SPECIFICATION OF NETWORK SERVICES AND MAPPING ALGORITHMS

Lukas Ruf
In&Out AG, Zurich, Switzerland
Tilman Wolf
University of Massachusetts, Amherst MA, USA
Karoly Farkas and Bernhard Pattner
ETH Zurich, Switzerland

ABSTRACT—***In recent year, the functionality of networking infrastructure has expanded to the point where routers not only provide data connectivity but also a variety of processing services. A major challenge in this context is to manage these processing resources and to allocate them to data transfers in an efficient manner. In our work*[1]***, we present a novel way of how end-system applications can specify resource requirements. We explore the performance of several heuristic approaches to solving the intractable problem of mapping requirements to system resources.***

I. INTRODUCTION AND MOTIVATION

The strategic vision for command, control, communications, computers, intelligence, surveillance, and reconnaissance (C4ISR) is to provide military forces with information technology to succeed in their mission. As part of this vision, the capability to collect, process, and disseminate information is an important aspect to achieving information superiority. The Defense Information Infrastructure combines communication and processing capabilities in a battlespace communication network.

In an advanced network infrastructure (e.g., tactical Internet, civilian Internet), there is a general problem of how to coordinate communication and processing tasks in a coherent manner. Advanced programmable routers allow for processing service to be deployed within the network and effectively create a distributed computing platform. In this paper, we focus on the issues of how to specify processing service tasks in the context of a communication network and how to map these tasks onto processing resources. In particular, we address three problems: 1.) **Service Specification**: a methodology for describing the required processing tasks, their logical dependency, and the data transfer operations between them. 2.) **Network and Node Specification**: a methodology for characterizing the capabilities and performance of processing resources, their interconnects, and their resource availability. 3.) **Mapping Algorithm**: an algorithm for determining an optimal or near-optimal allocation of services to processing resources. The mapping problem is particularly difficult as the general problem of distributing multiple processing resources optimally onto a resource graph is NP-complete [9], [16]. Nevertheless, it is an important problem that appears throughout distributed computing and computer networking. We therefore look at the general service mapping problem into context of three specific domains (illustrated in Figure 1): 1.) **Service-Oriented Computing**: In service-oriented computing, applications are distributed over numerous different computing and web services of service providers. Interactions between software components need to be specified and mapped to a heterogeneous processing environment. The middleware needs to determine a suitable mapping in order to hide the complexity of the underlying system from the application. 2.) **Service Provisioning on Routers**: Router systems process packets in order to forward network traffic. Depending on the router system, this processing can range from simple address lookups to complex payload scanning. Due to the performance requirements of such systems, numerous processing resources are available on router systems. In order to efficiently utilize this system, processing requirements need to be specified and mapped to the underlying hardware. 3.) **Application Mapping on Network Processors**: A typical processing resource of a router, called "network processor or NP," is implemented as a system-on-a-chip with numerous parallel embedded processor cores. Application components need to distributed across these cores in order to maximize the performance of this system. These scenarios can each be seen as independent mapping problems, or as a combined mapping problem with different levels of granularity. Our main focus in this paper is on the router level, but as shown, all levels are interconnected and in principle pose the same problem.

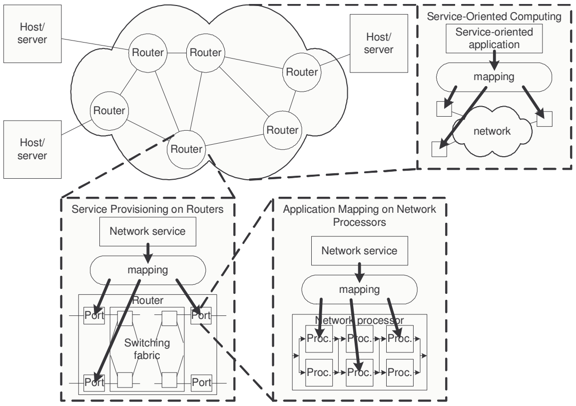


Fig. 1. Mapping Scenarios

The contributions of our paper are threefold. First, we introduce a novel **service programming language** that allows

---

[1] This research was carried out while at ETH Zurich.

the specification of services. Second, we present a **node specification** methodology that permits the description of capabilities of processing systems. Third, we discuss a **novel mapping algorithm** that achieves a good approximation to the intractable mapping problem. This mapping algorithm is then compared qualitatively to other existing mapping approaches.

