified as *hard real-time* systems. [9] Designing such systems has always been a challenge, but it is becoming ever more difficult as embedded technology enters more aspects of daily life and systems become much larger and more complex.

Initially, the substantial cost associated with digital technology meant that computer systems would only be used in situations that actively required computer control and would be otherwise impossible if attempted by a human operator. One such example is precision engineering, where a human would be incapable of the fine motor skills required and so computer-controlled actuators are used. However as technology progressed and the cost of microprocessors decreased, embedded systems were integrated into more and more devices where computer control was not essential but instead simply added new features or improved existing ones. Televisions, washing machines, telephones and cars are all examples of this trend. As silicon technology advanced further, processors became so cheap that they are now commonly used to actually reduce the cost of a system. Often a large number of integrated processing elements can be replaced by a single embedded CPU with no loss in computational power, thereby reducing the overall build cost of the system. Indeed, the pervasiveness of embedded systems is so great that of the nearly 8.3 billion microprocessing units shipped in the year 2000, 8.14 billion (98%) were used in embedded applications. [55]

Today, consumer expectation is increasing on a daily basis and the amount of functionality that is embodied by embedded systems is constantly growing. Embedded systems are required to become more powerful in order to meet these increasing demands whilst still maintaining their real-time properties to ensure that reliability and safety are not compromised. This makes the development of embedded hardware and software more difficult and error-prone, thereby increasing its cost and time-to-market.

The problem examined by this dissertation is that this increase in design size and complexity has not been met by a corresponding increase in designer productivity. Apart from in a few specific domains, designers must still use low-level circuit description tools to manually create their systems. This is time-consuming, error-prone, and requires expertise and experience. In an attempt to mitigate these problems, high-level synthesis systems have been developed that allow hardware design to take place at a higher level of abstraction. Unfortunately, no current system is powerful enough to have been adopted by mainstream designers. Current synthesis techniques impose restrictions on the designer that do not scale when they are used to describe entire systems, resulting in implementations that are either inefficient or do not meet the non-functional constraints of the application. In this document, the problem is examined further and potential solution areas are examined.

Chapter 2

Literature Review

This section discusses the role of embedded systems and how they have developed over recent times, evolving into Systems-on-Chip (2.1.1) and Networks-on-Chip (2.1.2). It then discusses FPGAs (2.2) and a number of techniques that are associated with the use of them. The review then moves to concentrate on the ways in which we can describe hardware designs with a software program, looking at both Hardware Description Languages (2.3.1) and the hardware synthesis of high-level programming languages (2.3.3). It then briefly discusses the topic of hardware/software codesign (2.4) before moving on to metaprogramming techniques. The main techniques mentioned are reflection (2.5), metaobject protocols (2.6), and aspect-oriented programming (2.7.1).

2.1 Design of embedded systems

At the time when the first embedded systems were being developed, the available fabrication technologies meant that only relatively small circuits could be built. Most designs were presented in the form of a schematic diagram showing the circuit as a design composed of logic gates or transistors. Software was not a concern because something as complex as a microprocessor could not be manufactured in a way that was small or cheap enough for its inclusion to be considered. Designing with schematics was advantageous as it introduced no inefficiency and no further tools were required to convert the design to a useable format for implementation in actual hardware. However, the lack of abstraction meant that such schematic diagrams were very difficult to work with when design sizes began to increase.

Since the development of the integrated circuit in the early 1960s and consequently the microprocessor in around 1970, the logic density and effective computational power of digital circuits has increased massively. Fuelled by this, the power and complexity of embedded systems have increased similarly, and schematic-level design is now no longer a viable option for most designers as it is too cumbersome to work with. To remedy this, Hardware Description Languages (HDLs) were developed[1] that

[1] In 1983 ABEL [1] was devised for targeting programmable logic devices but it was not until 1985 that Verilog, the first modern HDL, was developed. VHDL followed in 1987.

provide a higher level of abstraction and allow for large amounts of circuitry to be designed using source code similar to a traditional programming language. (HDLs are described more fully in section 2.3.1.)

2.1.1 The System-on-Chip paradigm

Previously, electronic devices were constructed using processing elements that were each contained within their own separate integrated circuits (ICs). These ICs were packaged in ceramic chips and communicated with each other over the routing on a printed circuit board to which they were all mounted. However as the complexity of embedded systems has continued to increase, it is now common to see ICs that embody an entire system rather than a single component. Rather than implement a single function, the ICs can now contain multiple logic cores all working in parallel to solve a single goal with little external routing required. This has become known as the concept of System-on-Chip (SoC), where a single device contains a number of logical modules which interact to solve a given task. SoC architectures have arisen because as silicon chip fabrication becomes more sophisticated, the circuits that can be imprinted upon them are faster, smaller and cheaper than an equivalent printed circuit board design. ICs often use less power and therefore generate less heat than printed circuit boards [59] which can be advantageous for devices that are built to operate on battery power. Finally, a SoC can be easier to design than the equivalent PCB-based design as the designer does not have to consider track layout, off-chip propagation delay or other extra-logical characteristics.

A particular challenge that arises from the SoC paradigm concerns the method of communication between the individual IP cores that the system is comprised of. This is commonly achieved by implementing ad-hoc bus architectures or by routing lengthy interconnecting wires across the silicon die. Whilst sufficient for the early SoCs, such designs do not scale well because of their reliance on a globally synchronised clock. Such a clock is difficult to support in a large system because of propagation delay and routing requirements [3]. Bus-based systems can also require a large amount of power and generate a lot of heat as driving a signal onto a bus involves asserting a large number of (often rather lengthy) communication lines.

2.1.2 From System-on-Chip to Network-on-Chip

To mitigate these problems, many newer SoC designs are moving away from buses and employing on-chip networks instead. As discussed by Benini and De Micheli [3]:

> "The most likely synchronisation paradigm for future chips is 'globally-asynchronous locally synchronous' (GALS), with many different clocks. Global wires will span multiple clock domains, and synchronisation failures in communicating between different clock domains will be rare but unavoidable events."

This describes a micro-network in which different modules of the SoC each form a communicating node on the network. The crucial difference in the Network on Chip

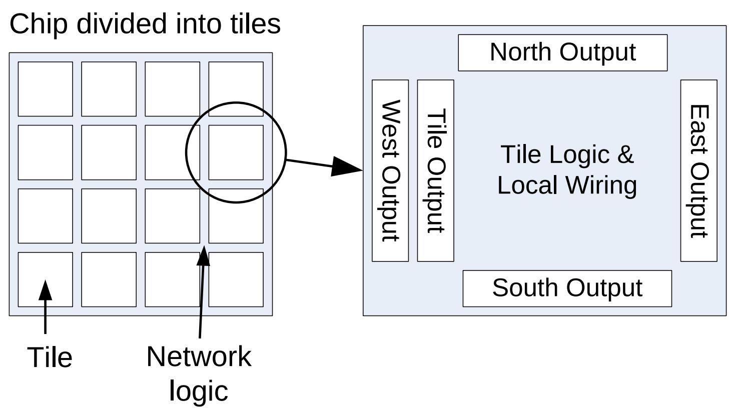

Figure 2.1: The Dally and Towles Network Architecture. The system is divided into a 2D grid of tiles with interconnection logic between the tiles to handle communication. [14]

(NoC) concept is that each node is a separate timing domain and data transfer between nodes is performed asynchronously. The presence of multiple clock signals affords designers a great deal of freedom and allows a number of the previously mentioned limitations to be overcome. Primarily, the use of a network reduces the problem of propagation delays between modules in the system by removing dependence on a global clock.

Also, as different sections of the SoC can be driven by different clocks, the critical timing paths of one section do not affect the paths of any other. This allows for parts of the system to be clocked at a very high rate, even if there are some sections that must be clocked slowly. Conversely, sections that do not require a high clock speed can be driven slower to use less power and generate less heat. Most NoC designs are based on a tiles, and one of the first such architectures can be seen in figure 2.1.

Tile-based networks are often proposed for NoC implementations because, as noted by Dally and Towles [14], unlike standard shared-medium networks such as Ethernet the available bandwidth of the network actually increases with the number of connected nodes. They are also more amenable to techniques that add fault-tolerance or adaptive load balancing due to the large number of available routes between sender and receiver.

Current work in the NoC field is still primarily concerned with exploring potential architectures or protocols that are likely to be of use in this emerging design paradigm. Zeferino and Susin [68] present a parametric architecture that can be adapted depending on the required network performance or available implementation area. More powerful networks require a larger amount of interconnect and logic to implement but they can support applications with higher bandwidth requirements. Wiklund and Liu [58] proposes an architecture that is designed to support hard real-time systems with strict bandwidth requirements. Rather than use a best-effort protocol to transmit a packet as soon as possible, as in Ethernet, their proposed network establishes virtual circuits based on an offline schedule to ensure communications will complete before deadline. Kumar et al. [40] concentrate more on protocol development and adapt the OSI model to create a four level communications stack that aids higher-level development. Pande et al. [49] considers the design of a network switch that is particularly suited to NoC-style networks. They use wormhole routing in their network and produce a design that is expected to consume approximately 2% of a large NoC-based design.

The main weakness in much of the above work is that it is difficult to evaluate their effectiveness in real-world situations, so theoretical examples are used instead. As NoC is only just beginning to emerge as a valid design style, there are few existing systems that are suitable for use as case studies or comparisons. As more and more commercial systems are developed that use on-chip networks and the research field matures, it will become clearer what the actual requirements of a commercial NoC are and so therefore which NoC strategies show the most promise.

2.2 Field-Programmable Gate Arrays (FPGAs)

An FPGA is an example of a class of programmable logic devices known as gate arrays. In a gate array architecture, transistors, logic gates, and other active devices are placed in a regular lattice pattern and connected with interconnect wires. These wires are configurable and can be arranged to connect the resources of the device in a structured manner. By placing the interconnect lines correctly, a process known as routing, the components on the device can be connected to form almost any desired circuit.

FPGAs were initially developed in 1984 by Xilinx and were marketed as an alternative way of evaluating ASIC designs. Previously, evaluating a designed circuit required that either the design was manually built from connecting discrete components, or it was fabricated as a custom-built ASIC. Both methods were time consuming and very costly. FPGAs changed this by giving the designer an implementation fabric onto which designs could be programmed. Evaluation could begin almost immediately, and once errors were found and corrected the device could be reused. This prototyping method drastically increased the efficiency of ASIC design, but also opened up new possible implementation methods. As the size and speed of FPGAs increased and their unit costs decreased, more and more embedded systems were developed that included an FPGA in the final circuit design, rather than an ASIC. This avoided the heavy set-up costs associated with creating a custom IC and is very suitable for products that are sold in low to medium volumes.

On a basic FPGA, the primary resources are Configurable Logic Blocks (CLBs), interconnect and input/output blocks (IOBs). (See figure 2.2) CLBs make up the majority of the components on the FPGA[2] and are used to create sections of logic that implement the primary functionality of the device. They are constructed from Look-Up Tables (LUTs) and flip-flops and can be programmed to perform one of a large set of logical functions on their inputs. CLBs are connected to each other by programmable interconnect which can be configured to selectively route signals across the FPGAs. In all modern FPGA architectures interconnect follows a hierarchical model. The majority of interconnect is named ‘local interconnect’ and is constructed from short wires that may only span a small number of CLBs. This is most commonly

[2]It is the vast amount of interconnect that actually consumes the majority of the silicon area of an FPGA [7]

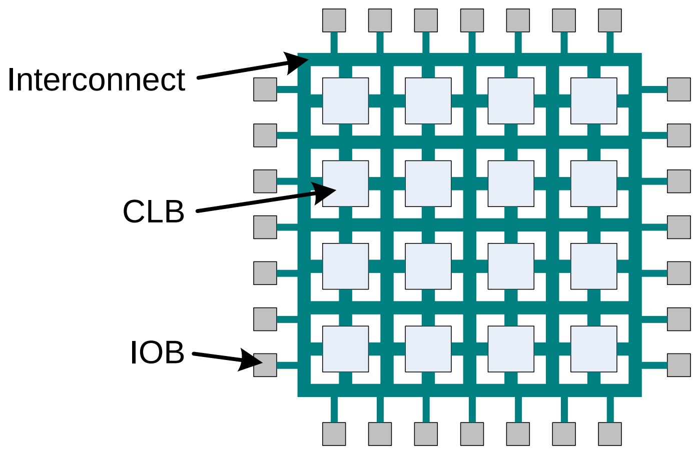

Figure 2.2: Simplified FPGA architecture showing CLBs surrounded by interconnect and interacting with the outside world though IOBs

used to connect the inputs and outputs of adjacent CLBs to form a single large logic function, such as a multiplier or shift register. In order to connect distant parts of the FPGA, 'global interconnect' exists which is comprised of longer wires that may span the entire width of the FPGA. Due to area constraints, global interconnect is much less common than local interconnect and so can often be a very limited resource. Finally, global clock nets are a special type of global interconnect which exist solely to propagate clock signals throughout the FPGA. Due to the complex hardware involved in reducing clock skew, there are generally only a few clock nets available on a device, 4 on the Xilinx Spartan-IIe [64].

2.2.1 Advanced FPGA architectures

Whilst all FPGAs are comprised of LUTs and interconnect, modern FPGAs contain a number of additional embedded modules for performing specialised tasks. These allow for greater design flexibility as they operate at a high speed and can be used to implement functions that would take up a great deal of the normal FPGA fabric. These modules are often connected via switching blocks to the global interconnect lines and can therefore be driven by any other part of the FPGA. Some of the more common additional modules are described here.

For performing complex control operations or processing large amounts of data, many high-end FPGAs include processor cores as part of their architecture. For example, the Xilinx Virtex-4 [65] contains four PowerPC 405 cores. These cores operate independently of each other and can be driven from different clock signals if required. The surrounding logic presents the cores with information and collects the results once processing is complete. Due to the fact that these cores are implemented as embedded ASICs rather than derived from the normal FPGA fabric, they can be clocked at much higher rates than most FPGA-based processor implementations.

Most applications require some form of memory to store programs or data. Whilst

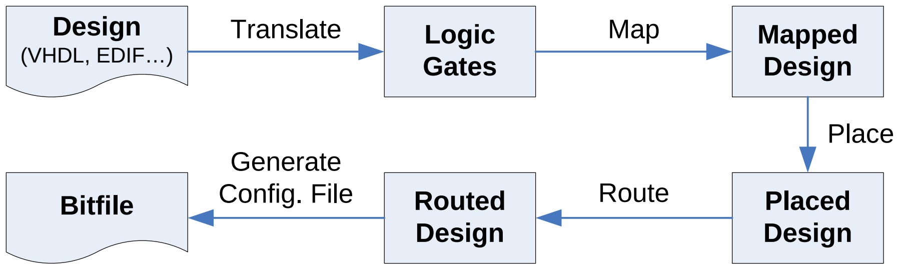

Figure 2.3: The standard FPGA design flow.

