Object-Oriented ASIP Design and Synthesis

Maziar Goudarzi[1,2], Shaahin Hessabi[1]
[1] Sharif University of Technology
Department of Computer Engineering
Azadi Avenue
Tehran, I.R. Iran
gudarzi@mehr.sharif.edu hessabi@sharif.edu

Alan Mycroft [2]
[2] University of Cambridge
Computer Laboratory
JJ Thomson Avenue
Cambridge CB3 0FD, UK
am@cl.cam.ac.uk

Abstract

SystemC-Plus from the ODETTE project provides the ability to simulate and synthesise object-oriented specifications into hardware. The current ODETTE compiler translates each object instance into a finite state machine; this effectively duplicates (modulo dead code removal) in hardware the methods of a class for each of its instances. We describe an alternative synthesis approach which parallels the software implementation of objects in which instance attribute values are stored in (registers or) memory and routed, on use, to a per-class implementation of methods. Interestingly, this technique can be seen as mapping a class library into an ASIP (application-specific instruction set processor) whose instructions correspond to *public* methods of the library. The ODETTE-synthesisable OO features correspond to an ASIP architecture with instructions only having direct addressing. We then show how to extend the synthesisable subset of ODETTE to include object instances addressed by pointer as is common in software. This corresponds to adding indirect addressing modes to the ASIP and highlights differences between software and hardware understanding of the notion of object. We view the ODETTE approach and our ASIP approach to OO hardware design as two extremes of a spectrum of design points in between, trading off concurrency against area/power consumption. This work is part of the ODYSSEY (Object-oriented Design and sYntheSiS of Embedded sYstems) project and introduces the OO-ASIP as the system building block in ODYSSEY.

1 Introduction

VLSI designers are (and will continue to be) facing a "quadruple whammy": more transistors per chip, harder to design transistors and interconnect in deep sub-micron technologies, more heterogeneous elements on the same silicon, and tighter time-to-market deadlines [Aug02]. Hence, chip design, synthesis, and manufacturing is continuously getting harder, more expensive, and more time-consuming. An application-specific instruction processor (ASIP) addresses this since it is tailored to a *set of target applications* and hence can be manufactured in larger volumes to reduce the unit price, while allowing programmability through software to shrink time-to-market and ameliorate design risk. This risks additional costs of lower performance and higher area and energy consumption when compared to ASICs. To minimise this risk, a five-step disciplined methodology to ASIP development is suggested by the MESCAL project [MES03], proposing disciplined benchmarking, defining the architectural space, efficiently describing the space to be explored, exploring the design space, and exporting the programming environment [Keu02].

On the other hand, the still-growing design productivity gap [ITRS01] motivates higher abstraction levels to increase designer productivity. Object-oriented methodology is one answer to this challenge especially when considering its already-gained reputation in software design and observing that software design routinely accounts for 80 percent of embedded-systems development cost [ITRS01].

We stay in the 5-step discipline of MESCAL, but we propose a new view of instructions in instruction-set-based ASIPs resulting in associating the ASIP, which we call an OO-ASIP, with a class library. This correspondence encourages OO-ASIP reuse wherever its corresponding class library is reused. Selected methods of the class library form the ASIP instruction-set, enabling the designer to control granularity of ASIP instructions according to his insight and expertise in the target application domain. Several previous works exist addressing OO hardware design and/or synthesis by extending VHDL [Rad00, Ash97, Sch95], synthesis from Java [Kuh01, You98, Hel97], and C++/SystemC [Gri02b]; however, none of them involve an ASIP approach. Wolf [Wol96] used designer-provided decomposition hints (objects and methods) in OO specifications to partition and co-synthesise an OO model on a distributed engine of heterogeneous processors. By contrast, we use those ideas also within processors.

The system-level building-blocks we currently use are objects of a C++ class library, which we call the "system class library" tailored to the set of target applications. Some basis classes and methods of this "system class library" (*hardware classes* and *methods*) are synthesised as the instruction-set of the OO-ASIP, while the other classes and methods are implemented in software to run on the OO-ASIP (the *software classes* and *methods*). Our prototype implementation translates C++ to SystemC for the hardware partition. This enables C++ to be used as the system-design language and then preprocessors convert parts of it to either hardware or software. It is noteworthy that our emphasis is on the OO methodology, and not merely on the C++ language; therefore, we do not concern ourselves with all special cases that can arise from arbitrary use of C++ features—see Section 5. C++, SystemC, and hardware synthesis from SystemC are enabling technologies that we employ to painlessly implement our ideas.