We structure this paper as follows. In Sec. II, we present related work. Sec. III introduces our service model to specify component based network services. Sec. IV presents our node model that allows the specification of processing infrastructures. Our mapping algorithm is introduced in Sec. V and compared to existing algorithms in Sec. VI. The work is summarized in Sec. VII.

II. RELATED WORK

There have been a number of efforts to develop abstractions for specifying services on network routers. The Click modular router project [12] defines two environments of code execution (EE) on Linux: one is the in-kernel EE and the other is a Linux user space EE. Processing services are provided by a specification of interconnected Click elements. While Click defines arbitrary service graphs by its specification language, it does not have the expressiveness to specify resource limits. Moreover, the language does not support the required flexibility of service extensions due to the architectural limitations of the Click EEs.

NetScript [6] defines a framework for service composition in active networks that is programmed by a dataflow composition language, a packet declaration language, and a rule-based packet classification language. The first defines a method to specify data path services as a composition of interconnected service components. The second defines the packet structure of network protocols, and the third defines the packet classification rules that are installed in the NetScript kernel. Service components (so-called "boxes") in NetScript provide a container for code or hardware-based service components, or other boxes in a recursive manner. NetScript's composition language cannot define control relations between control service components, does not provide capabilities to extend previously deployed network services, and lacks the expressiveness to specify resource and placement constraints of components.

Models for processing resources on network routers can be classified into *pool-extended node models* and *port-extended node models*. In the first case, processing is provided by a pool for shared processing elements that can be accessed by all ports. In the latter case, each individual router port is extended by a processing system (e.g., smart port card on WUGS [3]). Recently developed network processor (NPs) are typically placed on the ports of a port-extended model. NPs are embedded systems-on-a-chip that are optimized for handling high-speed network traffic. Typical configurations use tens to hundreds of parallel processors, embedded memory, and high-speed I/O. Commercial examples are the Intel IXP family of processors, the IBM PowerNP, and the Cisco Silicon Packet Processor.

In one of the first pieces of work that address the problem of finding an optimal solution to placing services in networks, Choi et al. propose to use a single metric for processing and communication cost [5]. By transforming the network graph into a "layered graph", processing is represented by inter-layer links, while communication links are constrained to intra-layer connections. A mapping solution is found by determining shortest path from the source node in the first layer to the sink node in the last layer. This method works with infinite capacities since the topology of the graph of processing elements needs to be invariant. ANCS [11] applies the layered graph method with network services modeled as Unix-like pipes of service components. Since this network model assumes homogeneous processing elements with infinite link, memory, and processing capacities, the layered graph method can be applied directly. XNP [4] addresses the problem of service mapping onto networks with finite link capacities. It solves this problem by the proposal of two different graph creation procedures that include links only if they meet the capacity requirements. While elegant, these solutions only consider constraints on bandwidth capacity. Other constraints (e.g., heterogeneity of processing element, memory types) that appear in real systems cannot be considered. A more general approach to providing a heuristic solutions to the mapping problem with multiple constraints is "randomized mapping" as proposed by Karp [10] and Motwani and Raghavan [13]. In one of the author's prior work, randomized mapping has been applied to mapping processing tasks onto network processor cores [17].

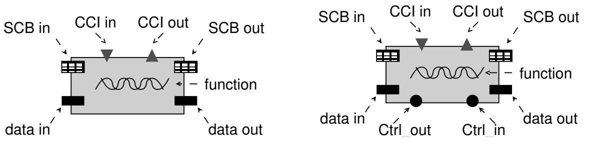
III. COMPONENT BASED NETWORK SERVICES

In this section, we introduce our service model and the Service Programming Language (SPL) that is used to specify a service. The key challenge is to make the service model expressive enough to allow the description of a wide range of services. At the same time, the Service Programming Language needs to be simple enough for users to use and for the service platform to process.

A. Service Model

1) Service Model Components: Our service model describes services as graphs of edges and vertices with edges representing chains of service components, and vertices defining the interconnection between them. Network services are defined by six fundamental concepts, such as name spaces, service control buses, service components, service chains, guards, and hooks, in the following way:

- **Name spaces** are abstract constructions of our service model that are used to avoid name collisions between services by defining a logical space. Within a name space, elements are identified by literals per service.
- The **service control bus** (SCB) provides service-internal signal propagation among the elements of one network service. The semantics of the signals on the SCB are service specific except for three signals labelled `ACCEPT`, `ABORT` and `CHAINEND` that are used for control and management operations of the service infrastructure [14].
- **Service components** provide the service functionality. Two types are defined: *data path service components* (DSCs) and *control service components* (CSCs). DSCs provide the functionality residing in the data plane to process regular network traffic. CSCs provide service internal control functions as well as control plane elements.