RAM can be synthesised by combining LUTs appropriately, this is a very inefficient use of resources. As a result, distributed RAM is a common feature in modern FPGAs. In Xilinx FPGAs, the distributed RAM modules are called BlockRAMs and on the largest Vertex-5 FPGA [66] there is over 11 megabytes of storage available to the designer. Different FPGA families have different RAM layouts, with some favouring a large number of smaller RAMs and others concentrating most of their memory in a few large RAM banks. The initial contents of Xilinx BlockRAMs can be set from the configuration bitstream when the device is programmed and each BlockRAM can be arranged to have different widths and depths. For example it could be accessed as n different 16-bit values, or $2n$ different 8-bit values. All Xilinx BlockRAMs are dual-ported, allowing for two processes to access the RAM simultaneously, but this is not the case for all FPGAs. The memory blocks on high-end Altera FPGAs can be configured to operate in single-port mode, or to relinquish some storage capacity to add extra access ports as required by the application.

In addition to embedded softcores and distributed RAM, many other design elements are being integrated into FPGA fabrics. Some devices include dedicated multiplier units that can perform calculations much faster than similar logic synthesised from the CLBs, clock management circuits distributed across the FPGA help to manage clock skew and create stable clock dividers and high-speed I/O modules such as the Xilinx RocketI/O modules allow off-chip communication at 11 Gbit per second. The use of these special design elements is normally controlled by the vendor's synthesis tools which attempt to infer when one of them may be of use to a design. For example if the synthesis tools create a large enough multiplier, it may choose to use an on-chip one rather than synthesise one directly from CLBs. However for many design elements such inference cannot be used and elements like RocketI/O transceivers or embedded processing cores must be manually instantiated using a hardware description language like VHDL.

2.2.2 The FPGA design process

FPGAs store their current configuration in specially reserved configuration memory. As this memory is volatile, the device must be reconfigured each time it is powered up. In the situations when the device has not been loaded with a set of configuration data, known as a bitfile, all user-accessible pins default to a high impedance state and the device remains idle. The process of creating a bitfile to program an FPGA with is described here and shown in figure 2.3.

1. **Design** - The required design is expressed in a form that is acceptable to the FPGA design tools. This may be a schematic, an EDIF netlist, source code written in a hardware description language (section 2.3.1), or source code written in a higher-level language that can, though the process of synthesis, be converted to a hardware description language (section 2.3.1).

2. **Translate** - The user input is translated to logic gates, essentially converting all forms of input to the schematic form.

3. **Map** - The resulting logic gates are mapped into CLBs and other atomic elements of the target FPGA fabric.

4. **Implement** - The design is implemented on the target FPGA. This stage, consists of two main operations:

    - **Place** - The mapped CLBs are placed onto the device. This is the first point at which a design can be determined to be too large for the target FPGA. The mapping algorithm will attempt to keep logically related CLBs together to minimise routing. The placement algorithm is a version of the bin packing problem and is NP-complete. [67] As a result this stage can take a long time to execute and at times of high utilisation the placer must resort to a simple exhaustive search.
    - **Route** - The interconnect between the placed CLBs is finalised. The routing algorithm attempts to use the shortest interconnection lines possible to reduce propagation delay and power consumption. Due to the large amount of interconnection required by most designs, it is possible for a design to fit onto a device at the placement stage but for routing to be impossible due to lack of space. Again, this problem is NP-complete.

5. **Bitfile creation** - The placed and routed design is converted to a bitfile that can be used to configure the target FPGA. The final bitfile can only be applied to the exact FPGA model for which it was created, even different FPGAs from the same family cannot share bitfiles.

Once this process is complete, the FPGA can be configured by loading the created bitfile into its configuration memory. Different FPGAs provide different programming interfaces but all modern FPGAs implement at least two main modes, a serial interface and a JTAG interface.

The serial programming interface is simple and is used to program the device from an attached memory chip very quickly. As previously mentioned, FPGA configuration memory is volatile and so the device must be reconfigured every time it is powered. As a result, if an FPGA is used in a complete system the configuration bitfile must be stored in some form of non-volatile memory, such as a PROM. The PROM can be connected to the FPGA via the serial interface and configured to transfer the stored bitfile to the FPGA very quickly. Most FPGAs, including the Xilinx Spartan and Virtex series, can automatically drive a connected memory chip to retrieve the configuration bitfile without additional logic. Whereas the serial interface only allows for bitfile transfer, the JTAG port [31] is a much more complex interface that can also be used for readback and debugging. JTAG is standardised as IEEE 1149.1 and is used in many embedded systems to assist with debugging and maintenance throughout the design and lifespan of the system.

2.2.3 Partial Dynamic Reconfiguration

A major benefit of an FPGA-based implementation fabric is the ability to reconfigure the device at runtime. Most systems that include an FPGA component simply configure it once when power is first supplied and the running design remains the same until the power is removed and the device shuts down. However, the configuration engines of modern FPGAs are much more flexible than this and they allow a running device to be stopped and reconfigured with a different bitfile, effective turning it into a different circuit that implements a different set of functionality. This gives rise to entirely new application areas and allows for an FPGA design which can respond to a mode change or other significant event in the system by changing its behaviour drastically. Effectively, this dynamic reconfiguration allows for a number of mutually exclusive features to be implemented on the same hardware at different times, thereby reducing silicon area and power consumption.

A good example application of this ability is a video decoder system that supports many different video formats. The system implements its decoding functions in hardware as this is much faster than a software-based solution. However, a dedicated ASIC is required to decode each different video format that it supports. If an FPGA is used instead, then different configurations for the various formats can be stored in a single ROM chip and then programmed in just before decoding commences. This allows for the system to display the speed of a hardware-based implementation but to support many different formats without dedicated hardware for each one. Also, firmware updates can be issued to add support for new formats after the device has been shipped.

Recent FPGAs also allow for a more fine-grained reconfiguration mechanism. Rather than reconfiguring the entire device as described above, it is possible to load a partial bitfile which only affects a subset of the device whilst the unaffected areas continue to run. This technique is known as *partial* dynamic reconfiguration [3] (PDR) and was introduced in the mid 1990s with the Xilinx 6200 FPGA[13]. Not all FPGAs support PDR and those that do display differing capabilities depending on their architecture. For example, the Xilinx Virtex-II must be reconfigured in whole columns whereas the Virtex-4 relaxes this constraint and allows for individual tiles from a grid to be independently reconfigured.

2.2.4 Applications of Partial Dynamic Reconfiguration

PDR has many theoretical uses that are related to the concept of 'virtual hardware'. By analogy to the concept of virtual memory, physical hardware could be made to appear larger than it actually is by swapping circuitry in and out of the FPGA from dynamic memory or magnetic storage (see figure 2.4). When a new feature is required by the system it is swapped in to the FPGA using PDR and activated. Later, when it is no longer needed it can be removed to free up space and reduce power consumption. Both of these operations need not affect the rest of the system and the parts of the FPGA that are unaffected by the reconfiguration remain active at all times.

[3] Different authors tend to use slightly different terms when describing this concept; other common terms are 'partial reconfiguration' and 'active partial reconfiguration'. This document will use 'partial dynamic reconfiguration', or PDR.

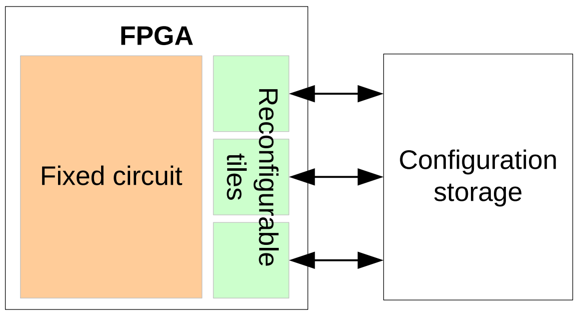

Figure 2.4: Illustration of partial dynamic reconfiguration. Self-contained tiles of the design can be swapped between the FPGA and external storage in the same way that conventional virtual memory swaps pages of data between main memory and magnetic storage.

Although PDR is still emerging as a design technique, numerous research projects have begun to exploit its potential in the field of reconfigurable computing. A common paradigm that PDR is applied to is that of a custom instruction set processor. Such a processor will generally provide a basic instruction set that is fixed but not very expressive and then the system designer can create new instructions that are dynamically loaded into the CPU when their opcodes are encountered at runtime. A classic example is the Dynamic Instruction Set Computer (DISC) [60]. DISC is entirely implemented on a standard FPGA, using a portion of the device as a fixed execution controller and the rest of the device as space in which to store custom instructions. Analysis of DISC shows that whilst the speed up obtained when performing general-purpose computing tasks is modest, application-specific tasks can be accelerated by up to 24 times by carefully selecting custom instructions that would be heavily used by the target application.

The Chimaera reconfigurable processor [28] is more of a hybrid approach. Chimaera is an ASIC-based CPU which contains an amount of embedded FPGA-style reconfigurable hardware. Chimaera uses this reconfigurable hardware to store the execution units of custom machine instructions, whilst the ASIC section performs all standard operations. Hauck et al. claim speed ups of up to 160 times in their system, but in reality such impressive results can only obtained in rather artificial circumstances with carefully hand-mapped examples. The authors state that general-purpose computing can be shown to be accelerated by approximately a factor of 2, but this is mitigated by the inevitable slowdown that is observed when moving from an entirely ASIC processor to an FPGA-based design. PDR has also been used to create a reconfigurable co-processor that sits alongside a standard CPU [29].

Marescaux et al. [43] and Huebner et al. [30] both propose NoC-based designs that may allow PDR to be used in a more general sense than the systems above. When an FPGA is partially reconfigured, care must be taken to maintain routing amongst all parts of the device that are still active. If an interconnection route passes through an area that is reconfigured then the connection will be broken. Reconfigurable processors avoid this by constraining the type and size of designs that can be swapped into the FPGA, whereas these network-based systems proposed attempt to relax this constraint and allow for general-purpose hardware to be swapped as required.

Currently, PDR is only employed in academic situations and has not been adopted for commercial or industrial systems due to a number of reasons. Primarily, because the concept is still relatively recent there is only limited tool support for PDR. Xilinx's PlanAhead hierarchical design tool attempts to simplify PDR but it is still considerably more difficult to use than a standard static design. As a result many designers will be discouraged from using what can be viewed as experimental technology. Also, a system that uses PDR can be difficult to verify from a safety-critical point of view due to the fact that during reconfiguration the device exists in an unknown state. After completion, extreme care must be taken to avoid metastability and other undesirable effects. Despite its potential benefits, it is unlikely that PDR will become a mainstream implementation choice until tool support has matured and at least some of these problems can be handled automatically.

2.3 Describing hardware with software

2.3.1 Hardware description languages (HDLs)

As mentioned in section 2.1, as the size and density of digital circuits develops it becomes increasingly impractical to describe them using schematic diagrams alone. The gate count of modern processors is measured in the tens of millions and a complete schematic diagram would be unmanageably large. Also, the poor level of abstraction afforded by schematics makes working with such diagrams difficult and error prone. To counter these problems, hardware description languages (HDLs) were developed. HDLs are a software-based technique for describing the arrangement of elements in a digital circuit. They describe the circuit's operation and organisation and frequently they can be used to simulate the described circuit and thereby verify its design. HDLs are a higher level of abstraction than schematics as they include notions of time and parallelisation. HDLs that express nothing more than the connections between circuit elements are called netlist languages, the most common example of which is EDIF [18]. Netlist languages are often used for storing schematics in a digital format and are notionally equivalent in terms of abstraction and complexity.

HDLs are implemented as standard textual languages that resemble software source code. A tool called a synthesiser is used to transform the HDL design into a netlist that can then be implemented as an ASIC or on an FPGA. This process is similar to compilation of standard source code and undergoes many similar steps, such as syntax checking, tokenisation and the generation of an abstract syntax tree. HDLs are easier to work with than schematics as they allow the designer to express relatively complex concepts that can be automatically translated to actual hardware by the synthesis tools. For example, the statement:

```
result <= (x + y) * z;
```

will automatically synthesise an adder and a multiplier of the correct widths and connect them as shown in figure 2.5.

The two most common HDLs that are in use today are VHDL [11] and Verilog [16]. They both provide a rich expressive environment for designing digital circuits and are widely supported by industry-level toolsets. VHDL (VHSIC Hardware Description

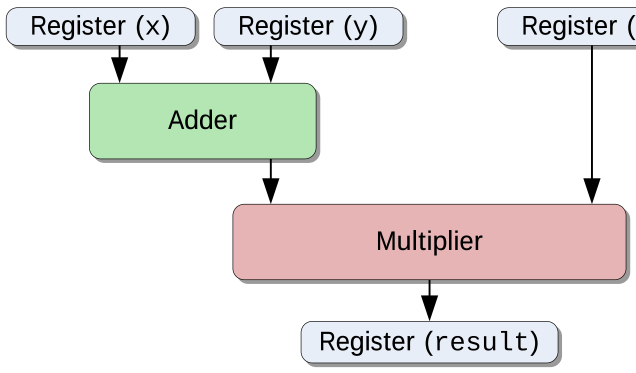

Figure 2.5: Pseudo-schematic showing the result of synthesising `result <= (x + y) * z;`.

Language) was initially developed as a way of documenting the behaviour of ASICs, but has now grown into a tool for designing, simulating and synthesising them also. As a result only a subset of VHDL can be directly synthesised to hardware, the rest of the language can only be simulated. VHDL was designed to be similar in style to the Ada programming language [2] and so it shares a very similar syntax, is case insensitive and is strongly-typed. Verilog, in contrast, was designed to be familiar to C programmers so it is case sensitive and uses a preprocessor. As with VHDL, only a subset of Verilog is synthesisable.

Both languages differ fundamentally from their procedural language counterparts, however. Ada and C are both imperative languages whereas VHDL and Verilog are declarative. This means that whilst in a standard programming language two consecutive statements are executed one after the other, in a HDL they are generally executed in parallel. This is because each statement describes a hardware element that is operational at all times. The designer must implement state machines and other techniques in order to achieve sequential execution.

Both languages offer support for structured system-level design because a designer can describe a component using high-level parts of the language that are outside their synthesisable subset. Whilst this code cannot be automatically synthesised to hardware, it can be simulated and and verified so the designer can be sure that the behaviour of the component is correct before attempting to express it as synthesisable code.

2.3.2 SystemC

A hardware-oriented language that is very well supported by industry today is SystemC. An IEEE standard, SystemC [32] is a system description language that can be used to design and simulate systems at multiple levels of abstraction. It is based on C++ [21] and implemented as a large library of classes and macros, thereby keeping the same syntax as C++. It can be viewed as both a hardware description language and a simulation language, as whilst its main aim is to verify the design of a system, a subset of the language can be synthesised directly to hardware. However, Grimpe and Oppenheimer [26] notes that the synthesisable subset of SystemC is equivalent in terms of expressive power to that of VHDL or Verilog, so it does not offer an extra layer of abstraction in the final hardware description.

