Towards a Generic Programming Model for Network Processors

Kevin Lee, Geoff Coulson, Gordon Blair, Ackbar Joolia, Jo Ueyama

Lancaster University, Lancaster, LA1 4YR, UK

***Abstract*— Network Processors (NPs) are emerging as cost effective networking elements that can be more readily updated and evolved than custom hardware or ASIC-based designs. Moreover, NPs promise support for run-time reconfiguration of low-level networking software. However, it is notoriously difficult to develop software for NPs because of their complex design, architectural heterogeneity, and demanding performance constraints. In this paper we present a runtime component-based approach to programming NPs. The approach promotes conceptual uniformity and design portability across a wide variety of NP types while simultaneously exploiting hardware assists that are specific to individual NPs. To show how our approach can be applied in a wide range of types of NPs we characterise the design space of NPs and demonstrate the applicability of our concepts to the various classes identified. Then, as a detailed case study, we focus on programming the Intel IXP1200 NP. This demonstrates that our approach can be effectively applied, e.g. in terms of performance, in a demanding real-world NP environment.**

I. Introduction

Network Processors (NPs) are an attempt by hardware vendors to fulfill the growing need for low-priced specialised network hardware that is more future proof than conventional custom hardware or ASIC-based designs, and can be applied in a wide range of situations (e.g. in networked devices, as edge-network routers and even in the network core). In addition, NPs are seen by some as potential vehicles for the deployment of active networking-derived technologies [1] which exploit the potential of NPs for run-time software reconfiguration. Conceptually, NPs are multiprocessor-based hardware units that support a number of network ports and provide software-based packet processing facilities that can be programmed with the aid of a toolkit. They have the ability to perform relatively complex packet processing at line speeds.

There is a downside to current NPs, however: they are notoriously difficult to program [2], [3]. This is because of their complex design (e.g. involving multiple processors, including both general purpose and specialised processors; and multiple memory and interconnect technologies), extreme architectural heterogeneity across vendors and products [4], and demanding performance constraints.

Therefore, NPs often exhibit elegant hardware designs which remain underexploited by software [5]; and their extreme heterogeneity tends to inhibit translation of software, software designs, or even skills across brands. The problem is exacerbated by the need for high performance and runtime reconfiguration, both of which add considerably to software complexity (because of its complexity, many NP software toolkits fail to provide any support at all for runtime reconfiguration).

The aim of the research discussed in this paper is to develop a generic programming model for NPs that accommodates complex architectures and architectural heterogeneity while also supporting design portability, high performance and runtime reconfigurability. Our approach is based on a run-time software component model. The model promotes conceptual uniformity and design portability across a wide variety of NP types while simultaneously exploiting hardware assists that are specific to individual NPs. It features a distributed runtime with low memory footprint, employs formally specified interfaces, supports components written in different programming languages, and uniformly abstracts over different processor types and different inter-processor communication mechanisms without loss of performance. It also explicitly supports run-time reconfiguration of software.

The remainder of the paper is structured as follows. In section II, we characterise the design space of NPs as a basis for arguing the genericity of our approach, and also survey a number of existing programming models provided both by the manufacturers of various NP products, and by independent researchers. In section III, we present our approach to programming NPs and show how this improves on existing approaches. Then, in section IV, we provide a detailed case study of the application of our approach to the Intel IXP1200 NP. Finally, in section V we offer our conclusions.

II. Network Processors

A. Classification

The field of NPs is notable for its great architectural heterogeneity. In general, however, it can safely be said that NPs universally provide programmable support for processing packets, and that this usually takes the form of one or more *packet processors*. These can be supported either on a single chip or across multiple chips. In addition, NPs universally support a number of MAC-level ports, some memory, and some form or forms of processor interconnect.