(a) Data Path Service Component (b) Control Service Component

Fig. 2. Service Components

In Fig. 2(a), the model of a DSC is visualized. A DSC provides a function according to the plugin model [7]. It extends the interfaces of the original plugins. In addition to the data in- and output ports, our DSC defines in- and output ports for the SCB and provides a component control interface (CCI)[2]. Fig. 2(b) presents the model of a CSC. CSCs are service components like DSCs. Hence, they offer the same component interfaces but export in addition multiplexed controlling interfaces (labelled Ctrl_in and Ctrl_out in Fig. 2(b)). Controlling interfaces are required to control other service components via their CCIs. Our model foresees that a CSC may be able to control multiple other service components. A logical multiplexing of the controlling interfaces is defined for CSCs implementing the controlling functionality for multiple service components.

- **Service chains** provide an aggregation of one or more DSCs that are strongly linked. A chain of strongly linked DSCs allows only for signal propagation along the SCB between service components, and between service components and the service infrastructure. No demultiplexing of network traffic is available between the elements of a service chain allowing for fast pipeline-style processing of network traffic by subsequent service components. The signal on the SCB labelled `ABORT` causes the service infrastructure to abort the current service chain.
- **Guards** provide the demultiplexing functions that control the acceptance of network traffic to enter service chains. Their definition has been inspired by the concept of Dijkstra's guarded commands. In our service model, guards are represented by DSCs that signal the acceptance (`ACCEPT`) or rejection (`ABORT`) of network traffic by the mechanisms of its SCB output port. Visually depicted, a guard is the first service component of a service chain that accepts a packet or rejects it.
- A pair of **hooks** confines a service chain. They initiate and terminate a service chain. Multiple service chains may be attached to hooks. Thus, hooks are key elements of the respective name space. Within a name space, they are identified by their label. They are created as part of the service program on demand. If ingress hooks are created, they must be bound to a network interface. Otherwise, they must refer to previously created ones. Egress hooks may be dangling, implying the discard of packets. The purpose of dangling outbound links is the provisioning of a hook for later service additions to extend provided functionality.

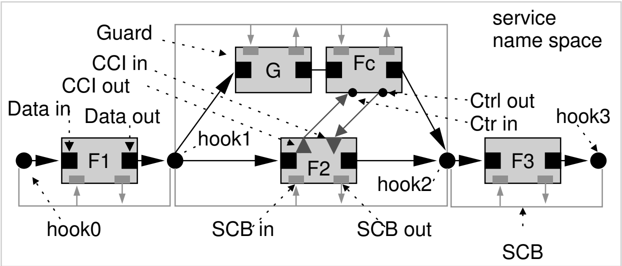


Fig. 3. Control and Data Path Relations Among Service Components

In Fig. 3, a service graph is presented that consists of four service components named $F1$, $F2$, $F3$ and Fc embedded between four hooks as well as of a guard labelled G that controls the packet acceptance for its service chain. It illustrates the data path and control relations between service components with Fc controlling $F2$. In Fig. 3, this controlling functionality is represented by the letter $'c'$ indicating control. Moreover, it visualizes the SCB covering service chains.

2) Dispatching Semantics: The graph representation of services raises the question which path is followed by network traffic as it is being processed. We define two different dispatching semantics for this purpose: copy and first-match-first-consume.

3) Resource Constraints: Service component instances have specific resource characteristics. Resource characteristics specify the amount and type of resources needed for the component instantiation and their execution. Resource characteristics define part of the parameter space the service infrastructure must be able to cope with. As an example, different instruction set architectures (ISAs) may be available on an NP.

B. The Service Programming Language

The specification of network services on a platform for multiport router devices requires a concise service programming interface (SPI). The SPI is required to cope with the flexibility of the service model introduced above. Since network services are modelled as a graph of interconnected service chains, a method is required that provides the appropriate specification. We define therefore our Service Programming Language (SPL).