SystemC models concurrent processes as 'modules' and describes the communications between them using a library of built-in primitives or user-defined composite types. System composition is modelled by 'ports', which are the connection points between which modules can exchange data. Transaction-level modelling can be used to describe these interactions between system components at a high-level of abstraction, and so simulation of the modelled system is often much faster and more comprehensible than VHDL or Verilog. Finally, one of the main strengths of SystemC is that it allows the designer to describe a system in very high-level abstract terms and verify it quickly through simulation. Then, in order to implement the design in actual hardware, individual components can be elaborated and a reification for each one created independently. Whilst this design style is also possible in VHDL and Verilog, SystemC provides more advanced modelling constructs and simulation is much more efficient.

Most of the criticisms of SystemC are related to its base language, C++. As C++ is a sequential language, describing concurrency and timing properties requires the use of preprocessor macros and library calls that can seem counter-intuitive, whereas in other HDLs these constructs are first-class parts of the language. This can be seen in figure 2.6, a SystemC description of a NAND gate. Many preprocessor macros are used (`SC_MODULE`, `SC_CTOR`...) in order to implement syntax that is not available in normal C++. Also, due to fact that SystemC is attempting to make C++ appear to be a declarative like VHDL or Verilog, it can be difficult to separate code that actually describes hardware from code that exists solely to aid the simulator. Figure 2.7 shows an equivalent VHDL description for comparison.

Also, as noted by Edwards [17] it is quite easy to inadvertently develop a non-deterministic SystemC model due to the fact that the simulator must mimic true concurrency on a sequential processor. This leads to a slightly different execution order each time the simulation is run, and so race conditions can develop when global variables or shared resources are used. This can be difficult to detect from the simulator without extensive testing and so the bug will only be discovered when the system is fully implemented in hardware.

2.3.3 High-level language synthesis

Whilst hardware description languages greatly aid the design of embedded hardware, newer design trends are beginning to strain current techniques. With the move to architectures based on the Network-on-Chip (section 2.1.2) paradigm and the increasing size of modern embedded systems, the current level of abstraction afforded by HDLs such as VHDL and Verilog is starting to prove insufficient. Current HDLs force the designer to express the functionality of their system in terms of state machines and interacting parallel processes. This works well for low-level modules that must interact with other off-chip device, but for larger modules and for general system-level behaviour it can become difficult and is a barrier to code maintainability and reuse.

As a result, there has been a great deal of research into the field of high-level language synthesis. The aim of such research is to increase the abstraction level of

```
#include "systemc.h"

SC_MODULE(nand2)  // declare nand2 sc_module
{
  sc_in<bool> A, B; // input signal ports
  sc_out<bool> X; // output signal ports

  void the_nand2() // a C++ function
  {
    X.write( !(A.read() && B.read()) );
  }

  SC_CTOR(nand2)  // constructor for nand2
  {
    SC_METHOD(the_nand2); // register do_nand2 with kernel
    sensitive << A << B; // sensitivity list
  }
};
```

Figure 2.6: A SystemC specification of a two-input NAND gate.

```
use ieee.std_logic_1164.all;

entity nand2 is port
(
  a,b: in std_ulogic;
  x: out std_ulogic
);
end nand2;

architecture struct of nand2 is
begin

  process (a,b)
  begin
    x <= a nand b;
  end process;

end struct;
```

Figure 2.7: A VHDL specification of a two-input NAND gate.

synthesisable HDL code, thereby making it simpler for the designer to create large systems without having to consider low-level implementation issues. Such high-level languages attempt to provide more expressive power than is afforded to the designer by standard HDLs. For example, this may include support for object orientation, transaction modelling or automatic generation of state machines. There are a number of high-level languages that can be synthesised to hardware, this section will detail some of the more commonly seen examples.

2.3.4 Handel-C

Handel-C [6] is a programming language developed by Celoxica for rapid prototyping of hardware designs to FPGAs and ASICs. It is implemented as a subset of ANSI C with a number of extensions that allow for parallelism and communication between parallel blocks. Due to Handel-C's stated goal of rapid prototyping, it does not allow the designer as much control over the implementation strategy as standard HDLs. Instead, it allows for more abstract concepts to be defined in code that looks very similar to standard sequential C. All state machine and control flow hardware is inferred and generated automatically by the Handel-C synthesiser. This means that an off-the-shelf solution that is written in ANSI C can often be synthesised directly to hardware, after minor rewriting for the conversion to Handel-C.

The process of synthesising Handel-C to hardware is very similar to the way in which normal C is compiled to object code. The compilation strategy is basically recursive descent, but rather than recursively generating machine code the synthesiser generates small hardware blocks which are recursively connected together. The resulting circuit is expressed in the EDIF format which can then be optimised by any number of tools before the design is converted to an FPGA bitfile by vendor-specific toolchains.

The Handel-C design process is fundamentally different to the way in which a circuit is built up with a hardware description language such as VHDL. As mentioned previously, VHDL is a declarative language whereas Handel-C (like its parent language C) is imperative. As a result, to describe two events that occur sequentially can be difficult in VHDL as a one-hot state machine must be described that switches between the two events in the correct order.

```
process Do_The_Tasks(clk, current_state)
begin
  if (clk'event and clk = '1') then
    if (current_state = 1) then
      Do_Task_One;
    elsif (current_state = 2) then
      Do_Task_Two;
    end if;
  end if;
end process;

process Change_State(clk)
begin
  if (clk'event and clk = '1') then
    if(current_state = 1 and Task_One_Finished) then
      current_state <= 2;
  end if;
end process;
```

In Handel-C this is much simpler because state machines are automatically inferred by the synthesiser and the inherently sequential nature of C can be exploited by simply calling the two items one after the other.

```
void main(void)
{
  Do_Task_One;
  Do_Task_Two;
}
```

Like most standard programming languages, normal Handel-C designs can only include a single clock source. The reason for this is that Handel-C was designed to have a very predictable timing model and the provision of multiple clock sources would undermine this aim. It is possible to make use of multiple clock sources in a Handel-C project but this is achieved by creating multiple designs, each with its own `void main(void)` function, and then linking them together using the Handel-C IDE. The designs can then export and import communication channels to share data asynchronously. This approach is quite limited because the encapsulation of design units is performed outside of the language and it gives no scope for modular composition, meaning that designs cannot be nested. Also, only channels can be shared between designs; variables, signals, interfaces and functions cannot.

There are a number of problems with Handel-C that limit its potential for exclusive use in the embedded systems market. First, as previously mentioned Handel-C has a very simple timing model that is intended to produce circuits with easily predictable performance characteristics. Essentially, the model states that "every assignment and communications statement takes one clock cycle, everything else has no cost". [57] This means that all assignments will take the same amount of time and so it is possible to inadvertently reduce the maximum speed of the entire circuit by introducing a single statement with a large propagation delay. This is illustrated in the following example:

```
void assignments(void)
{
  x = a + b;
  y = a >> 3;
  z = (a * (b + c)) / (d * e);
}
```

In this example, all three assignments take exactly one clock cycle. However, the assignment to `z` describes a large circuit with a significant propagation delay. If this is the longest path in the circuit then the maximum clock rate of the entire design is reduced. To avoid this, the programmer must manually split the assignment into multiple stages. This splitting cannot be done by the compiler as it would change the semantics of the program, and a statement that originally took one clock cycle would take longer.

Related to this problem is the fact that even though Handel-C appears to be standard C, it is in fact targeted at a very different implementation. Consequentially, a programmer who forgets this and writes in the style that they would for normal C will end up with an inefficient design. This can be seen when comparing loop termination constraints in the two languages. In C it is common to compose a loop as such: `for(x=0; x<10; x++)`. However it is more efficient in Handel-C to replace the less-than comparator with an inequality check: `for(x=0; x!=10; x++)`. The inequality check is a simpler circuit than the comparator that is required by the less-than operator and so it results in a smaller design. Finally, due to its reliance on standard C, Handel-C lacks real modular decomposition or encapsulation making it difficult to use when producing larger designs and limiting the language's potential for code reuse.

2.3.5 Other synthesised languages

Aside from those already presented there are a number of other languages that are targeted at hardware rather than at object code, but few have reached the level of commercial acceptance that Handel-C has attained.

The York Hardware Ada Compiler (YHAC) [57] has shown success in retargeting the Ada language to generate hardware designs in the form of EDIF netlists. (Ada is a very large language so only the Ravenscar subset [8] is implemented.) YHAC translates sequential Ada into hardware by generating one-hot state machines for each procedure. Unlike Handel-C, these state machines allow for individual operations to take multiple clock cycles, thereby reducing propagation delay in the final design and maximising potential clock speed. However, the inefficiency of a one-hot implementation style means that circuit scalability can be reduced and sizable programs can often be translated into very large circuits. YHAC can make use of Ada's native support for concurrency and will create a new state machine for each task, giving the programmer access to true parallelism. Therefore, YHAC implements Ada's coarse-grained concurrency model instead of the fine-grained model found in Handel-C.

Work by Cardoso and Neto [10] attempted to translate Java bytecodes into a dedicated hardware circuit that can be implemented on runtime reconfigurable hardware. The technique uses temporal partitioning, which separates bytecodes into graphs that do not need to occupy the device at the same time. Then, the system attempts to generate separate circuits for each temporal partition that then can be executed by a reconfigurable framework which implements the virtual hardware paradigm (discussed in [41] and [25]). The system extracts control dependency graphs, data dependency graphs and data flow graphs from the bytecodes and uses them to create the partitions. Problems with this approach are centred around its use of Java bytecodes as an input source. The translation of sequential byte codes to hardware can be rather inefficient as it results in circuits that are influenced more by the design of the Java virtual machine than by the source code from which they were generated. For example, the work does not make reference to custom datatypes, a one of Handel-C's strongest features. In Handel-C, if an integer will only ever take 4 values it is possible to implement it with only 4 bits rather than the default 16. This results in significantly reduced circuit size because all calculation units that operate on this variable need only be a quarter of the width. With the byte code approach, there is no way to carry this bit width information from the programmer to the compiler and so default implementation widths must be assumed.

Lava [4] takes a different approach to the languages above by using the Haskell functional language to describe circuits rather than an imperative language like C or Ada. Functional programming is based on the notion of computation as a series of function evaluations with little or no persistent state, as opposed to imperative languages that rely on a global state that is affected by procedures and subroutines. It is argued that functional programming is a better paradigm to describe hardware that performs digital signal processing due to the lack of state information in such designs. Lava works by embedding VHDL-style code in low-order functions and building a type system on top of this to ensure consistency. Higher-order functions can then be used to succinctly describe regularity that would ordinarily involve repetitive VHDL code (such as when describing look-up-tables, or densely-connected networks). As a result, Lava is very well suited for defining both the structure of a design and the way in which it is composed from smaller sections.

Lava allows for functions to have different interpretations, where only one of these creates an actual netlist. Other interpretations instead allow for designs to be simulated and verified. As Lava is a functional language, it is best used for describing circuits that could be considered pure functions, such as arithmetic operators or fourier transforms. As a result, Lava is not as much a general-purpose HDL as it is a domain-specific one. The scalability of such a system to support large designs is unclear as Lava's supporting literature states that the largest circuit it has been used to describe is a 128-bit wide combinatorial multiplier.

2.3.6 Summary

In summary there is currently no hardware-targeted language that that does not imply rather severe restrictions on the type of system that it can build efficiently. For example, Handel-C is good at creating circuits that are described by its one-hot state machine implementation strategy, but for designs that are best implemented using mainly asynchronous logic it can only describe an inefficient solution that is unnecessarily large and takes longer to execute than required. Similarly, Lava is based on a function composition paradigm that describes signal processing circuits well but cannot express state easily. This is a common problem with synthesis systems as they can only create circuitry using a single implementation strategy and, outside of a few pragmas, there is no language that allows the designer to override this to better suit their current project.

Unlike high-level synthesis, Hardware Description Languages give the designer the power to create almost any circuit that they require, but not without committing considerable effort to both development and verification. The end result is that high-level synthesis is desired for its speed and power when compared with HDLs, but it often cannot be used because it does not allow the designer enough control over the generated hardware. Instead, a hybrid solution may be employed. For example, high-level synthesis can be used to design the main processing cores of a circuit but their communications infrastructure may be described with an HDL such as Verilog.

It appears that high-level synthesis has hit a self-imposed limit and it will not improve without a change in the way it is realised. As mentioned above, synthesis works well within a narrow design space, but rapidly diverges from the optimal solution outside of this space, to the point where often large classes of designs cannot be implemented at all. A reason for this is that it is impossible to automatically synthesise a design artifact that cannot be directly expressed by the source language. For example, C is a single-threaded language with no concept of parallelisation or inter-task

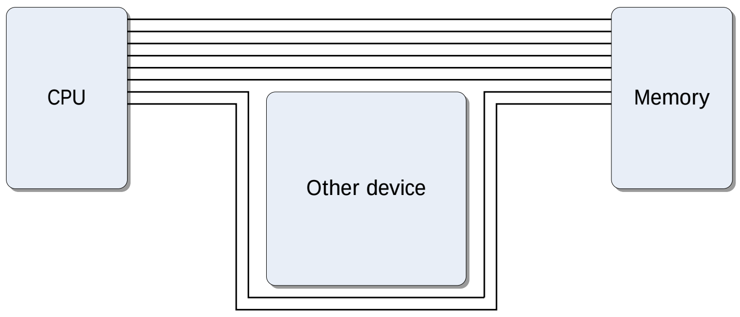

Figure 2.8: A common problem with undirected place and route. The last two wires of the CPU-memory bus are routed around a third component, resulting in a mismatched bus.

communication. As a result, Handel-C can only synthesise circuits that make use of these concepts through its extensions to C that enrich its original semantics.

These language extensions may be tempting, but they cannot completely bridge the semantic gap between design space and implementation fabric because both are changing at a pace that is far more rapid than that of the language. For example, the globally-asynchronous, locally-synchronous design paradigm was not of wide commercial interest ten years ago but is now recognised as an important new direction. As a result, new classes of hardware designs are required that are poorly covered by languages like Handel-C. Also, these designs may be implemented on currently unseen hardware because FPGAs are constantly developing with the introduction of new embedded cores or different communication methods. Extensibility of synthesis-based languages is therefore something that requires further investigation.

Finally, a problem common to both high-level synthesis languages and HDLs is that they rarely allow the actual physical layout of the design to be specified. They concentrate on describing functionality and a separate place and route tool is used to determine the actual physical location of the constituent circuit elements. The designer does not control this and the placer does not have access to any high-level structural information, only a netlist that is output from the synthesis tool. Often this is sufficient, but there are some situations when the designer needs to influence placing decisions. For example, it is often desirable to keep all wires of a bus running approximately parallel so they are of similar length and therefore display a similar propagation delay. If bus wires are mismatched then signal synchronisation issues can arise leading to data transmission errors. However, as most place and route tools do not have access to structural information, it can be hard to infer that a certain eight signals comprise a bus and should therefore be routed together. Instead the wires are routed individually and this can lead to the situation shown in figure 2.8.