In this section we attempt to capture the design space of NPs in terms of a small number of orthogonal dimensions. These are:

- the packet processor dimension - the range of types of available packet processors

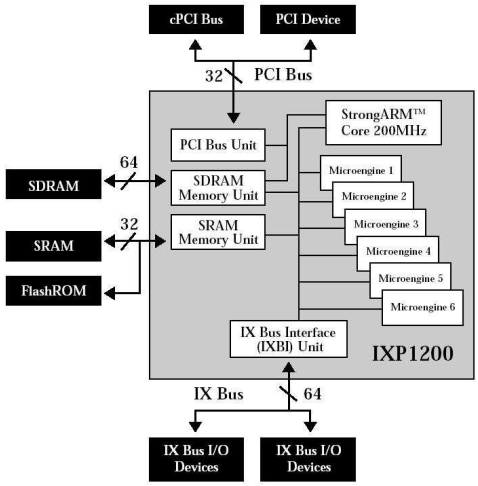

Fig. 1. The Intel IXP1200 (from [7])

- the interconnect dimension - the range of interconnect technologies employed
- the control dimension - the degree of support for centralised control

We then go on to show how prominent NP products map to this space. In so doing, we lay the groundwork for a discussion on how our component-based programming approach can accommodate the full diversity of NPs.

1) The Packet Processor Dimension: Most NPs feature multiple packets processors, but the nature of these can vary from units with very general instruction sets to single-purpose dedicated units for, e.g., checksumming or hashing, which are not programmable. Furthermore, some NPs feature only one type of packet processor and others support a number of different types.

For example, the Intel IXP1200 NP [6] (see figure 1) supports a uniform set of six so-called *microengines* which serve as packet processors. These are 233-600Mhz CPUs whose instruction set includes I/O to/from MAC-ports, packet queuing support, and checksumming. They support hardware threads with zero context switch overhead and can be programmed either in assembler or C. The IXP1200 also includes a general purpose StrongARM CPU which serves as a controller and also typically performs slow-path operations.

On the other hand, the Motorola C-Port [8] employs so-called *channel processors* which are generic packet processors grouped in sets of four that share an area of fast memory. But in addition it supports a range of dedicated, non programmable, processors that perform functions such as queue management, table lookup, and buffer management.

As a third example, the EZChip NP-1 [9] has no fully generic processors. Rather, it exclusively employs dedicated packet processors that perform specific tasks such as parsing packets, table lookup or packet modification. Although these are dedicated to their given 'domain', they are quite flexible and programmable within that domain.

2) The Interconnect Dimension: Different NPs provide different mechanisms for inter-processor communication: such as shared registers, buses (of varying types), shared memory (perhaps a range of types that make different trade-offs between capacity and speed), and dedicated channels.

For example, the IXP1200 provides a fast bus for communication between its microengines, MAC ports and memory. It also provides shared registers and a range of memory types (i.e. SRAM, SDRAM). The shared registers and memory are typically used together at the software level for inter-processor communication. The newer IXP2400 NP from Intel also provides 'next-neighbour' registers that provide a dedicated interconnect between two 'adjacent' microengines.

The Motorola C-Port employs shared fast memory for interconnection between grouped channel processors (as mentioned above). It also employs multiple onboard buses for communication between these groups, and shared memory that is managed by a dedicated processor.

Unlike the two examples above, the EZChip offers a very static and limited interconnect which arranges the packet processors in a strict pipeline topology. The Cisco PXF [10] uses a variant of this approach: it offers multiple parallel pipelines and some capability for communication between pipelines. Clearly, these architectures are less flexible, although potentially faster, than the bus-based interconnects discussed above.

3) The Control Dimension: Apart from the genericity/specificity of their packet processors, different NPs make different choices regarding centralisation/ decentralisation of control and management. For example, some NPs rely exclusively on external control in the form of a host workstation. Others (e.g. the IXP1200) incorporate a commodity CPU on the NP itself which runs an operating system, and others support sufficiently powerful and general packet processors that any of these can potentially serve as a locus of control and management.

The IXP 1200's on-board StrongARM CPU runs a commodity OS such as Linux. As well as handling slow-path packet processing, the StrongARM is responsible for loading code onto the microengines and stopping and starting them as required.