The SPL definition provides a formal language to specify network services. Our service model is described by six key productions[3] that allow the specification of the service components described above.

These key productions followed by the production name are: 1.) Service: `SERVICE` 2.) Service Component: `SERV_COMP` 3.) Service Chains: `SERV_CHAIN` 4.) Control Chains: `CTRL_CHAIN` 5.) Guards: `GUARD` 6.) Hooks: `HOOK_IN`, `HOOK_OUT` . The SCB interfaces are mandatory for every service component. Hence, they do not need to be specified explicitly.

Lst. 1 presents the key productions[4] of the SPL definition. Note that the namespace is identified by the `ID` production.

---

[2] In Fig. 2(a), interfaces are labelled SCB_in, SCB_out, CCI_in, CCI_out, data_in, and data_out respectively.

[3] Note that we refer to the *key = value* pair by the term *production* [1], and refer to the key by the term *production name*.

[4] Self-explanatory productions like, for example, `BW`, `CYCLES` or `MEM` are not provided here.

```
ID          = "#" VALID_NAME.
TIMED       = "timed="DELAY.
BW_RES      = "bwmin="BW "bwmax="BW [ "pps="NUMBER ].
CPU_RES     = "cpumin="CPU "cpumax="CPU.
RAM_RES     = "type="ID "rammin="RAM "rammax="RAM.
PROC_TYPE   = ("ia32"|"ia64"|"np4"|"np4_pp"|"ixp2400"|
               "ixp2400_pp"|....).
CTRL_INFO   = (STRING | "file=" VALID_NAME ).
COMP_SPEC   = ( "src" [ ID ] | "bin" ( PROC_TYPE | ID ) )
              [ "|" CPU_RES ] [ { "|"  RAM_RES } ].
COMP_IDENT  = ( [ "(" COMP_SPEC ")" ] VALID_NAME ID | ID ).
SERV_COMP   = COMP_IDENT [ ":" ID ] "(" [ CTRL_INFO ] ")".
CTRL_COMP   = [ TIMED ] SERV_COMP { "!" ID "@"NUMBER }.
CTRL_CHAIN  = "{" { CTRL_COMP } "}".
COMP_STRING= "{" { SERV_COMP } "}".
GUARD       = "[" [ "|" BW_RES ] [ SERV_COMP ] "]".
HOOK_IN     = (ID | ">" ID [ "copy" ]   "?" INTF).
HOOK_OUT    = (ID | ">" ID [ "copy" ] [ "?" INTF ]).
SERV_CHAIN  = HOOK_IN
               "@" [ TIMED ] [ GUARD ] COMP_STRING "@"
              HOOK_OUT.
SERVICE     = "{" ID [ "!"  CTRL_CHAIN ] { SERV_CHAIN } "}".
```

Listing 1. The Service Programming Language

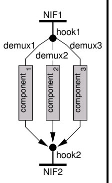
The fundamental concept of the SPL is the linear specification of arbitrary service graphs consisting of service and control chains. Based on the concept of hooks to which service chains are attached, graphs are created from the linear specification. Service chains may be added to hooks and removed therefrom at run-time. The language supports fast scanning/parsing mechanisms. It is context free and allows for easy translation to and from other notations and graphical user interfaces.

C. Service Model Example

As an example for the use of SPL, we briefly describe a service program and its corresponding visualization hereafter.

TABLE I

THREE PARALLEL SERVICE CHAINS

Visual.	Chain 1	Chain 2	Chain 3
	{ #threeparall > #hook1 ? NIF1 @/* HOOK */ [/*DEMUX1*/] { /*COMP_STR*/ (**bin** ia32) component1 #instance1ID (/*CTRL_IO*/) } @/* HOOK */ > #hook2 ? NIF2	/*extend*/ #hook1 @/* HOOK */ [/*DEMUX2*/] {/*COMP_STR*/ (**bin** ia32) component2 #instance2ID (/*CTRL_IO*/) } @/* HOOK */ #hook2	/*extend */ #hook1 @/* HOOK */ [/*DEMUX3*/] { /*COMP_STR*/ (**bin** ia32) component3 #instance3ID (/*CTRL_IN*/) } @/* HOOK */ #hook2 }/*End*/