2.4 Hardware/software codesign

Hardware/software codesign is an active field of research which is primarily motivated by the observation that, for most embedded systems, much of their functionality can be either implemented as dedicated hardware or as a software routine running

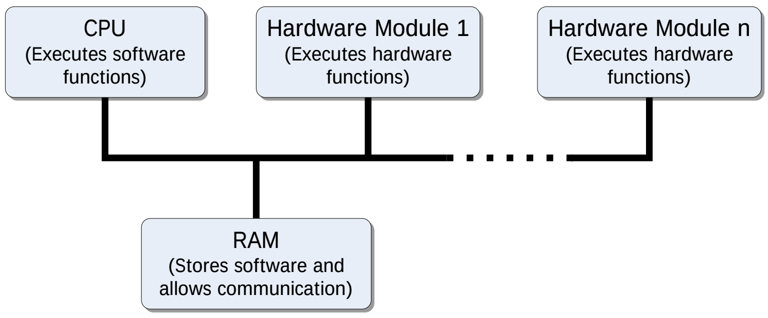
CPU
(Executes software functions)
Hardware Module 1
(Executes hardware functions)
Hardware Module n
(Executes hardware functions)
RAM
(Stores software and allows communication)

Figure 2.9: The target architecture of classical hardware / software codesign

on an embedded microcontroller. Both approaches carry inherent advantages and disadvantages and codesign attempts to balance these to find a sufficient design that meets restrictions placed on various metrics such as execution time, hardware density, power consumption or build cost.

In the classic description of codesign [61], the operation of a system is specified in an implementation-independent way along with a quantitative list of requirements that the final implementation must meet. The codesign engine then creates the hardware / software partition by assigning the various functions of the system to either hardware or software. The hardware functions are synthesised to dedicated hardware and the software functions are compiled to opcodes for execution on a predetermined processor core. Once these processes are complete, the entire system can be evaluated by a cosimulation engine that returns a set of performance metrics. If these metrics show that the system does not meet its specification the process is repeated. This time, however, the hardware / software partition is moved and so the implementation method for some functions is changed. This new implementation can then be regenerated and reevaluated until a solution is found that meets the system's initial constraints.

The target architecture for such classical codesign systems involves a single shared system bus, upon which sits a single CPU (to execute the software tasks), a number of custom hardware co-processors (to implement the hardware tasks) and a block of shared memory (see figure 2.9). More recent work has reduced this restriction; for example work by Niemann and Marwedel [47] describes a system that supports multi-processor architectures and Kalavade and Lee [33] examines the partitioning problem when applied to more general architectures.

Due to the lack of a truly implementation-independent language for expressing a system's functionality, all codesign frameworks must be either software-based, or hardware-based, depending upon the format in which the design is initially specified. In a system such as Cosyma [22], the entire design is described in a specially-created superset of C called C^X. C^X is still a software language, it simply augments C with a few necessary concepts, such as that of tasking. Therefore, initially a Cosyma design is entirely implemented in software. When the system is run, it uses a specified cost function to evaluate the design and then begins to automatically move parts of the code into dedicated hardware through the use of a synthesis engine. It uses the simulated annealing algorithm to search for a partition that meets the design requirements.

Conversely, the work by Gupta and De Micheli [27] is hardware-based because the initial system specification is a circuit design written in the language HardwareC. HardwareC, as its name implies, adopts most of the semantics and syntax of C but is modified to allow unambiguous hardware modelling. It resembles VHDL in part as it is a declarative language based on *processes* and *blocks* and it handles concurrency natively. In this work, only when the system cannot meet its stated requirements are sections of hardware moved to the software partition. This is achieved by translating from HardwareC to standard C using a code generation engine and then compiling the new code for a generic microprocessor. Despite their differing approaches, both Cosyma and the work by Gupta and De Micheli achieve similar levels of overall success, with acceleration factors ranging from no difference up to approximately twice as fast depending on the specific application.

2.4.1 Problems with codesign

There are a number of barriers to progress in the codesign field, but perhaps the most pressing is that the task of searching all possible partitions for an optimal solution is intractable and requires an exhaustive brute-force search in the general case. Codesign can be considered to be a search problem and so Wolpert's "No Free Lunch" theorem [62, 63] can be used to state that the problem of codesign can never be simply 'solved' in the general case. There is no single search algorithm that will be able to perform better than an exhaustive search of all possible partitions for every possible problem. As a result, heuristic-based search algorithms must be used and it is this that has been the focus of much recent work. [19, 20, 34, 56] It should be noted that this problem can be slightly mitigated in practice, as the *optimal* solution is rarely required, merely one that is good enough to meet the system requirements.

Secondly, most codesign frameworks rely on an accurate measurement of the performance metrics of the partitioned design to guide their heuristic search. In other words, the system makes decisions commonly based on the worse-case execution time (WCET) of both code and hardware and the expected size of compiled code and synthesised hardware. There is a massive body of work concerned with calculating the WCET of software but such analysis is very time consuming and relies on the target code being relatively small, written in an analysable language, compiled with a simple compiler and executed on a predictable CPU with a minimal amount of caching, branch prediction or otherwise complex behaviour. Also, WCET analysis is rarely automatic. Indeed, Ernst et al. [22] note that Cosyma incorrectly predicts the outcome of compiling C^X code due to the aggressive optimisation routines of the `gcc` compiler, and this can lead to a design that performs worse than before it was partitioned.

Similarly, it is difficult to accurately predict how large the outcome of hardware synthesis will be without actually performing it. For example, a design that uses many 4-input logic gates will take up more space on an FPGA with only 2-input LUTs than on an FPGA with 4-input LUTs. Without this level of implementation-specific knowledge any utilisation estimates can easily be incorrect. These analysis errors mount as the size of the design increases, making the codesign framework increasingly less effective. Similarly, it can be very difficult to predict the effect of communication and synchronisation delay in a partitioned system, and a few sources of such delay can negate the speed up that would otherwise be gained [61].

Currently, automatic codesign frameworks do not scale to work effectively on systems that are large enough to be of commercial interest. Some of the scalability and tractability problems must be solved before production-level embedded systems can benefit directly from such work.

2.5 Reflection

Computational reflection dates from Brian Smith's work in the early 80s [53] and has been described by Malenfant et al. [42] as a natural extension of the recurring trend in Computer Science to move towards languages with ever later binding times. Reflection is described as:

> "...the ability of a program to manipulate as data something representing the state of the program during its own execution. *Introspection* is the ability for a program to observe and therefore reason about its own state. *Intercession* is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called *reification*." [15]

There are two commonly identified aspects of reflection; structural reflection and behavioural reflection. Structural reflection allows a program to examine, reason about, and modify its own structure and all the abstract data types it uses. Informally speaking this means it can rewrite its own source code at runtime. Behavioural reflection does not change the actual code but instead affects its meaning and semantics.[4] For example, a C program including the line:

```
X = 2 * 3 + 4;
```

would execute leaving the value of X equal to 10. However, behavioural reflection may alter the language's precedence rules during execution so the meaning of the expression changes, the addition is calculated first and it now evaluates to 14. Alternatively, the meaning of the + symbol may be changed to represent subtraction and so result is evaluation to the value 2.

There does not exist a precise definition of what it means for a language to be reflective. This problem is further compounded by the fact that the implementation of reflection makes use of a number of techniques that existed before reflection became a research topic in its own right. (For example, the ability to manipulate programs as data, to inspect data structures at run-time, and to reason about metaobjects.) Therefore it is impossible to classify a language as reflective purely by considering the techniques it employs. Instead the intent and design goals of the language are often used, with Smalltalk (see section 2.5.1) being a commonly cited example of a language that was designed from the beginning to be reflective.

Absolute reflection is rarely achieved because if the concept is followed as far as theoretically possible the result quickly becomes unmanageable. For example, the

[4]As noted by [54] and others, reflection lacks a precise formal definition and so it is likely that the terms used in this section may appear elsewhere in slightly different contexts. This document attempts to use the terms in the most common way that they are presented.

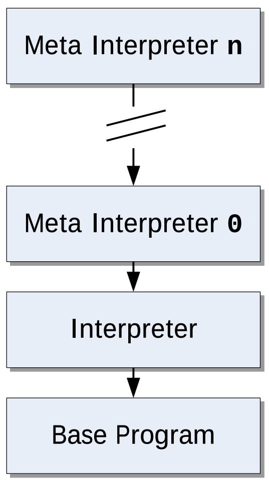

Figure 2.10: The Smith and des Rivières reflective tower. Note that these authors used the term 'reflective processor program' rather than interpreter to emphasise that interpretation is not required by reflection.

more powerful aspects of behavioural reflection are commonly implemented using run-time interpretation. However in an entirely reflective language the interpreter itself has a reification which can can be reflected upon. Therefore it becomes possible to create an altered interpreter, which must be run through a lower-level interpreter as a result. However, as this lower-level interpreter also has a reification it is possible to reflect on this also. This results in a theoretically infinite structure called a reflective tower, where each meta-level of interpretation implements the layer below it. Malenfant et al. [42] (see figure 2.10) Whilst a pleasing theoretical concept, it is debatable whether programs can be successfully written to make use of such power. Note that whilst interpretation is used here as an example technique for implementing reflection, it is not required and other methods are available.

There are a number of scenarios that benefit from the use of reflective techniques, but one of the most important is that intercession and behavioural reflection can allow a source language to influence the way in which it is implemented. In languages that support a high degree of reflection, the compiler itself is available to be reflected upon. This allows for a specific function to affect its own compilation and to control the opcodes that are eventually produced from the compilation process. For example, if a normal program is to be executed on a new processor with a custom instruction set then a new compiler would have to be written in order to generate correct object code. However, reflection allows this to be done from the source code level, maybe by simply including a new library that describes the desired instruction set architecture.

Kiczales et al. [37] give a further example of why it can be useful to control a language's implementation. Consider the following two records:

```
type point is
  record
    x : Integer;
    y : Integer;
end record;
```

```
type person is                        ...
  record                                --Many more data items
    surname : String;                 ...
    forename : String;              end record;
    age : Integer;
```

The `point` record has only two components and both are always populated with data. As a result its entire storage allocation is always used and access to both items must be fast. The `person` record however contains many data items which are empty, as the data items that are held on each person are different. Therefore, the record rarely uses its entire space allocation. Whilst the `point` record would benefit from an internal implementation that uses an array, the `person` record might be better implemented with a hash table. However, with a fixed language without reflection the compiler will treat both the same and use identical implementations, resulting in a compromise. Reflection allows the programmer to control the implementation style used and thereby increase the efficiency of the final program.

Another situation that benefits from reflection is where a program must interact with previously unseen classes and methods. This could occur in an operating system when it loads new programs, or in a web application that communicates with clients. Introspection makes it possible for the system to examine the available classes and methods of the new code and determine the best way to proceed, without requiring access to source code. It also allows the system to check for potentially unsafe behaviour in new code before executing it.

Reflection also introduces the possibility of runtime self-optimisation. For example, it is possible to create a system that profiles its own execution and reacts to improve its efficiency by altering its own implementation. This allows for a greater range of optimisations than traditional techniques because program flow in a non-reflective language can only diverge at points where the programmer has explicitly defined it (such as at `if` statements). In a reflective system it can happen anywhere. Similarly, reflection is also a powerful technique for implementing Metaobject Protocols (section 2.6 and the Aspect-Oriented Programming paradigm (section 2.7.1).

Clearly the main problem with reflection is that it can become very complex, and it can be difficult to use some of the power that it affords. Also, verification of reflective programs is currently an open problem as it is unclear as to how to reason about source code that can alter itself at any time. [52] Finally, many reflective systems must be interpreted in order to be implemented, which can result in a rather slow implementation. This is not the case for all systems, however, as careful restrictions on the reflective facilities available can allow a system to be compiled.

2.5.1 Smalltalk

Smalltalk[5] is an object-oriented, reflective language that was created at Xerox PARC during the 1970s. It bears a number of conceptual similarities to Lisp in that it is almost entirely written in itself, and is organised using meta-level objects that represent most parts of the language, including the classes, methods, compilers and even stack frames. Smalltalk has one of the most complete sets of reflective facilities of

[5]See http://www.smalltalk.org

any language in widespread use [51] and only avoids the provision of total reflection because of efficiency reasons.

Smalltalk was the first language to make the claim that in its object system truly everything is an object. Whilst languages such as Java are eventually built upon integral types that are 'boxed' into objects, in Smalltalk everything from numbers to blocks of code are first-class parts of the object hierarchy. All integers are singleton instances of the class `Number` and have their own methods that can be called. For example, the `Number` class provides a `negative` method that is used to check whether a number is negative. Note that this is not a library function, it is a true instance method. Therefore, to determine if the number 5 is negative, Smalltalk calls the `negative` method of the class instance '5', which will return `false`, an instance of the class Boolean. In C++ style syntax this would look like `if(5.negative)....`

To perform operations Smalltalk objects pass messages to each other, as can be seen in the following example:

```
mynumber := '18' asNumber.
mynumber := mynumber factorial.
mystring := mynumber printString.
```

In the above code, an instance of the class String (`'18'`) is sent the message `#asNumber` which causes it to be interpreted as a numerical string and returns a number object (18), which is saved in the variable `mynumber`. Then the `mynumber` variable is sent the `#factorial` message and the resulting number object is stored back in `mynumber`. Finally the `#printString` message is sent to `mynumber` which causes it to be reinterpreted as a string. As Smalltalk can deal with arbitrarily large numbers, the program ends with `mystring` holding a reference to a instance of the String class with data `'6402373705728000'`.

As previously mentioned, Smalltalk offers a great deal of reflective mechanisms to the programmer. For example, it is possible for an object to examine its own runtime stack or to refine the methods that it implements at runtime. Also, code blocks can be passed between objects and their contents examined and altered before being executed. This gives rise to the concept of classes cooperating to construct other classes, and is linked to metaobject protocols which are discussed in section 2.6. Figure 2.11 shows an example of one of the ways in which Smalltalk implements introspection and intercession.

2.6 Metaobject protocols (MOPs)

Often strongly related to the concept of computational reflection is the metaobject protocol. Described by Chiba [12], a metaobject protocol is a technique which allows programmers to customise the behaviour or implementation of a programming language. A common aim of such a technique is to balance the theoretical purity of high-level languages such as Prolog or Scheme with the high performance of more common lower-level languages like C or C++.

In an object-oriented language, runtime objects are instantiated from a more abstract view that is generally known as a 'class'. Classes describe the common behaviour of all derived objects, commonly by defining member variables or methods that other

```
"Declare 4 local variables for use."
| x y className methodName |

"Set className and methodName to string values."
className := 'MyClass'.
methodName := 'aMethod'.

"Evaluate className as code, rather than a string."
"x is set to the result of the evaluation."
x := Compiler evaluate: className.

"Check that the evaluation succeeded by asking if x"
"is a subclass of the class Class. Class is the ancestor"
"of all classes in Smalltalk and a descendant of the"
"Object class, the root of the hierarchy."
(x isKindOf: Class) ifTrue:
[
  "The evaluation was successful so create a new instance"
  "of MyClass. Save it in y."
  y := x new.

  "Ensure that y will respond to the method 'aMethod'."
  (y respondsTo: methodName asSymbol) ifTrue:
  [
    "It will, so execute it."
    y perform: methodName asSymbol
  ]
]
```

Figure 2.11: An example of Smalltalk's reflective capabilities. Smalltalk comments are enclosed in double quotation marks (`""`).