The Motorola C-Port, on the other hand, has no built-in centralised controller. Instead, it relies on a host workstation to load and supervise the operation of its 'channel controller' packet processors. Nevertheless, it is theoretically possible to dedicate one of the channel controllers to take the supervisory role, especially if fine-grained dynamic reconfiguration of the NP is a goal.

Similarly, the EZChip relies on a host workstation for control and management. In this case, there is no alternative because dedicating one of the packet processors, even if possible (cf. their lack of generality), would introduce an unacceptable bottleneck in the pipeline.

B. Software for Network Processors

The provision of software development environments for different NPs is almost as diverse as NP hardware architecture.

In this section we examine both proprietary and research-derived programming environments and show that each is hard to generalise beyond the specific architecture at which it is targeted.

In terms of proprietary software, we focus on programming models and development environments for the IXP1200 and the IBM PowerNP. Information on the software environments used by other NPs is unfortunately hard to obtain without signing non-disclosure agreements.

Intel's *MicroACE* [11] is targeted at the IXP1200 and other Intel IXA products. In this model, proxy-like software elements (called *active computing elements* or ACEs) on the IXP1200's StrongARM control processor are 'mirrored' by blocks of code (called microblocks) that run on microengines. Thanks to this mirroring, when the programmer loads a StrongARM element, the corresponding microblock is transparently loaded onto a microengine as a side effect. The microblock can choose to offload packets to its associated ACE for handling on the slow path.

Although it provides a useful degree of abstraction, the MicroACE approach is limited to IXP1200-like architectures that employ a tightly integrated control processor. Furthermore, the model leaves linkages between microblocks implicit in the way the microblocks are written: is not possible to combine microblocks in unanticipated topologies or to exploit interconnect mechanisms other than those explicitly chosen by the microblock author. Also, the ACE approach cannot be used to perform dynamic software reconfiguration as it takes no heed of the *integrity* of a running configuration: if a component is replaced, a neighbouring component will inevitably fail as components expect to interact directly.

Teja NP [12] is another commercial product targeted at the IXP1200, although it also runs on the IBM PowerNP series [13] which is very similar architecturally to the IXP1200. Rather than offer an abstract programming model like MicroACE, Teja focuses on the provision of an integrated tool chain and development environment. Although this eases the development of NP software it provides minimal architectural abstraction and therefore minimal design portability.

Turning to research-derived programming environments, *NetBind* [14] provides the abstraction of a set of packet-processing components that can be bound into a data path. This is done by adopting the convention of a standard entry and exit instruction sequence for microblocks, and offering the capability to dynamically 'morph' jump instructions in these sequences so that execution is transferred to the entry point of the microblock to be executed next. This separates the raw functionality of a microblock from the way it is composed with others, and also gives the NetBind programmer the ability to dynamically reconfigure compositions of microblocks.

NetBind goes beyond MicroACE in supporting flexible composition of microblocks, but it offers no abstraction over the NP's interconnects or over different sorts of processors (e.g. the microengines, StrongARM, and workstation host of an IXP1200-based router). It therefore offers no more design portability across different NPs than MicroACE.

NP-Click [15] is another component-based programming model for NPs; it is derived from an earlier PC-based software router model called Click. Again, NP-Click has been primarily targeted at the IXP1200. It is founded on a much richer model of components than NetBind. While communication between NetBind microblocks takes place over low-level untyped entry and exit points, Click components have typed *ports*; and connections between ports can be designated as either 'push' or 'pull' which provides declarative control over flow of control and threading. In addition, NP-Click abstracts, to a degree, over the different memory technologies offered by the IXP1200 by providing keywords such as 'global', 'regional' or 'local' which cause the associated component to be automatically allocated an appropriate memory type. Furthermore, it provides low level abstractions such as *malloc()* and *free()* to facilitate and manage the allocation of NP resources such as microengine LIFO registers.