In the rest of this paper, we first introduce our approach to corresponding an OO-ASIP to a class library. Then in Section 3 we show how such an ASIP is synthesised from a "hardware class library". Interesting synthesisable features that the scheme enables are presented in Section 4. System-design language and co-design issues are discussed in Section 5, and finally Section 6 summarises and concludes the paper.

2 The OO-ASIP Approach

An object-oriented class library can be seen as providing access to various (public) operations on instances of classes from that library. The operations (*methods*) effectively define an abstract data type. A user of such a class library will repeatedly provide an instance of such a class and invoke one of its public methods possibly with arguments; this returns an instance of the same or another class after possibly invoking other operations internally.

We wish to identify this process with that of machine code instruction execution. The above user now corresponds of a stream of instructions each having operands identifying arguments and results, and whose execution implements to the relevant method invocations. This work realises this correspondence by mapping a class library (the "hardware class library") into an ASIP. Further, by considering classical CPU operations on integer and floating point values as predefined classes and methods (add, subtract, multiple, comparison and the like) the ASIP can also provide traditional CPU functionality. Similarly, each class can be seen as having predefined operations to allow (attribute parts of) object instances to be input and output from external buses—this can be done in serial, parallel or via the more sophisticated IIOP protocol from CORBA [COR03].

Now, suppose we have a larger class library, which we suppose to consist of a "hardware class library" extended by further class definitions (the "software class library"). The hardware classes can be therefore seen as defining the ASIP and the software classes as being a program for it. A diagrammatic form of this partitioning is shown in Fig. 1 where, for simplicity of presentation, the metaphor is that of a single file containing a marker between class definitions separating the

hardware and software parts.[1]

code origin	program source	target partition
built-in	`class int { /* predefined */ };`	hardware (hardware class library)
user-provided	`class A { ... };` `class B { ... };`	
	`class C { ... };` `class D { ... };`	software (software class library)

Figure 1: Source separation versus target partition

Firstly, let us consider what happens if the class library provides a single *public* method, `main()` taking no input parameters (implying it is *static*) and giving no output results. This corresponds to the only instruction, which can therefore be coded in zero bits and hence is always executed. Such a library therefore directly corresponds to an ASIC—the program is fixed, all that varies is the data. Now consider the general case; we may provide several *public* methods, e.g. for various IP packet manipulations, each of which corresponds to an instruction. Therefore what we have designed is an ASIP. Now suppose that `main()` is not a hardware method and we add further software classes and that these include a `main()` method as entry point. Methods in such later classes either call each other (this just corresponds to an ASIP software function call) or call hardware methods (this just corresponds to an ASIP single instruction). Hence the ASIP contains primitive instructions corresponding to:

- hardware methods (each operand is given by an operand specifier which is an object identifier (*oid*), as above `int`/`float` is a special case).

  Note that so far these only use "direct addressing" operand format; an object is specified by an *oid*. Note that direct addressing here is not restricted to addressable memory—operands may equally well be specified as a processor special purpose register (obtained by routing) or a register or memory location obtained by register-file addressing or RAM addressing mechanisms.

- traditional subroutine call and return (to invoke software methods), control flow (if-then-else, loop, jump), and move instructions (to pass objects by value to non-inlined procedures).

Approach to HW/SW Partitioning and Co-design. Looking at the above ASIP approach a little more clearly gives the following interpretation: all methods defined in the "hardware class library" are synthesised into hardware (the details are given in Section 3). Public[2] methods are associated with instructions, whereas non-public ones can be more highly optimised across function boundaries e.g. by inlining. "Software methods" are then translated into a sequence of instructions representing method calls, and thus are the software for the ASIP, for example stored in flash ROM. What we have achieved is a form of *closely coupled* hardware/software design in which syntactically identical code can be translated into either hardware or software.

One issue here is that hardware and software tend to have rather different views on object representation (see Section 5 for more details); hardware tends to pass object instances around by value, copying them from register to register whereas software tends to avoid the overhead

[1]Note the connection between the hardware and the software partitions; the partitioning function may be given in any suitable way, such as by keeping software and hardware methods in separate files, by annotations on methods specifying their category etc.—we do not really concern ourselves with this issue here.

[2]It is arguable we might want a method to be *public* so that other hardware-implemented classes can use it, but not have it associated with an instruction. This would argue for an additional keyword, but for simplicity in the paper we just associate *public* with "being associated with an instruction".