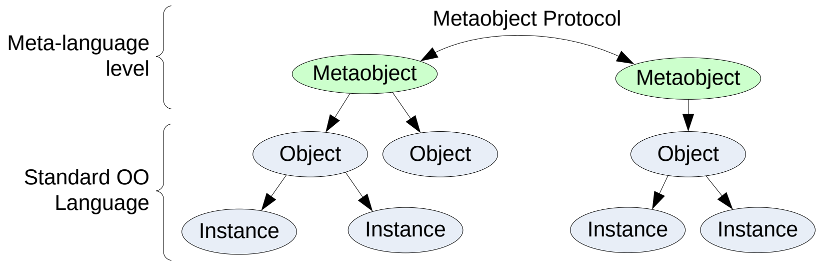

Figure 2.12: A metaobject protocol implemented above the layers of a standard object-oriented language.

classes can call, and it is at this level that all conventional object-oriented languages such as C++ and Java [6] operate. It is however possible to raise the object abstraction to the next level and to introduce ‘metaobjects’. Metaobjects describe metaclasses which can be instantiated to obtain a normal class. This class must then itself be instantiated in order to obtain a runtime object. The benefit of this is that metaobjects can describe the behaviour of a wide range of classes and so can therefore reason about the way in which such classes interact. This is known as a metaobject *protocol*. Note that whilst metaobject protocols have been used extensively in the implementation of reflective systems, metaprogramming itself is not reflection. Instead, it simply refers to the use of a higher abstraction level than that is normally afforded by object-oriented programming. Similarly, reflective techniques are not required to implement a metaobject protocol, although they are frequently used.

The next sections describe two of the more commonly used metaobject protocols, The Common List Object System Metaobject Protocol and OpenC++.

2.6.1 The Common Lisp Object System Metaobject Protocol

The Common Lisp Object System Metaobject Protocol (CLOS MOP) [36] is credited with being the first fully developed metaobject protocol and was developed as an extension to the Common Lisp Object System (CLOS). The CLOS is itself an extension to the declarative language Common Lisp that adds a complete object-oriented programming system. CLOS became so widely used that it is now an ANSI standard. The CLOS MOP is built on top of CLOS and is cited by most authors as the archetypal MOP.

Like all MOPs, the CLOS MOP was built to allow the programmer control over the implementation of the host language. It does this by enabling a number of reflective techniques, specifically programs can inspect the internals of a CLOS environment and also extend the CLOS language itself. This corresponds to aspects of both structural and behavioural reflection.

Like all metaobject protocols, the CLOS defines a metalevel architecture. As explained by Paepcke [48]:

“[This] describes the components of the system, its structural and proce-

[6]Java does actually provide some support for metaprogramming but it is rather undeveloped.

> dural building blocks and how they are put together. Examples of major building blocks are the manifestation of classes, slots[7] or methods in the language's implementation."

The CLOS MOP also defines a set of *protocols* which describe the ways in which the above building blocks can be manipulated to affect the runtime behaviour of the language. For example, at the times when a new class is defined the system's behaviour is governed by the *class initialisation* and *finalisation* protocols. These specify the runtime activities that together describe the process of defining a new class. Therefore, it can be said that the MOP operates at an abstraction level that is higher than the base language, and instead describes a metalevel view of the CLOS where its concepts are defined abstractly. By changing these protocols it is possible to affect a large amount of the way in which the CLOS is implemented.

The CLOS MOP is a runtime MOP, meaning that the metaobjects that describe the system exist and can be called and interacted with whilst the program is executing. Whilst this results in the highest level of reflective flexibility, it does require that the language is interpreted and therefore it executes slower than a compiled language.

2.6.2 OpenC++

OpenC++ [12] is a metaobject protocol for C++ that, like the CLOS MOP, allows for the language to be extended by meta-level programs written by the programmer. If no such meta-level program is given then OpenC++ is identical to C++. Unlike the CLOS MOP, however, the metaobject hierarchy is only available during the language's compilation phase and not to the program at runtime. As a result, OpenC++ executes as efficiently as a normal compiled C++ program and displays no other overhead, but none of its meta-level techniques can be used during execution because the MOP only exists whilst the program is being compiled. In effect, the OpenC++ MOP governs the translation from OpenC++ to C++ and the implementation of that C++ code, rather than its runtime behaviour.

An important aspect of the power afforded by OpenC++ is that it allows for the C++ language to be extended in a number of ways by the application programmer. Whilst not as general as the extendability of CLOS or Smalltalk, OpenC++ allows for new class modifiers, access specifiers, loop styles and closures to be defined. For example, the notion of distributed computing could be introduced to OpenC++ through the MOP, and a new class modifier defined - *distributed* - that is used to separate the distributed classes.

```
distribute class MyClass{ ... };
```

This code would have been syntactically incorrect before the language extension, but it can now be accepted and will affect compilation accordingly.

OpenC++ uses a two-stage compilation strategy that can be seen in figure 2.13. First, the meta-level program which deals with all aspects of the MOP is compiled using the OpenC++ compiler to create a compiler extension library. Then, this library is dynamically linked back into the compiler and used to compile the base-level

[7] *Slots* in Lisp are the constituents of a record or 'struct' type.

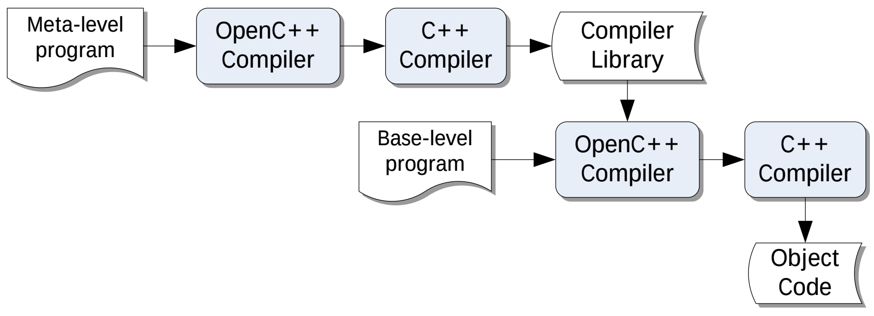

Figure 2.13: Overview of OpenC++ compilation

program (which does not concern the MOP in any way). The act of linking in the extension library has the effect of changing the compiler's behaviour and allowing it to accept programs that use extended syntax or metaobjects. In this way, the meta-level program controls the way in which the base-level program is translated from OpenC++ to normal C++. A normal C++ compiler is used to obtain object code from the result of this translation.

The obvious limitation of this technique is that the final compilation stage concerns only standard C++, and no metaobjects are present in the program at this point. The created executables have no greater power or expressive ability than one generated by standard C++. Instead, OpenC++'s strength lies in the way that it makes it easier to program complicated concepts by allowing language extensions and reflection on the OpenC++ compiler. This is shown in the development of the FRIENDS system, detailed in the following section.

2.6.3 The FRIENDS System

The FRIENDS system developed by Fabre and Perennou [23] is a set of meta-level libraries that allow for the development of group-based distributed applications with fault tolerance and secure communications. Developed in OpenC++, it is a good example of how metaobject protocols can be used to address higher-level concerns than that of base-level functional code. Applications can use the provided metaobjects transparently to add properties to their communication methods without any extra work from the programmer. Indeed, one of the stated aims of FRIENDS is to add meta-level properties to application code that are normally only provided by the operating system (which can be seen to be meta-level software).

The FRIENDS framework allows the application programmer to add fault tolerance mechanisms to an application simply by connecting the appropriate metaobject to the objects in their code. Most often, when developing an application which uses FRIENDS it will be written in standard C++. The only additional code is the OpenC++ `MOP reflect` declaration, which connects a base-level object to a metaobject. An advantage of this is that because all the FRIENDS metaobjects provide the same interface, communication mechanisms can be exchanged simply by switching metaobjects. It is possible to first develop an application locally in order to test its functionality and interactions, and then after these tests have been completed the communication metaobjects can be connected to make the application distributed and fault-tolerant.

As with all object-oriented systems, it is also possible to extend the metaobject system provided by the framework to develop custom communication mechanisms.

FRIENDS greatly assists the programmer to create applications with complex communication models, but due to its use of the OpenC++ MOP it can only be applied to programs written in C++. To partly address this, a metaobject-based fault-tolerant communications framework that is compatible with the CORBA ORB has also been developed. [39] Another problem is that FRIENDS does not allow for metaobject inheritance, so handling method inheritance in application-level objects is not currently possible. Also, as OpenC++ is a compile-time MOP it is not possible to adapt the communication style at runtime without extending the FRIENDS framework itself. For example, two communicating nodes in an embedded system may wish to communicate using a complex fault-tolerant protocol, but then to revert to a simpler, less robust protocol if battery power is low. This would not be possible without adding a new metaobject for this style of communications into the framework.

The final point which applies to many such meta-level techniques concerns verification of the final system. When applied correctly, systems such as FRIENDS can aid correctness verification as they reduce the amount of application code by encapsulating the communication logic in a single fixed library. Once the correctness of the FRIENDS library and the OpenC++ meta-level compiler has been verified the task of verifying a new application is simplified.

2.7 The inheritance anomaly and Aspect-Oriented Programming

The inheritance anomaly is an important observation that was first coined by Matsuoka and Yonezawa [44] in 1993, although the problem had been studied before then. It refers to the inherent problems that arise when attempting to combine inheritance and concurrency in concurrent object-oriented languages. Essentially the issue arises from the fact that most standard object-oriented languages require behavioural code and synchronisation code to be mixed together in class definitions, resulting in classes that are difficult to inherit from whenever alteration of the class's synchronisation constraints is required. A number of slightly different versions of the anomaly can be observed, and these are exemplified in [46] using the pseudocode example of a bounded buffer class.

```
class buffer{
   void put(Object x) when <buffer is not full>{ ... }
   Object get() when <buffer is not empty>{ ... }
}
```

The method declarations record two synchronisation constraints; that the buffer should only be read from when it contains data and only be written to when it is not full. However, if a new type of buffer that is dependent on its past history is desired then the first form of the anomaly is observed. For example, the history-dependent buffer may state that an invocation of `get` may only take place after at least 4 invocations of `put`, but the rest of the buffer's operation is identical. Whilst it may sound like the history buffer can inherit from the standard buffer, in fact both the `get` and `put` methods must be rewritten to count and check the number of `put` invocations. This results in the entire buffer being rewritten.

The other two versions of the anomaly are observed when an object's internal state affects the synchronisation constraints. Using the buffer example, the buffer can be thought to be in one of three states; *empty*, *partially-full* and *full*. `get` must ensure it can only be called when the buffer is in the *partially-full* or *full* states, but it is also responsible for updating the current state of the buffer from *full* to *partially-full* or from *partially-full* to *empty* as required. Now, if a new method is added that requires another state option then all other methods must be rewritten so that they will correctly change the buffer to this new state when they are invoked. This means that once again the entire buffer has been rewritten.

The third version of the anomaly is similar and is involved with synchronisation constraints that are affected by another object that is external to the class under synchronisation. For example, the above buffer may be extended using multiple inheritance to include a semaphore class, but all its methods must still be rewritten in order to ensure that they actually set and check the semaphore.

Essentially, the inheritance anomaly states that in the general case a derived class will end up reimplementing all the methods of its parent class if there is any change in its synchronisation constraints. This is a problem because it reduces code reuse and results in larger, more complicated programs.

2.7.1 Aspect-Oriented Programming (AOP)

It is impossible to formally prove that a language is susceptible to the inheritance anomaly, or indeed that it avoids it completely [45] but a lot of research has been undertaken recently to develop languages that attempt to use the principle of *separation of concerns* to reduce its effect. The aim of these languages is to specify synchronisation constraints away from the behavioural code, thereby allowing one to be altered without affecting the other. As described by Filman and Friedman [24]:

> "AOP can be understood as the desire to make quantified statements about the behaviour of programs, and to have these quantifications hold over programs written by oblivious programmers."

In other words, AOP attempts to give programmers the ability to define properties of a program's behaviour, and for these properties to still be enforced elsewhere, perhaps even in code written by other developers.

The most common techniques that are employed to achieve this goal are variations on an emerging programming style known as Aspect-Oriented Programming (AOP). AOP was introduced by Kiczales [35] in the mid 90s as an extension of their work on metaobject protocols. It is based on the idea that a program's design consists of a number of interacting *aspects* that define its operation. Common examples of aspects are behaviour, synchronisation constraints, failure handling, event logging, and peak memory usage. They argue that even though there are many aspects in a given system, functional behaviour is the only one that can be adequately described by current programming languages due to the fact that aspects tend to affect many

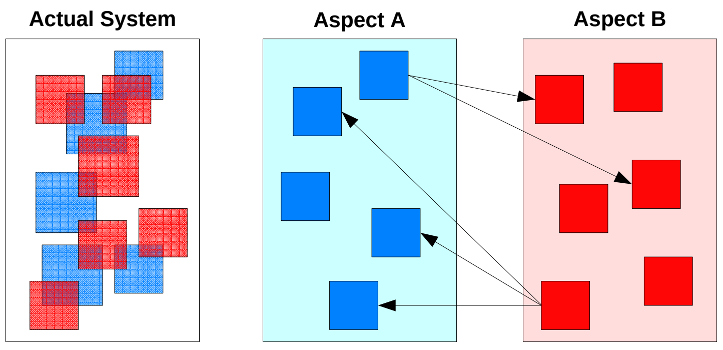

Figure 2.14: Aspect-Oriented Programming attempts to separate the actual system into a set of system models. Each model shows the system in terms of a different *aspect*.

different areas of the system and they commonly overlap with each other. This is termed *cross-cutting* in AOP parlance. The goal of AOP, therefore, is to provide a means for the programmer to separate out these aspects in the manner shown in figure 2.14. It is important to note that whilst reflection and metaobject programming are commonly linked to AOP they are merely convenient implementation methods and are not specifically required by the paradigm.

In an AOP implementation, the point at which the procedural code is joined together with the aspect code to produce the final behaviour is known as *weaving*. There are two broad categories of weaving, static and dynamic. Static systems weave at compile-time and can often be likened to a very flexible code preprocessor. Dynamic weaving is performed at runtime and is essentially the same as static weaving, except that aspects can be reprogrammed and moved around during the execution of the program. Currently, the most developed AOP implementations only support static weaving and dynamic weaving is limited to research languages.

2.7.2 AspectJ

AspectJ [38] is an extension to Java that adds many aspect-oriented features to the language. As it is a proper superset, all valid Java programs are also valid AspectJ programs. Since its initial development in 2001, it has become one of the most commonly cited examples of a static AOP language and is still supported by an active development community.