NP-Click does a much better job of abstracting NP architecture than NetBind, but it still falls short of providing a generic approach to NP programming. While it abstracts particular features of the IXP1200, it has no notion of abstracting arbitrary architectures in a principled way, and thereby encouraging design portability and transferable skills across NP types. That is, there is no necessary commonality between the abstractions provided over different architectures (e.g. NPs other than the IXP1200 may not use LIFOs). In addition, NP-Click provides no support for dynamic reconfiguration.

VERA [16], [17] is an extensible software router architecture that comprises three layers: the top layer provides the abstraction of a router, the bottom layer abstracts the hardware, and a 'distributed operating system' layer mediates between the two. The distributed operating system layer organises the available packet processors into a hierarchy of levels. Initial classification occurs on a 'low level' processor attached to the MAC-port, and if the packet requires further or more complex processing then a 'higher level' processor is used. This provides a high degree of abstraction, but it is heavily dependent on the IXP1200 architecture. For example, it is hard to see how this same abstraction of levels could be maintained on the EZChip NP (see section II).

Apart from the work discussed above, additional research has focused on creating toolsets for specific NPs such as C compilers, simulators, debuggers and benchmarkers; some of this work is described in [18], [19], [20]. Like Teja, however, this work focuses on making tools more usable rather than on providing programming model that promote design portability and transferable programming skills.

Finally, the Network Processor Forum [21], a Industry consortium that aims to facilitate and accelerate the development of NP products, is starting to take an interest in NP programming interface standardisation. To date their focus has been on hardware level interoperability, but they have recently announced the formation of a study group that will define a software API for network-computing applications. However, it is envisaged that this API will not address low level programming of individual NP products.

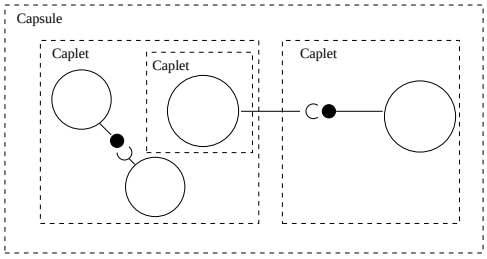

Fig. 2. Illustration of capsules and caplets

III. Towards a Generic Programming Model for Network Processors

A. Overview of the OpenCOM Component Model

A high-level view of our proposed component model, called *OpenCOM* [22], is given in figure 2. This depicts the central concepts of *components* (the filled circles), *capsules* (the outer dotted box), *caplets* (the inner dotted boxes), *interfaces* (the small circles), *receptacles* (the small cups), and *bindings* (the implied association between the adjacent interfaces and receptacles).

• **Components, Capsules and Caplets** *Components* are encapsulated units of functionality and deployment that interact with their environment (i.e. other components) exclusively through interfaces and receptacles. The key difference between our notion of components and Click's is that ours are deployed at run-time. The target of this deployment is either a *capsule* or a *caplet*. Both of these concepts represents a scope or locus for component deployment; the latter are sub-scopes of the former (they can be nested to arbitrary depth). If the deployment environment permits, caplets can be created and destroyed at runtime.

Each capsule offers a simple run-time API for component lifecycle management (i.e. loading components into the capsule and instantiating and destroying them), and interface/ receptacle binding (see below). To accomplish loading, the model supports the notion of *plug-in loaders*. New loaders can be added at runtime, and they can be selected according to their particular properties. Examples are given below. Importantly, the loading of components into a capsule can be requested by any component hosted by the capsule no matter which caplet is hosting it (this is referred to as *third-party deployment*).

• **Interfaces and Receptacles** *Interfaces* are units of service provision offered by components; they are expressed in terms of sets of operation signatures and associated datatypes. For language independence, OMG IDL [23] is used as a specification language. As in Click, components can support multiple interfaces: this is useful in recognising separations of concerns (e.g. between base functionality and management). *Receptacles* are 'anti-interfaces' used to make explicit the dependencies of components on other components: whereas an interface represents an element of service provision, a receptacle represents a unit of service requirement. Receptacles are key to supporting a third-party style of composition (to complement the third-party deployment referred to above): when third-party-deploying a component into a capsule, one knows by looking at the component's receptacles precisely which other component types must be present to satisfy its dependencies.