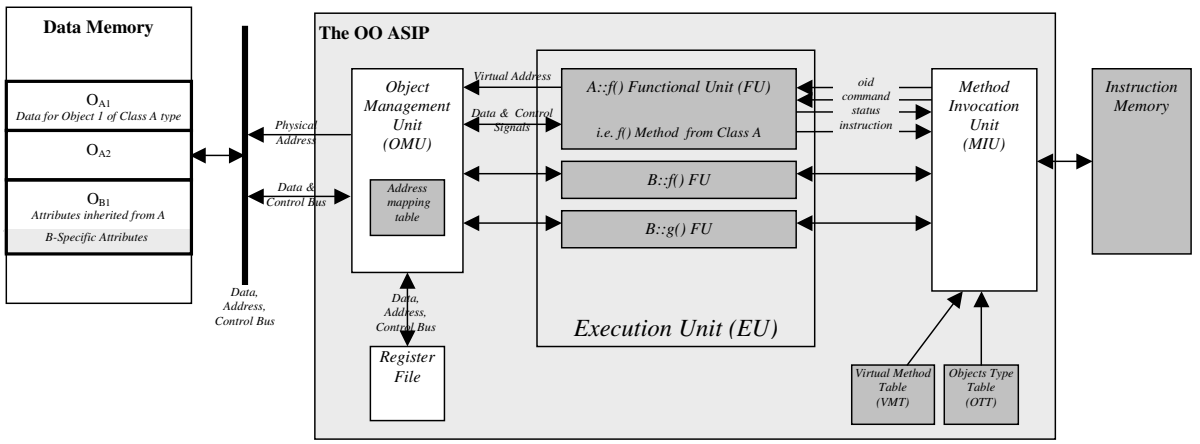

Figure 2: ASIP internal structure for a sample class library (class B is derived from class A while redefining its $f()$ method and introducing $g()$ method.)

of copying them (from RAM to RAM) by keeping them in a central memory addressed by a pointer. The ASIP design methodology presented above is capable of going beyond the ODETTE synthesisable features by allowing variables and attributes to take pointer type. This requires the addition of registers capable of holding pointer values (integer registers may as usual be reused for this purpose) and an indirect addressing mode to allow instructions to take operands from RAM memory locations. This allows a closer fit between hardware and software (clearly hardware designers need to be aware of the serialisation via RAM access forced by using pointers, emphasising to use objects not pointers in hardware.)

3 Synthesising an OO-ASIP

In this paper we focus on synthesis of the OO-ASIP from a *given* "hardware class library". Design of such a class library can be addressed by the MESCAL disciplined benchmarking [Aug02]. The internal architecture of the OO ASIP is shown in Fig. 2. The parts in dark grey are those that depend on the "hardware class library" or the software application. Other parts are invariant and comprise a template for the OO-ASIP. Hardware-class-library-dependant parts are inserted into this template to build an OO-ASIP tailored to the application domain corresponding to the "hardware class library".

Instruction Encoding. A "method-call" instruction consists of an opcode and one or more operands, which respectively correspond to the called method, called object, and call parameters. The important point in instruction encoding is that a virtual method and all its redefinitions must share a common opcode; otherwise it contradicts dynamic-binding because the methods will be statically distinguishable. For example, if class B is derived from A, and $B :: f()$ overrides $A :: f()$, they both must have the same opcode of (say) $0F$. This can be simply done by traversing the "hardware class library" and assigning a new opcode to each *first-ever introduced* method. The methods first introduced in "software class library" will have no hardware counterparts and hence use traditional subroutine call instructions for invocation.

The Method Invocation Unit (MIU). The MIU reads in instructions, executes traditional control-flow ones, and realises polymorphism by dynamic-binding of method-call instructions. Fig. 3 shows our prototype MIU implementing a subset of the JVM (Java Virtual Machine). It uses, as in JVM, the `invokevirtual` bytecode for "method-call" instructions—lines 15 to

26. After dynamic-binding (lines 19-21; detailed in Section 4.1), the MIU either invokes the appropriate hardware module that implements the method and waits for its completion (line 22), or calls the corresponding software routine (lines 23-26).

```
1 SC_MODULE(miu_module) {
2         sc_in_clk clk;        sc_in<bool> reset;
3  // Other declarations and definitions: Ports, parameters, variables, and signals
4
5 void main() {
6   method_id mid;  object_id oid;  class_id cid;  fu_id fu;  opcode_type  opcode;
7
8   pc=0;  ...  // OO-ASIP RESET state
9   while (1) { // Normal operation
10    opcode = instrmem[pc++];
11    switch(opcode) {
12      case IF_ICMPEQ  : if (stack[sp-1] == stack[sp]) pc = pc + instrmem[pc]; else pc++; sp-=2; break;
13      case IADD       : stack[sp-2] = stack[sp-1] + stack[sp]; sp-=2; break;
14      ...  // Any other JVM instructions
15      case INVOKEVIRTUAL: // Method invocation (may be dispatched to either hw or sw)
16        mid = instrmem[pc++];
17        oid = stack[sp];
18          // Polymorphism realisation
19        cid  = OTT[oid];
20        fu = VMT [cid][mid];
21        if (fu.ishw) // Method is in hardware
22          activate_fu(fu.fuid, oid);
23        else {  // Method is in software. 'fu.fuid' shows its starting address
24          stack[sp+1] = pc;  stack[sp+2] = oid;  sp+=2;
25          pc = fu.fuid;
26        } break;
27      default: exception_handling("Unsupported opcode:");
28    } // switch (opcode)
29    wait(); // will wake-up in next clock cycle
30  } // while(1)
31 }// main()
32
33 SC_CTOR(miu_module) {
34   SC_CTHREAD(main, clk.pos());
35   watching(reset.delayed()==true);
36  }
37 }; // miu_module
```