Tab. I presents a simple exemplary service program that defines a network service with three parallel service chains. The service program illustrates the linear specification of a service graph with parallel service chains. The service identifier (#*threeparallel*) is followed by the creation of hook1. No copy method is specified. Hence, its packet dispatching semantics follow the first-match-first-consume method in the top-down order of specified service chains. Hook1 is bound to one network interface (NIF) that is symbolized by the term *NIF1*. The service chain that consists of component_1 is attached to hook1, first. While the figure in Tab. I illustrates the demultiplexing of flows to the particular service chains by attaching abstract demux conditions to the links between hook1 and the respective service chain, no real demultiplexing is specified in the service program. However, demultiplexing conditions are indicated in the service program by the respective comments. All service chains lead into hook2, which is bound to the second NIF (*NIF2*). The second and third service chains follow the same principle. Their specification differs from the first service chain by that hooks are re-used, i.e. the newly defined service chains are attached to the existing hooks.

IV. MODELLING NETWORK NODES

Distributed computing platforms are complex systems, no matter if they are designed for service-oriented computing, network services on routers, or packet processing applications on network processors. The heterogeneity of processing resources and interconnects makes it difficult to manage and control these systems even on a theoretical level. In practical implementations, system- and vendor-specific device configuration issues complicate things further.

The node specification that is required for a service mapping algorithm needs to consider these system issues as well as be applicable to a broad range of different system designs.

In this section, we introduce a hierarchical node model to abstract from a variety of underlying hardware platforms. Then, we introduce the corresponding node specification language to allow for a concise system specification.

A. Node Model

Multiport router devices with programmable network interfaces define the hardware architecture of network nodes where the processing capacity scales with the number of installed "blades". These network processor blades provide the communication, memory and processing resources to be programmed at run-time with new and extended network services. The network processors located on the blades exhibit an architecture that consists of multiple specialized packet processor (PPs) cores for high-speed network traffic handling and one or more control processor (CP) cores for managing the PPs.

To model the potentially large sets of processor cores, we define a threefold organization of processing elements. Processing elements are categorized by the way they share memory and communication resources. First, *cores* are grouped into a *processor* if they share communication paths, memory resources, or both. Second, processors that share such resources are grouped into *clusters*. Third, clusters are organized into *tiers* depending on their sharing of direct communication paths with upper tiers. Thus, a cluster consists of peer processors with cores that communicate directly with upper tiers. A tier consists of multiple peer clusters that share the same parent cluster.

Fig. 4 presents an illustration of our node model. It is built of four elements: the three types of processing elements (clusters, processors, and cores) and hardware interconnects. We refer to hardware interconnects as ICo-Bus for *inter-core bus*, IP-Bus for *inter-processor bus* and IC-Bus for *inter-cluster bus*[5].

---

[5] We label the hardware interconnects by the term bus, since link-type hardware interconnects may be represented by a bus interconnecting only a pair of cores, processors, or clusters.

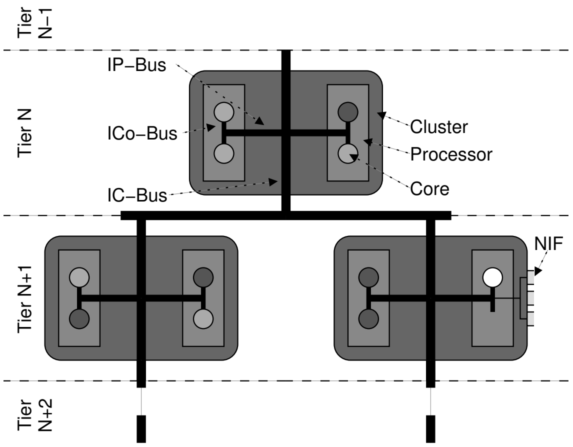


Fig. 4. Hierarchical Node Model

B. The Node Specification Language

Modelling of hierarchical network nodes demands for a specification language that provides the expressiveness to specify a network node at the required abstraction level. We present here our Node Specification Language (NSL) derived from the aforementioned node model.