AspectJ extends Java to include the concept of aspects which are modelled as a collection of four new concepts:

- Join points
- Pointcuts
- Advice
- Inter-type declarations

Join points are a way of referring to specific points in the source code. These may be method calls, exception handlers, variable assignments, initialisations or other points of note. A set of join points can be collected together and referred to by a single name, which is termed a *pointcut*. Pointcuts contain no extra sematic meaning and are simply shorthand for describing many point cuts at once. *Advice blocks* are blocks of code that are executed when control flow reaches a specified join point or pointcut. It is with advice blocks that AspectJ injects new code into the user-level program. *Inter-type declarations* allow a programmer to add methods, fields, or interfaces to existing classes from within an aspect. This is used to extend a set of classes in a general way, without having to change their source code. Some of these features can be seen in the following example.

```
aspect Logging
{
  //Pointcut
  pointcut resourceAlloc():
    //Join points
    //Calls to someClass.allocate
    call(int someClass.allocate(..)) ||
    //Calls to someClass.getRes
    call(int someClass.getRes(..)) ||
    //Initialisation of resType objects
    initialization(resType);

  //Advice
  before() : resourceAlloc()
  {
    writeToLog("Attempting to get a resource");
    //other log processing activities...
  }
  after() : resourceAlloc()
  {
    writeToLog("Resource allocation successful");
  }
}
```

The code above shows an aspect that encompasses part of the logging requirements of a hypothetical system. It is intended that every time a system resource is allocated a note is made in the system log. Without AOP this would have to be achieved by rewriting every allocation of the resource to call the allocation routine through a logging procedure instead. This method is unsatisfactory because if there is a later change in the logging requirements of the system then the code must be amended at multiple points, increasing the possibility for errors to be introduced into the code. Also it is possible to inadvertently omit some allocations or to incorrectly rewrite them, introducing errors into the logging code.

The example overcomes this problem and describes the intended behaviour using the pointcut `resourceAlloc`. Specifically, `resourceAlloc` is comprised of two calls to specific methods (`someClass.allocate` and `someClass.getRes`) and the initialisation of objects of the class `resourceClass`. These are the three resource allocation routines that are to be logged whenever they are called. The aspect then gives advice about what to do when the pointcut `resourceAlloc` is encountered, which in this case involves writing to the system log. Note that advice can occur both before and after a pointcut is executed. The result of the code above is that all calls from anywhere in the source tree to the three allocation routines are guaranteed to also call the logging routines without any input from the programmer. This is error-free and it is much easier to debug because the entire extent of the logging code is contained within a single aspect, rather than throughout the program's source code.

2.7.3 Other AOP implementations

PROSE [50] is an AOP implementation based on Java that allows the programmer to perform dynamic weaving and thereby affect the behaviour of objects at runtime. Often, dynamic solutions are implemented using reflection or a metaobject protocol, but PROSE instead performs its weaving directly in the Java Virtual Machine and inserts aspect advice directly into the the code generated by the just-in-time compiler. This results in a very efficient solution.

AspectWerkz [5] is another dynamic framework that performs its weaving by directly editing the compiled object's bytecode to insert and remove advice as required. Unlike the other solutions, however, AspectWerkz separates aspect information from source code by storing it in accompanying XML documents.

2.8 Conclusions

This literature review has examined embedded systems and the way in which they have changed over the past decade. It has also covered the design flow of embedded systems and looked at the reasons behind the recent trend to move from standard monolithic circuit designs to ones that incorporate the System-on-Chip and Network-on-chip paradigms.

The review then proceeded to look at FPGAs and their applicability for implementing embedded systems. The FPGA design flow is explained, and from this the chapter proceeds to examine the methods that are used when describing hardware designs with a software language. This includes hardware description languages, such as VHDL and Verilog, and high-level languages that can be targeted towards hardware using a logic synthesis tool, such as HandelC, Ravenscar Ada, and Lava. Examples of such languages are considered and their relative strengths and weaknesses are highlighted. Hardware/software codesign is also discussed briefly along with its merits and shortcomings.

The review then considers techniques from the software domain that attempt to make programming languages more flexible by opening up their implementation to the user. Reflection is introduced and examples are given in the language Smalltalk. Metaobject protocols are then discussed and shown to be often linked to reflection. The CLOS and OpenC++ MOPs are considered and the FRIENDS system that makes extensive use of a MOP is detailed. Finally, Matsuoka and Yonezawa's inheritance anomaly is mentioned along with its ramifications for the use of inheritance in concurrent object-oriented languages. This leads on to the concept of Aspect-Oriented Programming, which is one of the first attempts to mitigate the anomaly's effects. Examples are presented in AspectJ, the archetypal AOP language.

Initially a niche market, embedded systems are now present in a continually increasing proportion of daily life, and as a result they are expected to become more powerful whilst maintaining their reliability and limiting their power consumption. This makes them harder to design and increases their entry cost and time-to-market. High-level synthesis was an attempt to mitigate these factors by automating some aspects of the design process and increasing the productivity of the designer. However, in practice high-level synthesis is rarely suitable for use on an entire system, only specially selected parts of it. It is clear that hardware design must increase in abstraction from the HDL level in order to facilitate the creation of ever larger systems, as happened in the software domain with the introduction of high-level languages. Current high-level synthesis solutions are a step in the right direction but prove inadequate in a number of important areas as they abstract away important details that the designer may need to control.

Chapter 3

Preliminary work

One of the most important points identified in this document is that current high-level synthesis techniques do not appear to be a suitable choice for designing anything but the simplest systems. This chapter attempts to highlight some of the reasons behind this and discusses the limitations that are placed on system designers when high-level synthesis is employed.

3.1 Description of the problem

Current HLLs employ abstraction to make software development easier and more productive. They remove the need to describe fixed parts of the target implementation (such as memory maps or device I/O) and rely on tool support from compilers and assemblers to automate many programming tasks. In the majority of languages the primitive building blocks (such as data types, I/O channels or mathematical operators) are fixed and their implementation is defined by the compiler. This model works well in the software domain because all software is executed on an unchanging von Neumann architecture. However, in the hardware domain this is not the case and the designer rather than the compiler needs to control the implementation scheme.

For example, when creating a DSP system it is common to use processors with custom instructions that are tailored to the intended application. Standard HLLs such as C provide operators for addition ($+$) and subtraction ($-$) because they are common to all PC CPUs, but custom instructions cannot be used in the same way as first-class parts of the language. This means that a synthesis tool based on C cannot correctly express the custom DSP chip. This does not simply apply to mathematical operators however and often designers create architectural features like busses or on-chip networks that similarly cannot be expressed by any HLL. In general, if a concept cannot be expressed by the language, then a synthesis tool will not be able to correctly infer its use and implement it.

Related to this is that fact that most languages can only describe concepts at their highest level of abstraction and provide no means of expressing implementation de-

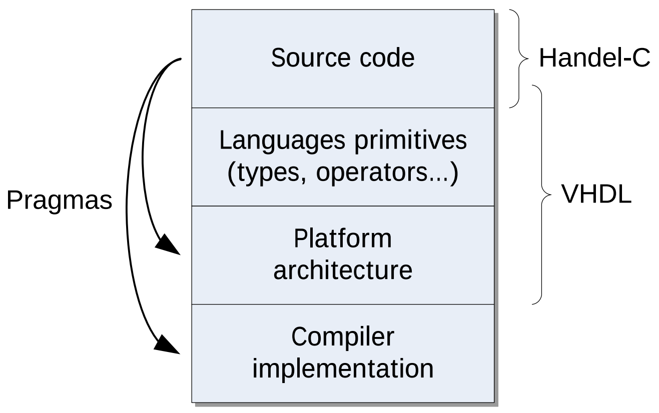

Figure 3.1: Abstraction levels exposed to the user by current hardware description languages. Handel-C only allows the user to describe systems at the highest levels of abstraction, apart from a small number of pragmas.

tails in a more concrete form (see figure 3.1). For example, a designer writing a case statement in C cannot drop a level of abstraction and choose whether it should be implemented as an indexed jump table or as a list of conditional jumps.[1]

Finally, HLLs also cannot adequately describe the non-functional constraints of components and so a HDL must often be used instead when timing or design size is important. Whilst it may be more difficult to work with, the lower level of abstraction afforded by HDLs allows for greater control over the final hardware. This leads to the situation depicted in figure 3.2 where, for example, Handel-C is used to implement high-level control circuitry but when greater implementation control is required VHDL components are used. The VHDL component is used to dig through the abstraction to expose a lower-level interface to Handel-C. The problem of languages only operating at a single abstraction level is starting to change with languages that are based on meta-object protocols (see section 2.6) but currently no such language exists for hardware synthesis.

All of these problems mean that existing high-level synthesis systems produce circuits that are inefficient for the majority of designs.

3.2 Problem analysis

To further understand the problem discussed above, this section presents an example application and then discusses possible descriptions of the system in two common languages, Ada and Handel-C. The aim was not to attempt to actually implement the example system; clearly this is impossible when using Ada as it is not a hardware description language. Instead, the example attempts to illustrate which aspects of the desired architecture can be sufficiently described (and therefore inferred) by a synthesis tool and which aspects cannot be described because of a lack of expressive power.

[1] C does allow the programmer to insert inline assembly code but this would not be of use because the jump addresses are not known until after compilation and assembly.

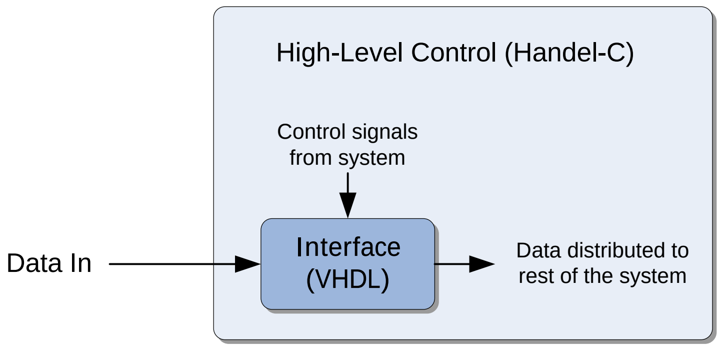
High-Level Control (Handel-C)
Control signals from system
Data In
Interface (VHDL)
Data distributed to rest of the system

Figure 3.2: A VHDL component inserted into a Handel-C design. In this example, Handel-C is incapable of adequately describing the non-functional properties of the interface so a VHDL model is used instead. The VHDL model then provides an interface that the rest of the system can use.

Ada is used in this example because it is a modern programming language that displays many of the common trends of imperative software languages and recent work [57] has shown success in re-targeting a subset of Ada to hardware. Handel-C is explored because it is a well-known high-level hardware description language that has gained acceptance in both industry and academia.

3.2.1 The desired architecture

The example used is a simplified control system for a robot that can react to its surroundings based on input from both a video camera and other sensors. The architecture of this control system is shown in figure 3.3. The vision system processes the incoming video stream using a range of filters to extract useful information about the environment and passes this information on to a main reasoning processor. This processing is performed with a pipeline comprised of three stages, with each stage feeding results to the stage directly following it. Other sensors detect features such as pressure and temperature and make this information available also. In order to interact with its environment, the robot is equipped with actuators that are controlled by a dedicated 'actuator control' module.

In order to reduce power usage the control system uses a network to pass information between processing nodes. As noted by Dally and Towles [14], networks commonly consume less power than bus-based solutions due to the reduced amount of wiring required. Also, the asynchronous nature of networks allows nodes to operate at different speeds, so less complex nodes can run slower and so therefore consume less power. The network supports atomic broadcast which is used to implement system-wide mode changes.

3.2.2 Ada description

A number of problems are observed when attempting to describe the architecture shown in figure 3.3 in Ada. The most obvious of these is that the design requires a

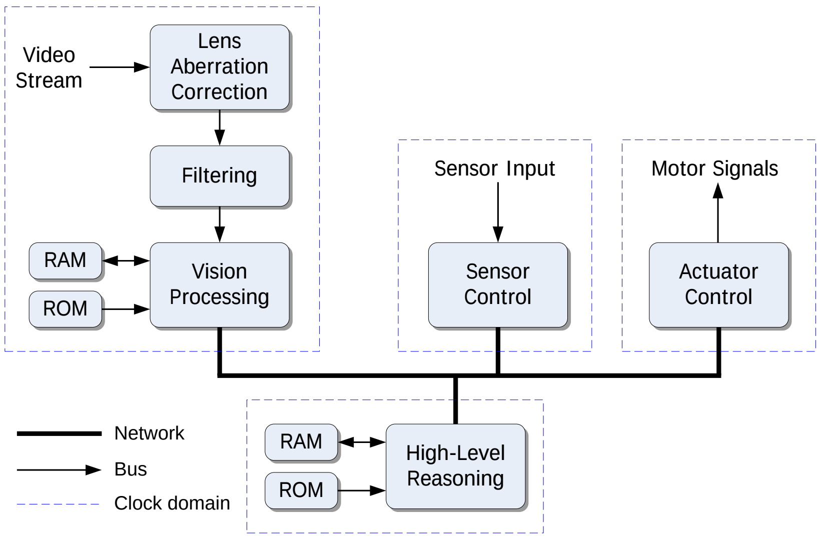

Figure 3.3: The desired robot control architecture. Dotted boxes denote different clock domains.

set of timing domains to be defined, but Ada possesses no concept of clocks or timing that can be used to do this. Due to this lack of clocking information, it is not possible to separate Ada tasks into timing domains without the use of the Distributed Systems Annex (DSA). Without this annex, tasks could either all be considered to be in the same domain, or all to be clock independent and therefore in separate domains, but not a mixture of the two. The function of the DSA is to allow a designer to separate an Ada program into a number of partitions that each execute on a networked computer. Within these partitions all tasks run at the same speed and all inter-task communication is synchronous, but communication across partition boundaries is asynchronous. Partitions are heavyweight entities; all inter-partition communication is implemented as a form of remote procedure call and calls are marshalled through the RPC stack for transmission over the network infrastructure. Such a sizable communications overhead makes DSA partitions unsuitable for use in separating small areas of a design.

The architecture of the vision pipeline intends incoming data to be processed as a continuous stream, however this cannot be sufficiently described by Ada so as a result the pipeline stages must be formulated as concurrent tasks with shared memory. When operation A begins, it loads data from memory, processes it, then writes it to the input buffer for operation B. Then, operation B must load the same data back out of memory in order to continue processing. Whilst this captures the semantics of the overall application, it is inefficient.

The two separate memory banks of the example architecture could not be cleanly modelled in Ada as it does not allow for such low level control over memory partitioning. Like almost every other general purpose programming language, Ada assumes a single pool of shared memory for all tasks and processes. Some separation can be obtained by using representation clauses to specify fixed addresses for all variables, and then splitting the memory banks based on address. This would allow variables to be placed into specific chips, but does not give control over the heap or other

```
protected Network is
  entry WaitForPacket(P : out Packet);
  entry SendPacket(P : in Packet);
private
  CurrentPacket : Packet := None;
end Network;

protected body Network is
  entry WaitForPacket(P : out Packet)
    when CurrentPacket /= None is
  begin
    P := CurrentPacket;
    if WaitForPacket'Count = 0 then
      CurrentPacket := None;
    end if;
  end;

  entry SendPacket(P : in Packet)
    when CurrentPacket = None is
  begin
    CurrentPacket := P;
  end;
end Network;
```

Figure 3.4: An Ada representation of a network that supports atomic broadcast. From inspection it is possible to observe that all readers will receive each transmitted packet, but there is no way of directly expressing this property in the language.

dynamic memory structures.