Figure 3: SystemC code fragment for the MIU.

Functional Units. Each *public* method of the "hardware class library" is implemented as a Functional Unit (FU). Unlike traditional processors, each FU can be *active* in the sense that it can contact the OMU–see below–whenever required to access individual object fields.

When the MIU sends the *command* signal to an FU in Fig. 2, the FU is provided with the *oid* of the object to work on, along with other operands of the instruction. Whenever required, the FU can call another method via the MIU (port `instruction` of the MIU in Fig. 2). Recursive calls involving hardware methods are not allowed.

An FU needs to have an interface to the MIU to get invoked, take its method parameters, and return values if any. It also needs an interface to the OMU to read and write object fields. A sample SystemC SC_MODULE template for an FU is shown in Fig. 4. The FU increments the first 32-bit data field of the object (line 14) when the MIU sends the `START` command (line 12). Obviously, the MIU interface varies depending on number and type of arguments of the corresponding method. The C++ code of the hardware method defined by the designer is modified by a preprocessor and inserted in such a SC_MODULE template. The preprocessing converts method calls to appropriate SystemC code to invoke the MIU, and also replaces field-access statements to corresponding OMU communication in SystemC. The processed code is then inserted in a template such as normal-font lines in Fig. 4, resulting in an FU in SystemC.

To synthesise such an FU we rely on behavioural synthesis technology. Obviously, each FU can also be hand-optimised for better synthesis results; however, we build up our approach on behavioural synthesis to leverage current synthesis technology, while our OO synthesis approach adds new features—see Section 4—not accessible in behavioural modelling. We currently use Synopsys *SystemC Compiler*©; hence, the C++ code of a hardware method is expected to conform to the *SystemC Compiler* synthesisable subset.

The Object Management Unit (OMU). The object data is stored in OO-ASIP registers or in main memory. The OMU is responsible for synchronising FUs access to these shared

```
1  SC_MODULE(sample_fu) {
2   // Declarations and definitions: Ports, parameters, variables, and signals
3     sc_in_clk clk; sc_in<bool> reset;
4     ...
5   // Definition of OMU-access routines
6     sc_int<32> OMU_read(object_id, index_type) {...}
7     void OMU_write(object_id, index_type, sc_int<32>){...}
8    ...
9   void main() {
10    status = RESET;
11    while(1) {
12      if (command == START) {
13        status=STARTED;
14        OMU_write(oid, 0, OMU_read(oid, 0) + 1); //actual method functionality
15        status=DONE;
16      } //if
17      wait();
18    }// while
19 SC_CTOR(sample_fu) {
20   SC_CTHREAD(main, clk.pos());
21   watching(reset.delayed()==true);
22  }
23 }; // sample_fu
```

Figure 4: SystemC code for a typical FU corresponding to a method with an `int` argument.

resources. It also provides a locking mechanism for on-demand atomic updates by each FU. It is designer's responsibility to use this facility wherever required; the preprocessor or synthesiser is not to check race conditions. To accelerate data access, the OMU implements caches per FU and takes care of their coherency. The OMU SystemC code only depends on the number of FUs and is generally OO-ASIP-invariant.

4 Synthesis Features

Many OO features (such as templates, abstract classes, and the like) are effectively done as preprocessing by C++ and hence we assume them supported with no effort as long as essential OO features of polymorphism, dynamic method dispatch, and special cases caused by hardware implementation are supported.

4.1 Polymorphism

Polymorphism implies dynamic type checking of the called object to find out the method that corresponds to the instruction. This is done by the MIU through two tables: OTT and VMT. The *Object Type Table (OTT)* shows current class membership of all objects available in the system, i.e. an $oid \rightarrow cid$ (class identifier) mapping. In software realisations, this is normally stored as a (hidden) tag in the object attribute storage. We hold this inside the OO-ASIP for higher performance, but this is not particularly important here.