The NSL consists of the following elements with their corresponding key-productions [1] (separated from the element by a colon): 1.) Node Graph: `GRAPH` 2.) Processors: `PROCESSOR` 3.) Network Communication: `COMM` 4.) Communication Specifier: `COMM_SPEC` 5.) Interfaces: `INTF` 6.) Clusters: `CLUSTER` 7.) Cores: `CORE` 8.) Communication Paths: `COMM_PATH` 9.) RAM: `MEMORY`. Lst. 2 presents the syntax of our NSL language[6].

```
CPU_FREQ       = "freq="CPU.
CPU_TYPE       = "isa="ALPHA_NUM.
MEMORY         = { "type="ID "mem="RAM }.
COMM_PATH      = ( "tierrecvdown" | "tiersendup"
                 | "tiersenddown" | "tierrecvup"
                 | "intfrecv"     | "intfsend"
                 | "clusterrecv"  | "clustersend"
                 | "procrecv"     | "procsend"
                 | "corerecv"     | "coresend" |
                 "neighbour").
COMM_SPEC      = BW DELAY [ "link" ].
NIF            = "{" BW { INTF } "}".
CORE           = "(" [ MEMORY ] CPU_FREQ CPU_TYPE
                   [ { COMM_PATH } ] ")".
PROCESSOR      = "(" [ CLUSTER_SPEC ] { CORE } [ NIF ] ")".
CHILDREN       = "[" COMM_SPEC "]" { CLUSTER }.
CLUSTER_SPEC   = "[" [ MEMORY ] [ COMM_SPEC ] "]".
CLUSTER        = "(" [ CLUSTER_SPEC ] { PROCESSOR }
                   [ CHILDREN ] ")".
GRAPH          = CLUSTER [ CHILDREN ] ";".
```

Listing 2. The Node Specification Language

V. SERVICE MAPPING

The mapping of a network service onto a network node represents a particular instance of the graph embedding problem. Modern router devices embed NPs at the network interface level. This embedding complicates the mapping problem since, already on-chip, a *heterogeneous multicore* architecture with *various memory types* and a specific *communication* infrastructure is given. Thus, a service mapping algorithm needs to match specified service constraints of processing capacity and capability, memory capacity and types, communication capacity, and network interfaces with finite resources capacities of router devices.

A. Mapping Problem

By the help of the service programming and the node specification language (see Sec. III and Sec. IV), network services and realistic router devices can be described in a concise way. For the mapping of network services to router devices, pre-processed service programs and node specifications are needed that have the relevant dependencies resolved. We name the pre-processed service program as *instance graph*, and the corresponding node specification as *core graph*. The instance and core graph result from the scanning, parsing and compilation steps of the input data processing, and have all dependencies resolved similar to an abstract syntax tree [1].

We can define the mapping problem similarly as it is defined by the layered graph method of XNP [4]. However, that XNP configures linear services with bandwidth capacity constraints in extensible networks (cf. Sec. II). Our mapping problem extends that problem space by the much larger set of constraints and the need to resolve the blocking-problem of blind alleys.

B. The SLESP Mapping Algorithm

Since the graph embedding problem is known to be *NP*-complete [9], [16] if more than one constraint must be considered for the mapping, the use of an exhaustive search method that investigates every possible solution falls short for devices with multiple processing elements. Heuristics are required to find a solution for the mapping problem even though it may be only near-to-optimal.

We propose SLESP (Single Layer Extended Shortest Path), a novel algorithm that solves the mapping problem with all constraints and copes with the problem of blind alleys using back tracking mechanisms. Our algorithm takes an instance graph, and finds the shortest or near-to-shortest path through a core graph.

```
SLESP(Gc, ce, Gi, ie)
  place(Gi[ie], Gc[ce]);
  Ce=create_candidate_set(Gc, ce, Gi, ie, ie++);
  while not empty Ce and not success_status
    cebest=pop(Ce);
    success_status=SLESP(Gc, cebest, Gi, ie++);
  if success_status
    return success;
  return not success;
```

Listing 3. Pseudo Code of SLESP

Lst. 3 presents the pseudo code of SLESP without the back tracking methods. SLESP works as follows. With the service, it starts at an ingress hook instance graph element ($ie_0 \in G_I$). Since this hook is bound to a specific network interface, and network interfaces are assigned to processor cores, the starting

---

[6] By analogy to the SPL syntax, self-explanatory productions are not provided here.

processor core ($ce_0 \in G_C$) is determined [7]. ie_0 is placed on this staring core ce_0. A spanning tree of candidate cores for the next instance graph element is calculated by Dijkstra's shortest path algorithm [8] that has been extended to cope with all constraints. The spanning tree is rooted at ce_0, and defines an ordered set Ce of suitable processor cores to host the next instance graph element. Our algorithm selects the best candidate core ($ce_{best} \in Ce$), places the next instance graph element, and recurs to the same procedures. The shortest path for a given instance graph is retrieved, thus, from the concatenation of the different inter-core communication paths for ie_{k+1} and ie_{k+2} plus the processing delays.