The network poses specific problems. As previously mentioned there is no way of describing multiple clock domains in an Ada program without using the DSA, and when using this method all inter-partition communications must be handled by remote procedure call mechanisms. It follows from this that it is possible to describe the network's high-level semantics, but not the low-level matters of network protocol which are performed automatically by the annex. Assuming that network nodes are modelled as tasks, network interactions must be described using Ada's inter-task communication mechanisms. However, in Ada all inter-task communication is one-to-one and so in order to distribute a packet to all potential receivers it is necessary to create an implementation like the one shown in figure 3.4. Whilst this would give the correct overall behaviour of an atomic broadcast, it is inefficient, displays very different timing properties and the meaning of the operation becomes obfuscated from the final code.

Also, in this implementation each packet arrives in a single operation, making it impossible to take action on a per-bit basis. For networks such as CAN which use bit-level arbitration this is clearly insufficient. Also, the semantics of the protected object used to model the network ensure that two packets will never be sent at exactly the same time. Therefore packets will not corrupt each other and it is difficult to introduce code that expresses the action to be taken when such an event occurs.

The 'vision processing' and 'high-level reasoning' modules in the example architecture represent embedded microprocessors that execute code stored in their associ-

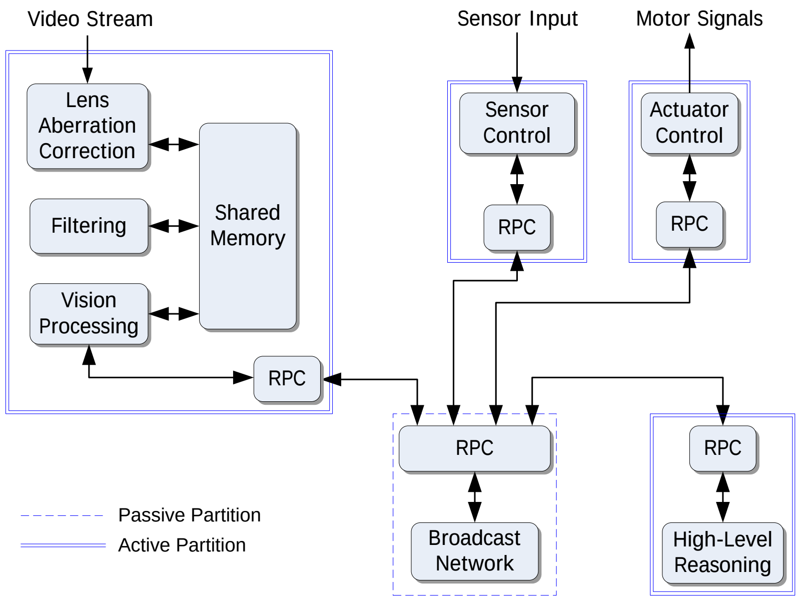

Figure 3.5: The example system described in Ada. The distributed systems annex is used to create separate clock domains but this requires all inter-partition communication to use the remote procedure call (RPC) mechanisms of the language. A passive network object is required to simulate broadcast of packets because Ada communications are one-to-one.

ated ROM and use their attached RAM for temporary storage. However apart from a few pragmas, Ada does not allow the designer any control over the manner in which code is implemented.[2] Therefore, a hardware mapping of Ada would require all modules to be implemented in the same way, all as embedded processor cores or all as dedicated hardware.

3.2.3 Ada results

Due to the difficulties listed in the previous section, one of the nearest descriptions that can be obtained using Ada is shown in figure 3.5. The most striking difference between this and the intended architecture is the way in which the separate clock domains communicate. The example specified that a network should be used that would reduce interconnection costs and support atomic broadcast. However, in order to separate the Ada design into different clock domains the DSA had to be used. As previously mentioned, the DSA requires all partitions to communicate using RPC and as a result the broadcast facility of the network has to be simulated using an extra protected object and code like that of figure 3.4. Although the high-level behaviour of this system is correct, the semantics of the network are completely disassociated from the solution and a synthesis tool would not be able to infer the correct implementation.

[2] Ada representation clauses can be used to refine how individual variables or records are implemented but the translations between source code and machine code are not configurable.

The vision pipeline is also incorrectly implemented as it must be described as a set of tasks running over shared memory. Ada does not allow low-level control over memory architectures so the result is a sub-optimal solution which displays the constant copying problem and may also introduce extra bottlenecks as the tasks must negotiate for control over the memory, reducing their ability to execute simultaneously. Finally, as noted previously all design nodes are implemented as software running on a CPU as Ada has no way of marking which code should be implemented directly in hardware.

3.2.4 Handel-C description

As Handel-C is a hardware description language its semantics include a number of hardware-specific features that Ada does not. For example, Handel-C includes low-level bit manipulation and variables of explicit widths. However, as a language it is not as expressive so higher-level concepts that Ada can use Handel-C cannot.

One major difference between the languages is the way that they handle concurrency. Handel-C uses the `par` statement to indicate instructions that can be executed at the same time. This allows for very fine-grained concurrency of individual statements whilst coarse-grained concurrency must be simulated, usually by enclosing function calls in a `par` block. Ada does not provide support for fine-grained concurrency but does support true coarse-grained concurrency with its tasking model. Ada tasks are heavyweight objects but they can be effectively used to represent individual system modules. Conversely, Handel-C does not explicitly separate system modules at all and instead treats the entire program as a single monolithic circuit. Therefore, Handel-C makes it difficult to discuss the ways in which system modules should interact as there is no concept of modular decomposition in the language.

Handel-C projects can only communicate across clock domains using channels, which causes a problem when attempting to implement a network in Handel-C. Channel communication is one-to-one only, so therefore to simulate a network which implements broadcast it is necessary to create a 'network controller' with a pair of channels (receive and transmit) that is connected to each normal network node. Whenever a node sends a message through its transmit channel, the controller echoes the message to all receive channels. This is shown in figure 3.6. This is very unwieldy as the implementation must be changed to incorporate different numbers of nodes and a synthesis tool will not be able to infer a network structure from this code. Also, the message is not received simultaneously by all readers.

Many of the other problems exhibited by Ada are also present in Handel-C. As noted previously, the vision pipeline would be best implemented as a data stream but because this cannot be directly expressed in Handel-C registers are automatically synthesised between pipeline stages and so the constant copying problem arises. Also, Handel-C does not provide a method that would be suitable for tagging code to be compiled to softcore instructions rather than synthesised to hardware.

One advantage that Handel-C has over Ada is the way that it can express memory. Handel-C allows the designer to describe memory layout easily by declaring RAM and ROM chips with language keywords. To declare a block of RAM that is 8 bits wide and 16k words deep the designer can use:

```
chan 8 node1_to_net, node2_to_net, node3_to_net;
chan 8 net_to_node1, net_to_node2, net_to_node3;

void network_server{
  int 8 message;

  while(1){
    //Wait for message
    prialt{
      case node1_to_net ? message:
      case node2_to_net ? message:
      case node3_to_net ? message:
        break;
    }

    //Echo it to all nodes
    net_to_node1 ! message;
    net_to_node2 ! message;
    net_to_node3 ! message;
  }
}
```

Figure 3.6: A Handel-C implementation of a network supporting broadcast. Data is written onto the network a byte at a time and echoed to all nodes. Simple extensions can prevent echoing to the transmission source or protect the server from locking on non-responsive reader node.

```
ram unsigned 8 VideoRAM[16384];
```

This can then be accessed like an array according to normal scope rules. A ROM is declared in a similar way and its contents can be specified in an initialiser. Normally, Handel-C will attempt to synthesise RAM and ROM from the target architecture but attributes can be added to the definition to specify that they are ‘off chip’. Memories declared in this way are assumed to be separate chips that will be connected to the design when it is implemented.

3.2.5 Handel-C results

Figure 3.7 shows one of the closest possible implementations when describing the example system in Handel-C. As with the Ada solution in section 3.2.3, the network displays the greatest variation from the original design. As previously discussed, to overcome the one-to-one nature of inter-clock domain communications, multiple channels and a network controller are used. This is an inelegant solution and it does not accurately reflect the designed behaviour.

The other problems that are visible were also displayed by the Ada solution. The vision pipeline is implemented with register banks between the stages due to the implementation strategy of the Handel-C synthesis tool. This is a slightly better solution than was afforded by Ada as the stages do not all compete for access to a shared memory module, however buffering should not be required at all and as a result they introduce inefficiency. Finally, all modules are implemented as dedicated logic and

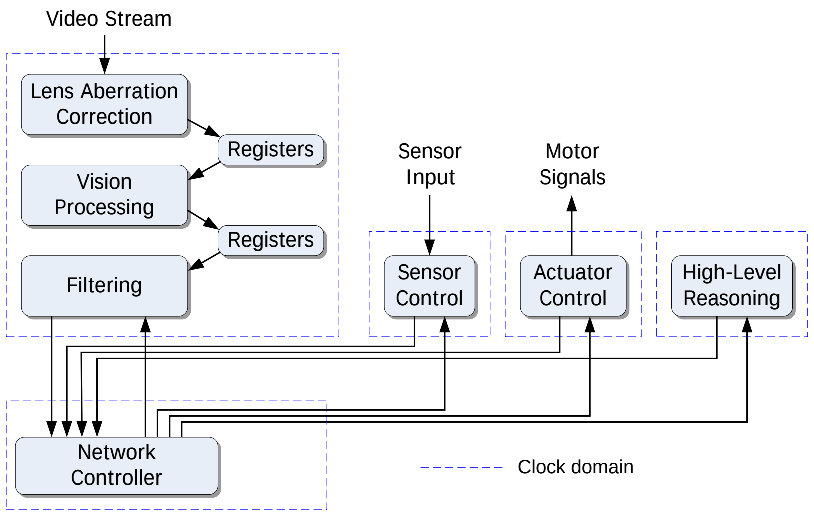

Figure 3.7: The example system described in Handel-C. As with Ada, a network controller is required to simulate broadcast of packets because Handel-C channel communications are one-to-one.

none use a softcore because, as with Ada, this cannot be expressed in the language.

3.2.6 Summary

The following table lists the major features of the example architecture and discusses whether or not they could be accurately described by either Ada or Handel-C.

Feature	Can be described?
High-level system behaviour	Yes. Both languages allow the high-level aspects of a system's operation to be described with little concern for lower level implementation details.
Concurrency	Yes. Ada tasks are notionally concurrent (coarse-grained) and Handel-C supports operation-level parallelism (fine-grained).
Separate clock domains	Partial. Both languages support such a feature, but their implementations are limited. In Ada, the DSA must be used which places tight constraints on communication. In Handel-C the partitioning is done outside the language and does not support modular design.
Non-uniform memory architectures	Partial. Handel-C allows explicit definition of RAM and ROM chips, registers are inferred for other variables. Ada provides no support for this and a shared memory model is assumed, with different DSA partitions having a different memory scope.
User-defined communications	No. Communication methods between concurrent blocks are limited to a few language-defined options that the user cannot extend. (Ada rendezvous and POs, Handel-C channels) Both languages enforce a single cross-timing domain communication method. (Ada RPC, Handel-C global channels) Networks, for example, cannot be described.
One-to-many communications	No. All communication must be one-to-one in both languages and it is impossible to express something of the form $(X\ and\ Y) := Z$. This is easy to construct in VHDL. The behaviour of one-to-many communications can be simulated, but the semantics do not exist to describe it adequately.
Use of softcores versus dedicated hardware	No. Both languages have a single, fixed implementation strategy. It is impossible to hint at the method in which a concurrent block should be implemented.
Streams, pipelines	No. Both languages do not support pipelining. When an operation completes both languages store the result to memory (or a register) automatically.

From examining the table above, it can be seen that both languages are able to describe the high-level functionality of a system adequately, but the adoption of a high abstraction level causes the implementation details to be inaccessible. The process of elaborating the system specification to a low-level implementation cannot be controlled, and so any decisions made by the synthesis tools are fixed. This primarily appears to affect communication between design entities. From the high-level system view, variables are simply read and assigned to. However, there are a great many different ways in which information can flow, be it a network, shared bus, dedicated connection, shared memory or another custom communication method. Whilst functionally similar (the data flows from source to target) they all have very different non-functional properties and can drastically affect the effectiveness of the solution.

Neither of the languages integrate clock domains into their syntax. Whilst they each provide limited support for separate clock domains, they are very coarsely-grained approaches that are performed at a program-wide level whereas a more suitable solution would allow for the mapping of parallel blocks to different clock signals.

It is not possible in either language to designate some aspects of the system as hardware and some as software. The same implementation strategy is used for the entire system and to describe the use of a softcore the designer would have to create the softcore from scratch, manually compile the code they want it to run, and then store this code in a linked program ROM.

Another problem highlighted by this study is that there is no link between the capabilities of the implementation fabric and the way in which it is used. FPGAs are very diverse and often include many embedded modules such as processing cores, memory or multipliers in addition to the reconfigurable array. The use of these from a high-level synthesis solution is often impossible, however, because they are not directly expressible by the source language. Vendor-specific tools can infer the use of multipliers and memory chips, but not modules such as processor cores or network interfaces. As a result, it is not possible to describe a system that uses these modules.

In summary, the implementation scheme used by high-level synthesis systems is a compromise that gives decent performance across the majority of its expected uses but very poor performance in other situations. There are only a few situations where hints or pragmas exist that can change the default behaviour; Handel-C's RAM and ROM statements are one such example. Therefore, the designer is forced to either accept the way that the synthesis tool builds implementations, or to attempt to coerce it into a more appropriate solution by programming in a stylised manner.

Chapter 4

Research Proposal

The literature survey conducted in chapter 2 has highlighted a number of potential avenues for further research. It has been determined that current high-level synthesis techniques are insufficient for designing most embedded systems and so it is this problem that is to be the focus of the proposed research. This section introduces the main research questions that are to be addressed by forthcoming work.

4.1 Research questions

Embedded systems are most commonly designed using Hardware Description Languages but due to the relatively low level of abstraction afforded by such languages, synthesis tools that operate on higher-level languages like C or Ada have been developed. These solutions are not satisfactory, however, because they do not present the designer with the sufficient expressive power to represent many of the newer design trends, such as the use of on-chip networks. It appears that one of the main reasons for this is that their implementation is fixed with the language definition. Aside from a few explicitly defined parts of the language, the designer is given no influence over the translation from their source code to a circuit description. No matter what the input code, a high-level synthesis engine will always use the same synthesis approach; a compromise that will be appropriate in some situations but poor in others.