The *Virtual Method Table (VMT)* is a matrix containing an entry for each class of the system, and for each *public* method, designating its corresponding FU. Hence, it is a mapping function of type $(cid, mid) \rightarrow (FUid)$. Naturally, the row of a derived class is identical to its immediate parent except for its overriden and newly introduced methods.

The MIU consults the OTT to find out the cid of the called object, and then maps the (cid, mid) to an $FUid$ through the VMT. The resulting $FUid$ is not necessarily a hardware unit but may point to the starting address of a software routine; however in either case, it is the appropriate method of the called object and is invoked accordingly, effectively realising *polymorphism*.

It is worth considering how the *virtual* method dispatch mechanism works as the source and destination of the call vary between hardware and software implementation. Non-virtual calls are just a special case where we know the implementation format of the callee.

First assume all actual arguments are put on a stack before each method call. This provides the same mechanism for parameter passing irrespective of the caller and callee being in hardware

or software. Given a software caller, i.e. a software method `f()` which invokes a call `obj.g()`, `g`'s arguments are pushed on the stack. The `g` component of the VMT (we assume the destination class to be A) for `obj` is then extracted. This is then (according to a tag bit) either interpreted as a software method (in which case the call effects a traditional branch-and-link instruction) or as a hardware method in which it effects an OO-ASIP instruction A`::g`. The same would hold for a hardware caller.

Now assume registers are used instead of a stack to gain higher performance. For a software caller, the scenario is the same as above, but the parameters are placed in assumed registers `a1`, ..., `a`n. Given a hardware caller, with notation as above, a hardware destination just corresponds to the activation of the appropriate FU as above (and the movement of arguments to the callee's input register). A software destination is more tricky as the call arguments (held internally in the caller's FU) must be routed to the input registers (`a1`, ..., `a`n as above) for `g()`.

It is worthy of note that the register-based calling scheme above means that each method declaration (and all its overriding instances—i.e. a *mid*) uses a common set of argument registers, but separate *mid*s use separate argument registers. This simplifies the software/hardware interface for presentational purposes; in practice we would seek to use a single set of argument registers (cf. procedure call on MIPS or ARM), but doing so is more complicated and puts additional requirements for a software callee to save argument registers before using a hardware instruction which could use a software method. This is not a concern when using a stack for parameter passing.

Overriding Hardware by Software. Polymorphism allows the designer to uniformly view and treat all objects of a certain class and all its derivatives, and rely on the run-time virtual-method binder to invoke the appropriate method. Apart from the above abstraction not provided by other modelling and synthesis technologies, such as behavioural modelling and synthesis, polymorphism enables outdated or malfunctioning hardware parts to be replaced by software implementations requiring no change in hardware. This also allows post-manufacturing repair of chip designs that are found to have some design errors. Such designs would otherwise need a very long redesign and remanufacturing cycle which could result in loss of part or even the whole market window in today's rapidly shrinking time-to-market. The MIU in our OO-ASIP is capable of dispatching the virtual methods to software, and hence, the designer only needs to use objects with overriden correct methods.

4.2 Pointer Synthesis

The ASIP synthesis methodology we present is capable of going beyond the ODETTE [Gri02b] and OASE [Kuh01] synthesisable constructs by allowing variables and attributes to take pointer type. The integer registers in the OO-ASIP can hold object addresses and hence provide pointers to objects. This is also required for polymorphism as we follow the C++ approach to polymorphic objects, which implies having a pointer to a base class.

4.3 Dynamic Object (De)Allocation

Many researchers have implemented objects as hardware modules. Obviously, this implies an inability to dynamically (de)allocate objects unless a reconfigurable hardware platform is exploited. Breaking an object to a separate data-storage and an operations engine, and routing the data to the engine on demand, (as done in software implementations of OO models) allows us to only need to dynamically (de)allocate the storage, and not the engine, when (de)allocating objects. This storage can be either in internal OO-ASIP registers or the memory (trade-off

between access speed and capacity) while the management can be done in hardware or software (trade-off between (de)allocation speed and hardware complexity). However in any case, to (de)allocate objects the *OTT, VMT,* and *OMU mapping-table* need to be accordingly updated.

4.4 Recursion

As in ODETTE, not all class hierarchies can be synthesised as hardware [Gri02b]. For example a recursive method for factorial might well be acceptable in a software partition but not as hardware. The well-formedness rule we currently impose is: *no cycle in the call-graph contains a hardware method* (otherwise valuable state would be lost). Note that we can cope with "software calls hardware calls software" and also "hardware calls software calls (different) hardware" not involving recursion (and hence bounded) by techniques similar to those employed to allow multiple levels of interrupt in hardware.