VI. EVALUATION

To evaluate the performance of the proposed service mapping algorithm, we analyze its worst case time complexity. To put these results into context, we analyze and compare a total of three algorithms: (1) our proposed SLESP algorithm, (2) the layered graph (LG) mapping algorithm that was developed by Choi et al. in the context of XNP [4], and (3) the randomized mapping (RM) algorithm that was developed by Weng et al. for mapping tasks on network processors [17] (see Sec. II for a more detailed discussion of the latter two algorithms).

A. SLESP Algorithm Complexity

The time complexity, $COMP$, of our SLESP mapping algorithm is heavily dominated by the requirement to recalculate the shortest path using an algorithm that extends Dijkstra's shortest path calculation method. Dijkstra's shortest path algorithm [8] provides a time-complexity of $COMP_{DS} = O(|V|log|V| + |E|)$ (if implemented with priority queues) [2].

In our current implementation, the extended shortest path algorithm is applied per instance graph element. In addition, we compare the capacities of each vertex and each edge once per recursion, which leads to a worst case complexity of $COMP_{DSext} = O(|V|(log|V|+1) + 2|E|)$ per service component placement. Thus, for a given service program with N instance graph elements that are to be mapped onto a processor core graph with V vertices (cores) and E edges (hardware interconnects), we derive the complexity for large N as follows:

$$\begin{aligned} COMP_{SLESP} &= COMP_{DSext} + |V|(COMP_{DSext} \\ &\quad + |V|(COMP_{DSext} + |V| \ldots)) \\ &= \sum_{i=0}^{N} |V|^i COMP_{DSext} = \frac{COMP_{DSext}|V|^{N+1} - 1}{|V| - 1} \\ &= O(COMP_{DSext}|V|^N) \\ &= O(|V|^{N+1}(log|V|+1) + 2|V|^N|E|) \end{aligned}$$

The complexity is dominated by the $|V|^N$ term, which is a result of our algorithm potentially searching the entire core graph exhaustively due to the back tracking mechanisms. However, it still performs better than a straightforward exhaustive search algorithm, where each edge is handled individually and the complexity is $O(|V|^N * 2^{N|V|})$.

B. Layered Graph Algorithm Complexity

In XNP [4], the layered graph (LG) method with capacity tracking calculates a spanning tree through a layered core graph and takes only bandwidth capacities into consideration. The basis core graph plus N identical copies (for N service components) define the layers of that graph. The LG with $N+1$ layers is built by the insertion of virtual inter-layer processing links between pairs of adjacent layers at candidate processing cores. With a set of P_i candidate processing cores per service component, the authors define the complexity for their algorithm as follows:

$$COMP_{LG} = O((N+1) * (|V|log|V| + |E| + \sum_i P_i) + (N|V| * |E|))$$

The complexity depends on the number of services and the structure of the graph. The worst case scenario is however less complex than SLESP – at the cost of not considering all possible mapping solutions.

C. Randomized Mapping Algorithm Complexity

The randomized mapping (RM) method randomly places service elements on processor cores and evaluates the results. This process is repeated and the best overall mapping is retained. After a certain number of repetitions, the algorithm is expected to converge. The complexity depends on the number of rounds, R, the number of service elements, N, and the cost of analyzing a placement, q, and can be stated as

$$COMP_{RM} = O(R(N+q)).$$

While a possible mapping can be found quickly for very small R, nothing can be said of the quality of the solution. Higher quality solutions are more likely encountered as R is increased.

D. Comparison

To summarize the analytic performance of the three algorithms, Table II shows the results of the above discussion. The run-time (RT) performance and quality of mapping (MQ) results are indicated by '+', '∘', and '−' representing a decreasing order. We can conclude, that our SLESP algorithm has the highest worst case time complexity, however it can handle all the given constraints and implements back tracking mechanisms. The LG algorithm of XNP is less complex but it cannot handle all the constraints and it does not implement back tracking. The RM algorithm has the lowest complexity level but it does not guarantee an optimal solution and it can be applied only in case of specific scenarios (see [17]).