• **Pluggable Loaders and Binders** Finally, *bindings*, created via the above-mentioned API, are associations between a single interface and a single receptacle that reside in a common capsule (but not necessarily a common caplet). Similarly to plug-in loaders, OpenCOM also supports a notion of *plug-in binders*. Again, the idea is to give access to an extensible range of binding mechanisms with varying characteristics. See below for examples. As mentioned, the creation of bindings is inherently third-party in nature; it can be performed by any party within the capsule (i.e. not only by the 'first-party' components whose interface or receptacle participates in the binding).

B. Applying OpenCOM in NPs

We now consider how the above concepts can be applied in the diverse range of NP types outlined in section II. First, the scoping-related notions of capsules and caplets are useful in distinguishing different processors and types of processors on the NP in a generic manner (i.e., the packet processor dimention). For example, in an IXP1200, we might map a single capsule onto the entire NP, and sub-scope individual microengines, and the StrongARM control processor, as caplets. The capsule runtime in such a mapping would reside on the StrongARM where it could run in a standard operating system environment. An alternative mapping could encapsulate all the microengines in a single caplet. A plug-in loader associated with this caplet could then perform intelligent load balancing of components across microengines, thus providing a higher level of abstraction than the first alternative. The notion of caplets is also useful in isolating untrusted code, which is important in active networking environments. For example, a Java sandbox could be isolated as a caplet.

The IXP1200 is situated towards the 'centralised' end of the control dimension defined in section II-A. In an NP with less centralisation, such as the Motorola C-Port or the EZChip, the capsule abstraction could span both the NP itself and its hosting workstation. In this case, the capsule runtime would execute on the host. Alternatively, the capsule abstraction could be restricted to the NP itself, and the capsule runtime could execute on one of the general packet processors, if present. This would be possible in principle on the Motorola C-Port, but not on the EZChip which has no general purpose processors.

The pluggable loader concept is closely associated with capsules/caplets. Typically, at least one loader will be provided for each type of caplet, and each will know how to load components into the hardware environment underlying its particular caplet type. For example, in the IXP1200, there would be (at least) one loader for the StrongARM caplet and another for the microengine caplets. Importantly, the OpenCOM API allows selective transparency in the use of loaders. If full transparency

is desired, one can issue a call such as *load(component_c1, caplet_1);* which will deduce an appropriate loader type from meta-data attached to *component_c1*, and use this to load the component into the designated caplet. This essentially masks the fact that different components may be implemented in different machine languages. Even more transparency can be requested by issuing a call of the form *load(component_c1);* which causes the runtime to load *component_c1* into a default capsule using a default loader. Alternatively, one can opt for complete control and zero transparency by issuing a call of the form *load(component_c1, caplet_1, loader_3);*.

The pluggable binder concept is equally central to the component model's abstraction power. In this case, the abstraction is over the interconnect dimension. For example, on the IXP1200 we can encapsulate the NetBind binding mechanism (see section II-B) as a plug-in binder that is competent to bind components that have been loaded into a common caplet that represents a single underlying microengine. But equally well, we can provide a binder that is competent to bind components on different microengines (e.g. based on a shared memory or a next-neighbour register mechanism), or even between components on a microengine and components on the StrongARM. Again, the use of plug-in binders is selectively transparent. If we dont know or care in which caplets our two target components are located, we can say *bind(interface_1, receptacle_15);* and an appropriate loader will be selected according to location-related meta-data attached to the components that own the specified interface and receptacle. On the other hand, if it is important to select a particular mechanism, we can say *bind(interface_1, receptacle_15, loader_4);*. And so on.