4.5 Dynamic OO-ASIP Evolution in Hardware

Incorporating some FPGA blocks inside the chip enables dynamic evolution of the OO-ASIP to include more method implementations and hence new classes in the same way as it could evolve through software. This implies new hardware ports in the MIU and the OMU which can be pre-manufactured for a pre-determined degree of expandability. Other than configuring the FPGA, only the VMT needs to be updated (as in evolution-by-software) to reflect method implementations for the new classes.

5 Language Design Issues and OO-ASIP Co-Design

We have presented this work from a C++ perspective; while this offers certain expressivity advantages, seemingly harmless early design decisions later favour implementation of components either as hardware or software. Particular problematic aspects we wish to identify are (*i*) name visibility, (*ii*) passing of object values to and from methods (including aliasing issues and particularly the '`this`' pointer), and (*iii*) concurrency.

Name visibility. Firstly, let us observe that C++ is well-adapted to describing statically allocated storage and operations on it (this substantiates our claim earlier that the `int` and `float` values may be considered as built-in classes). For example a 2-address instruction-format integer CPU could be effectively described by

```
class Int { private: word32 bits;
            public: void add_r(Int v) { this->bits += v.bits; }
            public: void add_k(word32 k) { this->bits += k; }
            public: void sub_r(Int v) { this->bits -= v.bits; }
            // etc etc
          }
Int r0, r1, ..., r31;
```

merely assuming a type of 32-bit data words and writing `+=`, and `-=` not as code, but to summarise the intended effect. In this framework typical assembly language instructions might well be expressed as `r8.add_r(r7);` instead of the equivalent traditional `add r8,r7`.

It is tempting to believe that writing C++ in such a style transfers directly to hardware and software components of an OO-ASIP, but the boundary between hardware and (later developed) software means that the 'whole program' is not available at time of OO-ASIP synthesis. Let us

take the above integer CPU as a concrete example. In ODETTE the whole program is available to the compiler and therefore, by whole-program analysis, we can find all the uses of `r0`, ..., `r31` and hence generate effectively one data-path for each user; we can also inline-expand operations like `r8.add_r(r7);` so that `r8` can be implemented as a simple register and does not have to be addressable as would otherwise be required by the implicit address-taking implied by C++'s `this` pointer. By contrast, in our OO-ASIP approach the software classes are in general not available at the time the hardware classes are to be synthesised. Hence there is no restriction on software classes, for example, taking the addresses of statics `r3` and `r4` and then passing them to a method which updates one or the other. This is highly undesirable as it would mean that our simple static variables `r0`, ..., `r31` must be allocated to addressable memory instead of simple registers *just in case* software took their address. One partial solution would be to make `r0`, ..., `r31` to be `private` members of `Int`; however this then means that the instruction form `r8.add_r(r7);` would not be expressible, and instead only multiple parameterless static methods could be used, as if `class Int` were of the form:

```
class Int { private: static word32 r0, r1, ..., r31;
            public:  static void add_r8_r7() { r8 += r7; }
            public:  static void sub_r1_r7() { r1 -= r7; }
            // etc etc
          }
```

This is actually a reasonable solution from the hardware synthesis perspective, in that parameterless and result-less static methods only require a single-bit wire for their activation. However, rather than requiring such extreme styles to be selected by wholescale code changes (which prejudges the hardware/software choice) it would be desirable to have a finer granularity manner in which to express usage constraints on a class, for example an "export-to-software-for-reading-only" pragma for a variable. Overall, what we need is a design flow which allows 'vanilla' object-oriented designs to be coded naturally, with separate directives or embedded pragmas which transform between styles optimised for hardware and those optimised for software.

Object Aliasing. The second issue concerning C++ arises from its programming language heritage, and particular aliasing. Aliasing (allowing one object to be accessed by two or more references) is a natural way of providing multiple accesses to an updatable object in software; however from the hardware point of view it tends to imply at least multiplexers and in general a RAM-like addressing structure with the concomitant sequentialisation. To a very large extent, use of aliasing is built into C++ at the fundamental level, in that the implicit `this` argument to a method is of pointer type (and therefore many objects are implicitly address-taken, and thus risk being allocated to RAM-like storage instead of registers). We might refer this to passing `this` *by reference*. However, there is an alternative view of objects (stemming back at least to Hoare's seminal work [Hoa72]). There a method updating an object creates a new object which can be passed back to the caller: this is similar to the ideas of passing parameters *by value or result*—the familiar `in`, `out` or `inout` choice. This can often be more effective at the hardware level in that a smallish object can reside in registers and merely be copied from one to another, instead of being addressable. Clearly, *at the extremes* both hardware and software prefer to pass small objects by value, and large objects by reference. One problem with C++ is that it forces early selection between these modes and indeed favours call-by-references because of the `this` model. Consider a system which performs pipeline-style operations on an object. In software we would probably place the object in memory and pass a reference on to each stage; in hardware we would generally prefer to pass the object between stages by value. In work related to this project Ennals, Sharp, and Mycroft [Ena03] consider a type system which allows objects to be passed along a pipeline safe from potentially concurrent aliased access. To summarise,

again what we need is a design flow which allows 'vanilla' object-oriented designs to be coded naturally, and then specialised with a single pragma instead of early binding.