TABLE II

COMPARISON OF ALGORITHMIC COMPLEXITY AND MAPPING QUALITY OF THE THREE MAPPING ALGORITHMS

Algo.	Complexity	RT	MQ
SLESP	$O(\|V\|^{N+1}(log\|V\|+1) + 2\|V\|^N\|E\|)$	−	+
LG	$O((N+1) * (\|V\|log\|V\| + \|E\| + \sum_i P_i) + (N\|V\| * \|E\|))$	∘	∘
RM	$O(R(N+q))$	+	−

Despite this undesirable exponential complexity of SLESP, the practical usage scenarios show that the algorithm is still a

---

[7] Note, in case multiple cores control one network interface, one is selected at random.

useful approach [15]. In particular, since it can find solutions that cannot be found by the algorithmically simpler layered graph method.

VII. SUMMARY AND CONCLUSIONS

In this paper, we have introduced a methodology for specifying services and service platforms as well as a novel algorithm for mapping service nodes to devices. The Service Programming Language (SPL) has been proposed as a context-free service programming language of our service model. The node model that we have presented allows the description of a range of different computational platforms. The model considers functionality and performance constraints for processing, memory, and communication. The structure of a service platform can be represented via clusters of processors that are organized hierarchically. The mapping algorithm that we have introduced is compared to two other algorithms that have been published in prior work. Our complexity analysis evaluates our mapping algorithm and shows that SLESP can yield better mapping results.

In summary, we believe that a concise methodology for specifying network services and processing systems is important for designing middleware for service-oriented computing platforms. Our proposed mapping algorithm is an important step towards achieving a system that can hide hardware complexities and automatically manage network processing resources.

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers – Principle, Techniques and Tools*. Addison-Wesley, 1986.

[2] M. Barbehenn, "A note on the complexity of dijkstra's algorithm for graphs with weighted vertices," *IEEE Trans. Comput.*, vol. 47, no. 2, 1998.

[3] T. Chaney, A. Fingerhut, M. Flucke, and J. Turner, "Design of a Gigabit ATM Switch," in *Proc. of INFOCOM'97*, Apr. 1997.

[4] S. Choi and J. Turner, "Configuring Sessions in Programmable Networks with Capacity Constraints," in *Proc. of IEEE ICC*, May 2003.

[5] S. Choi, T. Wolf, and J. Turner, "Configuring Sessions in Programmable Networks," in *Proc. of IEEE INFOCOM*, Apr. 2001.

[6] S. da Silva, D. Florissi, and Y. Yemini, "Composing active services with NetScript," in *Proc. DARPA Active Networks Worshop, Tucson, AZ*, Mar. 1998.

[7] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router Plugins: A Software Architecture for Next Generation Routers," in *Proc. of the ACM SIGCOMM'98 Conf.* Vancouver, British Columbia, Canada: ACM Press, New York, NY, USA, Sep. 1998.

[8] E. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, 1959.

[9] M. Garey and D. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W.H Freeman and Co., 1979.

[10] R. M. Karp, "An introduction to randomized algorithms," *Discrete Applied Mathematics*, vol. 34, no. 1-3, pp. 165–201, Nov. 1991.

[11] R. Keller, J. Ramamirtham, T. Wolf, and B. Plattner, "Active Pipes: Service Composition for Programmable Networks," in *Proc. of IEEE MILCOM*, Oct. 2001.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. Kaashoek, and C. Modular, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18(3), Aug. 2000.

[13] R. Motwani and P. Raghavan, *Randomized Algorithms*. New York, NY: Cambridge University Press, 1995.

[14] L. Ruf, R. Keller, and B. Plattner, "A Scalable High-performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors," in *Proc. of 2004 ACS/IEEE Int. Conf. on Pervasive Services (ICPS'2004), Beirut, Lebanon*. IEEE, Jul. 2004.

[15] L. Ruf, T. Wolf, K. Farkas, and B. Plattner, "Mapping Network Services On Heterogeneous Multiprocessor Devices," ETH Zürich, Switzerland, Technical Report 248, Mar. 2006.

[16] Z. Wang and J. Crowcroft, "Quality of Service Routing for Supporting Multimedia Applications," in *JSAC*, vol. 14. Institute of Electrical and Electronics Engineers, Sept. 1996.

[17] N. Weng and T. Wolf, "Profiling and Mapping of Parallel Workloads on Network Processors," in *Proc. of The 20th Annual ACM Symp. on Applied Computing (SAC)*, Santa Fe, NM, Mar. 2005.