Note that the abstract model of binding provided by the pluggable binder framework is consistent across all types of NP regardless of the nature and diversity of the interconnects between packet processors. For example, it can uniformly accommodate the fixed hardware channels supported by the pipeline-oriented EZChip, or the bus and shared memory interconnects of the Motorola C-Port, in just the same way as the various mechanisms supported by the IXP1200. Of course, different NP architectures may impose constraints on the form of possible bindings. For example, it would not be straightforward to directly bind components on non-adjacent processors on the EZChip NP; although even here it would be possible (if perhaps undesirable) to provide a plug-in binder that implemented this type of binding by transparently instantiating a forwarder on the intermediary processor(s).

The component concept alone is capable of providing considerable abstraction power in terms of accommodating dedicated non-programmable processors such as those provided by the Motorola C-Port. These processors can be accommodated by representing them with a 'dummy' component and an associated special plug-in loader and binder pair. Loading the component and binding it to the client component has the effect of making the service provided by the dedicated processor (e.g. table lookup) look as if it were a normal software component.

A final crucial property of the component model is its radically third-party nature in terms of loading and binding. Thanks to this, a component on an IXP1200 microengine can load and bind two components on the StrongARM control processor, or even on the host workstation, if that comes within the scope of the capsule.

Note that in this paper we omit, for lack of space, any discussion of the important OpenCOM notion of *component frameworks* which is used to support safe dynamic software reconfiguration. Information on this is available in the literature [22].

IV. Case Study: OpenCOM on the Intel IXP1200

For the past year we have been working to deploy and evaluate the OpenCOM component model on the Intel IXP1200. The IXP1200 was selected because of its open and well documented architecture, and because it is a richly-featured NP in terms of the three dimensions presented in section II-A.

To generate useful components with which to populate the implementation, we have taken as our starting point various modules (e.g. classifiers, forwarders, schedulers etc.) provided by the NetBind project [14] at Columbia University. We have transformed these bare modules, which are written in C or assembler, for both the StrongARM CPU and the microengines, into standard OpenCOM components by attaching appropriate meta-data (e.g. IDL interfaces, and loader and binder attributes) to produce standardly-packaged and deployable units.

The mapping we currently employ of OpenCOM capsules and caplets to the IXP1200 involves a single capsule that encompasses both the NP and the host workstation, and contains separate caplets for each of: the host workstation (actually, a single Linux process on the workstation); the StrongARM (again, a single Linux process); and the six microengines. The OpenCOM runtime runs in the StrongARM caplet; all the other caplets are 'slaves' of this 'central' runtime and incur only minimal memory overheads (see below). The memory footprint of the central runtime itself is of the order of 300Kb, although we believe that there is considerable scope for reducing this. The central runtime in the StrongARM caplet communicates with the other caplets by means of so-called *caplet channels*. The role of these is to bootstrap plug-in loaders and binders associated with non-central caplets, and to support communication between their two parts: such loaders/ binders are implemented as a 'delegator' part that resides in the central StrongARM caplet, and a (minimal) 'delegate' part that resides in the other caplet. As examples, we now briefly describe example loader and binder plug-ins that are associated with the microengine caplets.

The *microengine loader plug-in* is of interest in that it provides the illusion of dynamic loading despite the fact that the microengine hardware only allows modification of its instruction store when the CPU is stopped [11]. The basic capability provided by the microengine hardware is to stop the microengine, read/ write arbitrary instruction store locations, and then restart it at a hard-wired address. To

achieve transparent loading it is therefore necessary for the loader to not only load the new component but also to patch the (hard-wired) restart address so that subsequent execution resumes at the point it left off. The loader also has the ability to autonomously move code around within the instruction store to avoid memory fragmentation as components are loaded and unloaded. The loader is also of interest in that it constrains the form of OpenCOM components it is willing to load. The general notion of particular loaders somehow restricting the components they can work with is a general and powerful pattern in OpenCOM. In the present case, the IDL interfaces of loaded components may only support operations that accept and return a single integer. This restriction, which is enforced by inspecting the component's IDL meta-data at load time, is imposed partly to simplify the design of the associated binder (see below), and partly because the assumed model of component composition on the microengines (borrowed from above-mentioned NetBind work) is that components are bound into a more-or-less linear sequence and cooperatively work on queues of network packets whose addresses are passed as integer arguments.