A further language issue arises about polymorphism and our requirement to be able to add further software classes after the OO-ASIP has been designed, particularly if we wish to pass objects by value. Assume a class hierarchy

```
class A { virtual void m(); ... };
class B:A { ... };
class C:B { ... };
class D:A { ... };
```

In the C++ *by reference* interpretation of `this`, all variants of `m()` (either in class `A` or overriding versions) will require a pointer to the invoking object. The pointer will remain constant in size, but the size of object it points to will vary (according to class `A`, ..., `D`) but will share a common layout for initial segments; as is well-known from software, this is not problematic. Now consider passing an object to or from `m()` by value. In ODETTE's SystemC-Plus a value type which can hold all values of classes inheriting from class `A` is denoted `PolyObject(A)`. In ODETTE the whole program is known, so the number of bits required to hold a value of `PolyObject(A)` is just the largest size of any class which inherits from `A`. In our OO-ASIP setting in ODYSSEY, this is not sufficient, in that arbitrarily large software classes may later inherit from `A`. What we need is a linguistic form which allows a variable to have type of (say) class `A`, `B`, `C` but (say) not class `D` nor of any other class inheriting from these. (This might be appropriate for some internal queue of elements of type `A`, `B` or `C`.) Thus we need the expressiveness to say, for example, `PolyObject(A-C)`. Note that Java's keyword `final` is *not* what we want; this stops inheriting from a class altogether—what we want is just to constrain the values reaching a particular piece of hardware.

Concurrency. The third issue, which we have rather avoided in this OO-ASIP presentation so far, is *concurrency*. Clearly hardware is potentially very concurrent, but C++ is a very sequential language with the language definition specifying very precisely the serialisation of various operations (see "sequence point"). The combination of hard-to-analyse aliasing and very specified serialisation means that it is very hard to extract significant concurrency from C++ programs; worse still a small source change can produce a large change in the result of any "available concurrency" analysis. Yet again what we need is a design flow which allows concurrent 'vanilla' object-oriented designs to be coded naturally, and then specialised with a single pragma (e.g. sequentialise for software, or run concurrently for hardware methods) instead of early binding of object oriented concepts to temporal ordering.

Ideal language requirements. To conclude this section, let us emphasise the need for a design flow from a high-level object-oriented design at a UML-like level, principally to a software/hardware-neutral algorithmic design language. This language may contain pragmas or other localised directives to indicate hardware or software implementation, but not the wholescale distributed information which would be encountered in a C++ or HDL representation; it should be directly mappable to C++ or an HDL in a way which distributes the information automatically rather than in the course of program development. Specific aspects in which the software/hardware-neutral language should differ from C++ are additional specification techniques for: visibility, aliasing and sequencing/concurrency.

6 Summary and Conclusion

The main thrusts of this paper are firstly to identify a class library with an ASIP; secondly to show how a larger (hardware+software) class library can be partitioned into an OO-ASIP and a program for it; and finally to detail the corresponding synthesis and discuss issues in hardware/software co-design implied by the OO perspective. We have described a novel approach to synthesising object-oriented models which enables object-oriented hardware/software co-design with close coupling between the hardware and software parts. This is done, for the first time to the best of our knowledge, by synthesising an object-oriented ASIP from the "hardware class library" of the OO model. Practical benefits include:

- enabling the hardware to evolve through software. This enables post-manufacturing overriding of outdated or faulty-designed hardware by correct software implementation.
- providing an invariant platform capable of realising additional applications modelled with the "hardware class library" from which ASIP was synthesised. System design by such ASIPs seem to be the philosophy of choice in the near future [Keu02].
- using a single linguistic framework (C++/SystemC) for both hardware and software components throughout the system design flow, thereby simplifying late-in-the-design-process allocation of methods into hardware or software.

