Overhead-free Polymorphism in Network-on-Chip Implementation of Object-Oriented Models

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

Alan Mycroft[2]
[2] *Computer Laboratory*
University of Cambridge, Cambridge, UK
alan.mycroft@cl.cam.ac.uk

Abstract

We unify virtual-method despatch (polymorphism implementation) and network packet-routing operations; virtual-method calls correspond to network packets, and network addresses are allocated such that routing the packet corresponds to dispatching the call. As the run-time routing structure is inherent in Network-on-Chip platforms, this unification implements polymorphism for free.[1]

1. Introduction

Object-oriented methodology is an approach to narrowing the design productivity gap and to decreasing time-to-market by increasing the design abstraction level. In OO design methodology, polymorphism (and the associated virtual method despatch) is a key feature. However, it implies run-time support for dynamic binding of operations; this imposes performance, area, and power overheads to the design that is known as the major drawback of OO hardware synthesis.

On the other hand, the technology trend in System-on-Chips is toward Network-on-Chip paradigm due to deep submicron effects [1]. These on-chip networks imply routing infrastructures for run-time packet-switched communication. We take advantage of this NoC inherent structure and show how method-calls can be dispatched at the same time that packets are routed. This realises polymorphism with no circuitry additional to that already required for routing in the NoC, and hence introduces no overhead.

2. Polymorphism in a Network-on-Chip

We follow an ASIP approach and realise OO models as cooperating hardware and software [2]. The internal architecture of the chip is shown in Fig. 1. Those methods of the class library that are to become hardware are each implemented as a *Functional Unit* (FU); the other methods are implemented as software routines in the traditional processor(s); objects data are stored in a shared data memory and all FUs and processors have cached-access to it through the Object-Management Unit.

In this paper, we wish to view virtual-method calls as packets sent over the network from the caller module to the called one, carrying the parameters of the call as the packet data payload. The return-value(s), if any, is also sent in another packet from the callee back to the caller. We assign the FU addresses and object numbers such that routing of a packet is equivalent to dynamic binding of the corresponding virtual-method call.

2.1. FU network-address assignment scheme

We assign a unique bit-field identifier *cid* to each class in the system model, and another unique identifier *mid* to the methods of each class, provided that overridden methods in a derived class use the same identifier as in the parent class.; i.e., if class *B* is derived from *A* and overrides *A::f(args)* by *B::f(args)*, both *f(args)* methods share the same identifier M.

Since an FU corresponds to a certain method of a certain class, we assign *FUid* = <*mid.cid*> as its network address where the "." operation represents a bit-field concatenation.

2.2. Object numbering scheme

Objects are assigned a number *objn,* unique among all objects of the same class; consequently, *oid* = <*cid.objn*> (*cid* shows the class identifier) distinguishes each object from others; e.g. the objects of a class *A* numbered 1 will be numbered 1.1, 1.2, 1.3, etc.

2.3. Method dispatch without polymorphism

When the type of the called object is known at compile time, method dispatch will be straightforward since the receiver is statically known; e.g, *a.f(params)* in C++ is statically known to invoke *A::f(args).* We first present our scheme in this case, and then extend it to polymorphism where the type of the called object is not statically known.

We view each method call as a network packet. Each

[1] 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.

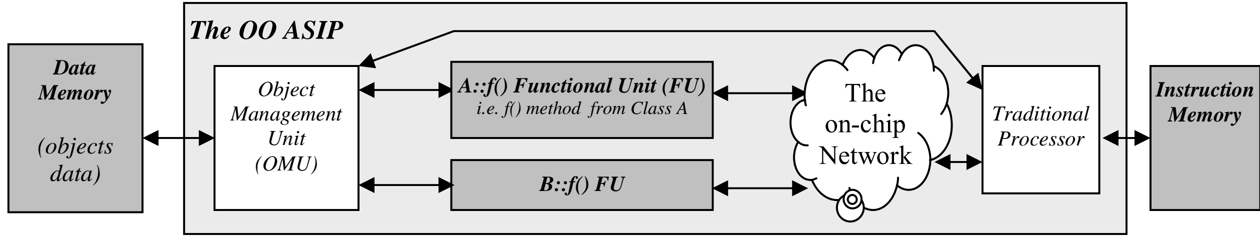

Fig. 1 Internal structure of the ASIP with an on-chip network.

method call is identified by a method, an object, and the parameters of the call; hence, the bit-field concatenation of these items represents the method call and comprises the packet to be sent; i.e. *<mid.oid.params>*.
The *oid* can be expanded to *<cid.objn>*. Hence, the packet becomes: *<mid.<cid.objn>.params>*
or simply *<mid.cid.objn.params>*
or *<<mid.cid>.objn.params>* with a change in viewpoint, or finally *<FUid.objn.params>* when observing that according to our FU-address-assignment, *<mid.cid>* designates an *FUid* which is conveniently the unit that must handle the method call and hence is the destination of the packet.