The *intra-microengine binder plug-in* is strongly coupled to the loader just described. It builds on the above-mentioned NetBind technique (see section II-B) of creating bindings by 'morphing' jump instructions. However, the binder is more complex than the NetBind implementation because, together with the loader, it supports multiple instantiations of components (unlike NetBind which only supports singleton components). The single argument and return value are passed via a designated register, so the binder does not need to employ stubs or skeletons. The necessary entry and exit point information is obtained from IDL meta-data attached to the packaged component, which is transformed from relative offsets to absolute offsets by the loader. It is important to notice, by the way, that the IDL-specified interfaces do *not* incur performance overhead. In fact, the overhead of the binder in calling a null operation with no arguments or return values is only five (1-cycle) instructions. These subsume passing on the stack a pointer to the per-instance state vector of the called component, and the return address. Note that NetBind incurs an overhead of just two 1-cycle jump instructions (for the call and the return). But this is because NetBind does not support multiple instantiations of components. However, note that we could easily retrieve the NetBind performance in the OpenCOM environment simply by implementing and installing an new binder plug-in that assumes components that observe the NetBind calling convention and (therefore) does not support multiple component instantiation. The essential point is that OpenCOM's plug-in architecture enables us to support any appropriate trade-off. More generally, it is crucial to note that the performance of the OpenCOM programming model as a whole is almost entirely dependent on the performance of the binding mechanisms used. Almost all the value-added features of OpenCOM are confined to the central runtime and do not 'get in the way' when components communicate with each other on the NP's fast path.

Apart from the microengine loader and binder discussed above, we are currently implementing loaders that load components into StrongARM and host workstation caplets; and binders that bind components across any pairwise combinations of the three caplet types. Bindings between the microengines and the other two caplet types are considerably more complex than intra- and inter-microengine bindings as they require stubs and skeletons to map the parameter and return value to a bus packet. To minimise memory overhead, the microengine-side stubs/ skeletons are hand coded rather than being generated automatically from the IDL specification.

V. Conclusions and Future Work

In this paper we have characterised the design space of NPs and proposed a component-based programming model that, we have argued, can be applied generally within this design space. The component model, mainly through its plug-in loaders and binders and its associated notion of caplets, provides a high degree of design portability and potential for skill transfer. We have also demonstrated how plug-in loaders and binders can exploit NP-specific features to provide both high performance (for example, our microengine binder incurs comparable overheads to NetBind on the IXP1200), and value-added behavior (for example, our microengine loader/ binder supports multiple instantiations of components and transparently optimises instruction store use as components are dynamically loaded/ instantiated/ destroyed). Most importantly, we have argued that our abstractions are *generally applicable*. NP-Click also abstracts NP-specific features - e.g. it provides an API to manage and allocate microengine LIFO resources on the IXP1200. But this API would make no sense on an NP that did not support LIFOs. The OpenCOM approach would be to provide a plug in binder (a generic abstraction) that internally uses, manages and allocates LIFOs (if present) to build a reusable binder plug-in.

OpenCOM also supports run-time reconfiguration. In this paper we have discussed the basic mechanisms behind this (i.e. receptacles, and the ability to bind and unbind components at runtime). But we have not elaborated on OpenCOM's approach to managing integrity, consistency, safety and security when performing reconfiguration operations. As mentioned, we rely on the notion of *component frameworks* to support this. We have already explored the provision of component frameworks in other domains in which we have applied OpenCOM (e.g. Middleware [24]); one aspect of our future work will be to further explore this interesting and demanding area in the NP domain.

The main thrust of our future work, however, will be to explore the use of OpenCOM in other NP environments. We are already looking at the more advanced IXP2400 from Intel and the IBM PowerNP; but we would also like to provide further evidence for the generality of our approach by looking in more detail at NPs elsewhere in the design space outlined in this paper.