We wish to view our ODYSSEY ASIP approach as one extreme in implementing OO models in hardware; we use only one implementation per method. The other extreme would be the Radetzki and ODETTE approach of implementing N modules for N objects in the system. We observe that the ODETTE approach offers highest possible concurrency in inter-object method invocations, but also causes significant area overhead. On the other hand, ours uses the minimum area, but sadly implies sequential method invocation. Between these two extremes, there is a spectrum of design points where the number of method implementations can vary from 1 to N for N objects in the system. We observe that the optimum answer is often somewhere in this design space constrained by the available concurrency in the OO model, and the available area on the chip (or power allowed to dissipate). Exploring this design space is our next step in OO hardware design and synthesis. The architecture we currently envisage is a network of OO-ASIPs each handling a subset of all objects in the system.

We also intend to extend this work to OO design of embedded systems where software is an essential component. Programming the OO-ASIP is hence of high importance. Currently, we have chosen a very small subset of the JVM and have implemented it in the MIU to execute general software as well as method-call `invokevirtual` instructions. As work under investigation and development, we intend to customise the GCC-3.1 compiler-suit support for picoJava processor [Sun03] to our subset of the JVM.

This is the first milestone, introducing OO-ASIPs as the building block, of the work in progress in the context of the ODYSSEY project [ODY03].

Acknowledgements

We wish to thank the British Council for funding the first author's visit to Cambridge, and also the Ministry of Science, Research, and Technology of the Islamic Republic of Iran for a partial scholarship. This work was partly supported by (UK) EPSRC grant GR/N64256.

References

[Ash97] Ashenden, P.J., Wilsey, P.A., Martin, D.E., *SUAVE: Painless Extension for an Object-Oriented VHDL*, Proc. VHDL International Users Forum (VIUF, Fall Conference), 1997.

[Aug02] August, D.I., Keutzer, K., Malik, S., Newton, A.R., *A Disciplined Approach to the Development of Platform Architectures*, Microelectronics Journal, Elsevier, 2002.

[COR03] *CORBA: Common Object Request Broker Architecture*, `http://www.corba.org`

[Ena03] Ennals, R., Sharp, R.W. and Mycroft, A, *Linear Types for Packet Processing*, Draft manuscript, see `http://www.cl.cam.ac.uk/users/am/papers`

[Gri02a] Grimpe, E., Oppenheimer, F., *Aspects of Object Oriented Hardware Modelling with SystemC-Plus*, System-on-Chip Design Languages, Extended Papers, Best of FDL'01, HDLCon'01, 2002.

[Gri02b] Grimpe, E., Timmermann, B., Fandrey, T., Biniasch, R., Oppenheimer F., *SystemC Object-Oriented Extensions and Synthesis Features*, Forum on Design and Specification Languages, 2002.

[Hel97] Helaihel, R., Olukotun, K., *Java as a Specification Language for Hardware-Software Systems*, Proc. International Conference on Computer Aided Design (ICCAD), 1997.

[Hoa72] Hoare C. A. R., *Proof of correctness of data representations*, Acta Informatica, 1, 1972.

[ITRS01] *International Technology Roadmap for Semiconductors (ITRS)–Design*, 2001. `http://public.itrs.net/Files/2002Update/2001ITRS/Design.pdf`

[Keu02] Keutzer, K., Malik, S., and Newton, A. R., *From ASIC to ASIP: The Next Design Discontinuity*, Proc. International Conference Computer Design (ICCD), 2002.

[Kuh01] Kuhn, T., Oppold, T., Schulz-Key, C., Winterholer, M., Rosenstiel, W., Edwards, M., and Kashai, Y., *Object-Oriented Hardware Synthesis and Verification*, Proc. International Symposium on System Synthesis (ISSS'01), 2001.

[MES03] *The MESCAL Project: Modern Embedded Systems, Compilers, Architectures, and Languages*, `http://www.gigascale.org/mescal/`

[ODY03] *The ODYSSEY Project: Object-oriented Design and sYntheSiS of Embedded sYstems*, `http://www.cl.cam.ac.uk/users/mg342/odyssey`

[Rad00] Radetzki, M., *Synthesis of Digital Circuits from Object-Oriented Specifications*, PhD Thesis, University of Oldenburg, 2000.

[Sch95] Schumacher, G., Nebel, W., *Inheritance Concept for Signals in Object-Oriented Extensions to VHDL*, Proc. EURO-DAC with EURO-VHDL, 1995.

[Sun03] Sun Microelectronics, *picoJava Microprocessor Cores*, `http://www.sun.com/microelectronics/picoJava/`

[Wol96] Wolf, W., *Object-Oriented Co-synthesis of Distributed Embedded Systems*, ACM Transactions on Design Automation of Electronic Systems (TODAES), July 1996.

[You98] Young, J.S., MacDonald, J., Shilman, M., Tabbara, A., Hilfinger, P., Newton, A.R., *Design and Specification of Embedded Systems in Java Using Successive Formal Refinement*, Proc. Design Automation Conference (DAC), 1998.