Another important factor that contributes to the inefficiency of high-level synthesis solutions is that they use an abstraction model that is taken from the software domain and attempt to apply it to the hardware domain. To explain why this is incorrect, consider that when writing software the implementation architecture is already known. In the standard PC architecture a CPU is connected to a single block of shared memory which contains both data and code. Peripheral devices are memory mapped and interrupt schemes are predetermined. Although some systems may offer multiple execution units, this is masked by the underlying hardware and operating system. As a result, the application programmer does not need to consider these details and instead relies on the compiler, assembler and linker to assist in code generation. Whilst the compiler will add a slight inaccuracy when compared to handwritten opcodes, because the process is largely automatic a very high quality translation is generally achieved. This abstraction has enabled programmers to become more effective and write a greater volume of code.

In the hardware domain, however, none of the above constants apply. Rather than writing software to execute on a fixed architecture, the software is in fact attempting to describe the architecture itself. This does not present a problem when using a specially designed hardware description language like VHDL, but it does when using a software language like C. High-level languages are so named because they contain a certain level of abstraction that allows them to disregard architectural concerns. This makes them insufficient to describe architecture in an unambiguous manner and so current synthesis tools use a mixture of inference and fixed architectural strategies to 'fill in the gaps', but as previously mentioned these are a compromise and inappropriate for some designs. It appears a self-imposed limit has been reached by high-level synthesis and its inefficiencies will not be overcome without a radical redesign of the way in which it is applied.

The main research questions proposed are detailed below.

4.1.1 How can implementation architecture be expressed from within the source language?

Due to their origins as software languages, synthesised high-level languages cannot describe many common architectural features that the designer may require. For example, Handel-C only allows channels or signals to be used for communication between parallel blocks and the implementation of these is fixed. No provision is given for asynchronous communication styles, such as networks, or custom communication methods that may make use of special on-chip resources. The same problem can be observed with other high-level languages.

Similarly, non-uniform memory architectures are not always considered. Software language implementations expect to operate in a von Neumann architecture with all data stored in a single contiguous memory space. As a result unless specific constructs are added to the languages, as in Handel-C, memory maps cannot be correctly expressed. This problem affects synthesis of languages such as Ada, Java and Lava.

Finally, multiprocessor architectures are not considered by any existing technique. Whilst it is possible in many modern languages to split code into parallel tasks, the actual assignment of these tasks to individual processor cores is performed outside of the language by a static scheduler or by a run-time support system. Related to this is the fact that languages assume a symmetric multiprocessor architecture but asymmetric solutions are common in embedded systems where a powerful main processor may be assisted by a number of smaller, less powerful support cores. Assigning tasks to the correct cores is again something that must be specified outside of the implementation language.

This question therefore asks if there is a way to express architectural information in a high-level language in such a way that allows the synthesis tool to provide a better implementation. This requires a more precise definition of 'architectural information' and consideration of the form in which it would take.

4.1.2 How can the implementation of language constructs be influenced?

No current high-level synthesis language provides a mechanism for affecting the way in which its synthesis tool implements a given language construct. Certain pragmas may exist for special situations but in general the implementation strategy is fixed and the designer cannot influence the toolchain in any meaningful way. It may be possible to achieve a better solution if the designer can, when appropriate, give extra information that allows for a tighter mapping between the source code and the intended implementation. This would not be needed for the majority of the system, only the times when the default solution is not good enough.

This type of facility is partially available in the software domain through the use of reflective techniques and metaobject protocols that allow the implementation of a language to be affected by application source code. No such techniques have been applied to hardware design however, and their use in this context has not yet been evaluated. It may be possible, through the creation of a reflective synthesis tool, to build an extensible high-level synthesis language with an open implementation that combines the abstraction of a high-level language with the expressive power of a low-level HDL at the times when it is required.

This research question therefore asks in what ways the operation of the synthesis tools can be influenced. Are reflection and MOPs useful techniques to achieve this and if so do they need to be fully or partially implemented? Is it useful to only borrow a small number of ideas from these techniques, or does the main benefit only result from a more complete implementation?

4.1.3 How can non-functional properties be expressed?

Due to the abstraction models that they inherited from the software domain, synthesised high-level languages lack the ability to express many of a system's non-functional properties. For example, whilst it is possible to express an arithmetic function that combines a set of variables, it is not possible to state that the power usage of such a function should be kept below a certain level. Similarly, no high-level language allows the designer to specify the amount of space a design should consume. Were such facilities available, the synthesis engine could choose the implementation style that the designer has deemed to be most appropriate.

For example, when synthesising an integer multiplication there are a number of options available. The fastest solution is a single-clock combinatorial multiplier that will consume a great deal of logic and power but have the highest performance. Conversely, a shift-add multiplier could instead be built that takes many clock cycles to complete but is much smaller and consumes less power. It is possible to envision a synthesis system in which such decisions can be made automatically by the synthesis tool according to the non-functional properties expressed by the designer.

It is unclear which functional properties would be useful to express and in what way they would be used. The idea described above treats non-functional properties like system constraints in a codesign system, but it is possible to also consider other applications. For example, a system in which the designer can embedded the timings of a bus transaction and the synthesis tool will automatically create an interface to match them.

4.2 Continuing work

The following table describes the timetable for the research that is to be completed over the next 27 months. Approximately 1.5 months are unallocated to allow for unexpected delays.

1. Investigate a number of different methods for controlling the implementation strategy of the synthesis tool. These may involve the use of code templates, extension of the language type system, reflective techniques, metaobject protocols or some combination thereof. Determine in which ways the chosen method can extend the facilities that are already available in existing synthesis systems and which of these are the most important. *(3 months)*
2. Design and create a prototype tool that implements a basic set of synthesis features and only the most important additions, as identified in the previous stage. This prototype will not be able to synthesise an entire language, only enough for it to be reasonably evaluated. *(6 months)*
3. Evaluate the prototype to determine if it is capable of producing designs that are more efficient than is possible with existing systems. If the evaluation shows that there is promise in the chosen method, extend the system to become a complete synthesis tool by including support for an entire high-level language. If not, determine why and explore other options. *(10 months)*
4. Evaluate the complete tool by comparing its output against that of existing synthesis systems. A number of example designs will be created and implemented for this purpose. *(Less than 1 month)*
5. Write up the final thesis. *(6 months)*

Bibliography

[1] The ABEL hardware description language. http://mazsola.iit.uni-miskolc.hu/cae/docs/xabel.html, April 2007.

[2] J. Barnes. *Programming in Ada95*. Addison Wesley, 1995.

[3] Luca Benini and Giovanni De Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35(1):70–78, 2002. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/2.976921.

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *International Conference on Functional Programming*, pages 174–184, 1998. URL citeseer.ist.psu.edu/article/bjesse98lava.html.

[5] Jonas Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 5–6, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-842-3. doi: http://doi.acm.org/10.1145/976270.976273.

[6] M. Bowen. *Handel-C Language Reference Manual, 2.1 edition*. Embedded Solutions Limited, 1998.

[7] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-programmable gate arrays*. Kluwer Academic Publishers, Norwell, MA, USA, 1992. ISBN 0-7923-9248-5.

[8] Alan Burns, Brian Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In *Ada-Europe '98: Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*, pages 263–275, London, UK, 1998. Springer-Verlag. ISBN 3-540-64536-5.

[9] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. ISBN 0792399943.

[10] Joco M. P. Cardoso and Horacio C. Neto. Macro-based hardware compilation of Java(tm) bytecodes into a dynamic reconfigurable computing system. In *FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0375-6.

[11] Jr. Charles H. Roth. *Digital systems design using VHDL*. Pws Pub. Co., 1998.

[12] Shigeru Chiba. A metaobject protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA, October 1995. URL citeseer.ist.psu.edu/chiba95metaobject.html.

[13] Katherine Compton, Zhiyuan Li, James Cooley, Stephen Knol, and Scott Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):209–220, 2002. ISSN 1063-8210. doi: http://dx.doi.org/10.1109/TVLSI.2002.1043324.

[14] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. *DAC*, 00:684–689, 2001. doi: http://doi.ieeecomputersociety.org/10.1109/DAC.2001.935594.

[15] Francois-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[16] Doulos. The designer's guide to Verilog. URL http://www.doulos.com/knowhow/verilog_designers_guide/. Date retrieved: January, 2007.

[17] S. A. Edwards. Design and verification languages - tech. report cucs-046-04. Technical report, Dept. of Computer Science, Columbia University, 2004.

[18] Electronic Industries Association. Electronic design interchange format version 2.0.0 - technical report ansi/eia-548-1988. Technical report, 1988.

[19] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. Hardware/-software partitioning with iterative improvement heuristics. In *ISSS '96: Proceedings of the 9th international symposium on System synthesis*, page 71, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7563-2.

[20] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. System level hardware/software partitioning based on simulated annealing and Tabu search. *Design Automation for Embedded Systems*, 2:5–32, 1997.

[21] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[22] Rolf Ernst, Jorg Henkel, and Thomas Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test*, 10(4):64–75, 1993. ISSN 0740-7475. doi: http://dx.doi.org/10.1109/54.245964.

[23] Jean-Charles Fabre and Tanguy Perennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998. URL citeseer.ist.psu.edu/fabre98metaobject.html.

[24] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[25] William Fornaciari and Vincenzo Piuri. Virtual FPGAs: Some steps behind the physical barriers. pages 7–12, 1998. URL citeseer.ist.psu.edu/fornaciari98virtual.html.

[26] Eike Grimpe and Frank Oppenheimer. Extending the SystemC synthesis subset by object-oriented features. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 25–30, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-742-7. doi: http://doi.acm.org/10.1145/944645.944652.

[27] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Des. Test*, 10(3):29–41, 1993. ISSN 0740-7475. doi: http://dx.doi.org/10.1109/54.232470.

[28] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. *5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, 00:87, 1997. ISSN 1082-3409. doi: http://doi.ieeecomputersociety.org/10.1109/FPGA.1997.624608.

[29] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. *5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, 00:12, 1997. ISSN 1082-3409. doi: http://doi.ieeecomputersociety.org/10.1109/FPGA.1997.624600.

[30] Michael Huebner, Tobias Becker, and Juergen Becker. Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 28–32, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-947-0. doi: http://doi.acm.org/10.1145/1016568.1016583.

[31] Institute of Electrical and Electronics Engineers. IEEE standard test access port and boundary-scan architecture (IEEE std 1149.1-1990). 1990.

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

[33] A. Kalavade and E. A. Lee. The extended partitioning problem: hardware/software mapping and implementation-bin selection. page 12, 1995.

[34] Asawaree Kalavade and Edward A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *CODES '94: Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 42–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-6315-4 (PAPER).

[35] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/242224.242420.

[36] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-262-61074-4.

[37] Gregor Kiczales, J. Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow. *Metaobject Protocols: Why We Want Them and What Else They Can Do*, pages 101–118. The MIT Press, Cambridge, MA, 1993. URL citeseer.ist.psu.edu/399658.html.

[38] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4.

[39] M. O. Killijian, J. C. Fabre, J. C. Ruiz-Garcia, and S. Chiba. A metaobject protocol for fault-tolerant CORBA applications. page 127, 1998.

[40] Shashi Kumar, Axel Jantsch, Mikael Millberg, Johny Oberg, Juha-Pekka Soininen, Martti Forsell, Kari Tiensyrja, and Ahmed Hemani. A network on chip architecture and design methodology. *ISVLSI*, 00:0117, 2002. doi: http://doi.ieeecomputersociety.org/10.1109/ISVLSI.2002.1016885.

[41] X. P. Ling and H. Amano. WASMII: a data driven computer on a virtual hardware. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[42] J. Malenfant, M. Jacques, and F. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of Reflection 96*, 1996.

[43] T. Marescaux, J. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, and R. Lauwereins. Networks on chip as hardware components of an OS for reconfigurable systems. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications, Lisbon*, 2003. doi: 10.1007/b12007.

[44] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[45] Giuseppe Milicia and Vladimiro Sassone. Jeeg: temporal constraints for the synchronization of concurrent objects: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):539–572, 2005. ISSN 1532-0626. doi: http://dx.doi.org/10.1002/cpe.v17:5/6.

[46] Giuseppe Milicia and Vladimiro Sassone. The inheritance anomaly: ten years after. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1267–1274, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-812-1. doi: http://doi.acm.org/10.1145/967900.968159.

[47] Ralf Niemann and Peter Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 2:165–193, 1997.

[48] A. Paepcke. User-level language crafting. In *Object-Oriented Programming : the CLOS perspective*, pages 66–99. MIT Press, 1993. URL `citeseer.ist.psu.edu/paepcke93userlevel.html`.

[49] P. P. Pande, C. Grecu, A. Ivanov, and R. Saleh. Design of a switch for network on chip applications. In *ISCAS '03. Proceedings of the 2003 International Symposium on Circuits and Systems*, 2003.

[50] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-469-X. doi: http://doi.acm.org/10.1145/508386.508404.

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

[52] Juan-Carlos Ruiz-Garcia, Jean-Charles Fabre, and Pascale Th&#233;venod-Fosse. Testing metaobject protocols generated by open compilers for safety-critical systems. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 134–152, London, UK, 2001. Springer-Verlag. ISBN 3-540-42618-3.

[53] Brian Smith. Proceedural reflection in programming lanugages. Technical report, Massachusetts Institute of Technology, 1982.

[54] J. Sobel and D. Friedman. An introduction to reflection-oriented programming. In *Reflection '96*, 1996.

[55] David Tennenhouse. Proactive computing. *Commun. ACM*, 43(5):43–50, 2000. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/332833.332837.

[56] Frank Vahid. Modifying min-cut for hardware and software functional partitioning. In *CODES '97: Proceedings of the 5th International Workshop on Hardware/Software Co-Design*, page 43, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7895-X.

[57] Michael Ward. *Improving the Timing Analysis of Ravenscar / SPARK Ada by Direct Compilation to Hardware*. PhD thesis, York University Computer Science Dept., 2005.

[58] Daniel Wiklund and Dake Liu. SoCBUS: Switched network on chip for hard real time embedded systems. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 78.1, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1926-1.

[59] J. Williams, N. Heintze, and B. Ackland. Communication mechanisms for parallel DSP systems on chip. 2002.

[60] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. *3rd IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '95)*, 00:0099, 1995. doi: http://doi.ieeecomputersociety.org/10.1109/FPGA.1995.477415.

[61] Wayne Wolf. A decade of hardware/software codesign. *Computer*, 36:38–43, 2003.

[62] David H. Wolpert and William G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe, NM, 1995. URL `citeseer.ist.psu.edu/wolpert95no.html`.

[63] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. URL `citeseer.ist.psu.edu/wolpert96no.html`.

[64] Xilinx Corporation. Spartan-IIe FPGA family: Complete data sheet. DS077, July 2004.

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

[66] Xilinx Corporation. Virtex-5 FPGA configuration user guide. *Xilinx User Guides*, UG191, 2006.

[67] Andrew Chi-Chih Yao. New algorithms for bin packing. *J. ACM*, 27(2):207–227, 1980. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322186.322187.

[68] Cesar Albenes Zeferino and Altamiro Amadeu Susin. SoCIN: A parametric and scalable network-on-chip. In *SBCCI '03: Proceedings of the 16th symposium on Integrated circuits and systems design*, page 169, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2009-X.