In other words, a call to *oid.mid(params)* is equivalent to a network packet *<mid.oid.params>* which is routed for no extra cost to the FU with the address of *FUid* while the *objn* and *params* respectively represent the object to work on and the parameters of the call. So, the packet looks like:

<table>
<tr><td colspan="3">Header</td><td colspan="2" rowspan="2">Data payload</td></tr>
<tr><td>Other headers</td><td colspan="2">destination address</td></tr>
<tr><td>…</td><td>method-id</td><td>class-id</td><td>objn</td><td>method parameters</td></tr>
<tr><td></td><td></td><td colspan="2">object-id</td><td></td></tr>
</table>

The grey part shows the packet fields. In addition to *method-id*, *class-id,* and *objn*, other headers may also be required, e.g. the source address to allow the called FU to return a result if required. The *objn* field and method *parameters* are sent as the data payload of the packet.

Example 1: Suppose that an OO model defines a class *A* with method *f(args)*. We assign number 1 to the class and the unique number M to the *f(args)* method. Our network-address assignment scheme suggests *<mid.cid>* for the corresponding FU, resulting in M.1 for *A::f(args)*. Defining *a7* as an object of class *A,* our object-numbering scheme suggests *<cid.objn>* or 1.7, for example, as the identifier of the *a7* object. Now, calling *a7.f(params)* corresponds to the following packet:

…	M	*1*	*7*	*params*

The packet destination address is M.1 (see the above packet format), and hence when sent over the network, the routing structure conveys it to its destination address, M.1, which corresponds to *A::f(args)* FU as expected.

2.4. Method dispatch with polymorphism

Polymorphism is expressed in various ways in different languages. We follow a C++-like approach by allowing method-calls on pointer-to-objects; polymorphism implies the pointer may point to different objects (constrained by the class hierarchy) at run-time. However, unlike C++ we use the object numbers (see Section 1.2) to represent pointers instead of memory addresses of objects.

The previous section showed how a packet is assembled for a statically known call. To implement polymorphism, we simply put the run-time value of the pointer in the *object-id* part of the packet (which overlaps both header and payload) and send it on the network as before; depending on the dynamic pointer value, the packet may reach different FUs, but in all cases it will be the appropriate one due to the FU address assignment scheme.

Example 2: Suppose that one derives two classes *B* and *C* from class *A* in Example 1, such that both of them override *A::f(args)* and all implement as hardware FUs. Assume that we assign numbers 2, and 3 respectively to the classes, and hence, the three FUs are numbered M.1, M.2, and M.3 respectively (recall that all three *f(args)* methods use the same identifier M). Further assume that the system model defines only one object from each class, respectively named *a, b, c* and numbered 1.1, 2.1, and 3.1.

A pointer to class *A* (e.g. *ap*) may dynamically point to *a*, *b*, or *c*; i.e. contain 1.1, 2.1, or 3.1 respectively. So, calling *ap->f(params)* should be dispatched to *A::f(args)* or *B::f(args)* or *C::f(args)* depending on *ap* run-time value. Assembling a packet with the *ap* value results in <M*.1.1.params*> or <M*.2.1.params*> or <M*.3.1.params*> packets that are routed to M.1 or M.2 or M.3 respectively corresponding to *A::f(args)* or *B::f(args)* or *C::f(args)* FUs. This is what polymorphism implementation implies.

Prototype implementation: We used SystemC for a prototype implementation. Its simulation results confirm correctness of the scheme and its implementation.

References

[1] L. Benini, G. DeMicheli, Networks on Chips: A New SoC Paradigm. In *IEEE Computer*, Volume: 35 Issue: 1, pp. 70-78, 2002.

[2] M. Goudarzi, S. Hessabi, and A. Mycroft, Object-Oriented ASIP Design and Synthesis. In *Forum on Design & specification Languages (FDL'03)*, Frankfurt, Germany, 2003.