REFERENCES

[1] Ruf L. Pletka R. Erni P. Droz P. Towards high-performance active networking. *Proceedings of the Fifth Annual International Working Conference on Active Networks (IWAN 2003)*, December 2003. Kyoto, Japan.

[2] C. Sauer K. Keutzer C. Kulkarni, M. Gries. Programming Challenges in Network Processor Deployment. In *Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, October 2003.

[3] Agere Systems Proprietary. The Challenge for Next Generation Network Processors. April 2001.

[4] Mel Tsai, Chidamber Kulkarni, Christian Sauer, Niraj Shah, and Kurt Keutzer. A benchmarking methodology for network processors. In *1st Workshop on network processors along with HPCA 2002*, February 2002.

[5] Paulin P G. Network processors: A perspective on market requirements, processor architectures and embedded s/w tools. *STMicroelectronics*, 2001.

[6] Adiletta M. et al. The next generation of intel network processors. *Intel Technology Journal, Volume 6, issue 3*, August 15 2002.

[7] Radisys Corporation. IXP1200 White Paper: Using the Intel IXP1200 Network Processor to optimize Packer-Processing Application Development, 2001.

[8] Motorola Research. Architecture guide, C-5e/C-3e network processor, silicon revision B0, 2003.

[9] EZchip technologies. Network processor designs for next-generation networking equipment white paper, 2003.

[10] Cisco Systems Inc. Parallel express forwarding on the Cisco 10000 series, 2003.

[11] Intel Press. MicroACE, design document, revision 1.0. *Intel Press, Intel Corporation*, 2001.

[12] Akash Deshpande, Kevin Crozier, and Mandeep Baines. The Teja Software Platform for Network Processors.

[13] James Allen et al. IBM PowerNP Network Processor: Hardware Software and Applications. *IBM Journal of Research and Development*, 47(2/3):177–194, March/May 2003.

[14] Campbell A.T., Kounavis M.E., Villela D.A., Vicente J.B., de Meer H.G., Miki K., and Kalaichelvan K.S. NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers. In *5th IEEE International Conference on Open Architectures and Ne twork Programming (OPENARCH'02)*, June 2002.

[15] Kurt Keutzer Niraj Shah, William Plishker. Np-click: A programming model for the intel ixp1200. In *2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA-9), Anaheim, CA*, February 2003.

[16] Karlin S. and Peterson L. VERA: An Extensible Router Architecture. In *4th International Conference on Open Architectures and Networ k Programming (OPENARCH)*, April 2001.

[17] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a Robust Software-Based Router Using Network Processors, Oct 2001.

[18] Wagner J. Leupers R. C compiler design for an industrial network processor. *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*, 2001.

[19] Memik G. Mangione-Smith W H. Hu W. Netbench: A benchmarking suite for network processors. *ICCAD*, 2001.

[20] Gries M. Kulkarni C. Sauer C. Keutzer K. Comparing analytical modeling with simulation for network processors: A case study. *Design, Automation, and Test in Europe (DATE), Munich, Germany*, March, 2003.

[21] Network Processing Forum Working Group. Network processing forum backgrounder, Oct 2002. http://www.npforum.org/.

[22] Geoff Coulson, Gordon Blair, David Hutchison, Ackbar Joolia, Kevin Lee, Jo Ueyama, Antonio Gomes, and Yimin Ye. NETKIT: A Software Component-Based Approach to Programmable Networking . In *ACM SIGCOMM Computer Communication Review*, volume 33, No 5, October 2003.

[23] Object Management Group, Inc. CORBA 3.0 - IDL Syntax and Semantics chapter, formal/02-06-07.

[24] G. Coulson, Blair G.S., M. Clarke, and N. Parlavantzas. The Design of a Highly Configurable and Reconfigurable Middleware Platform. *ACM Distributed Computing Journal*, 15(2):109–126, April 2002.