Application Analysis and Resource Mapping for Heterogeneous Network Processor Architectures

Ramaswamy Ramaswamy, Ning Weng and Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
{rramaswa,nweng,wolf}@ecs.umass.edu

Abstract

Network processors use increasingly heterogeneous processing resources to meet demands in performance and flexibility. These general-purpose processors, co-processors, and hardware accelerators pose a challenge to the software developer as application components need to be mapped to the appropriate resource for optimal performance. To simplify this task, we provide a methodology to automatically derive an architecture-independent application representation from a run-time instruction trace. This is done by considering data and control dependencies between each instruction of the trace. By using a novel clustering algorithm, called maximum local ratio cut, we group the instructions according to their dependencies and mutual cohesiveness. The resulting annotated directed acyclic graph (ADAG) gives insights into the application behavior and its inherent parallelism (multiprocessing vs. pipelining). The ADAG can further be used to map and schedule the application to a network processor system.

1 Introduction

Computer networks have progressed from simple store-and-forward communication networks to more complex systems. Packets are not only forwarded, but also processed on routers in order to be able to implement increasingly complex protocols and applications. Examples for such processing are network address translation (NAT) [8], firewalls [22], web switches [3], TCP/IP offloading for high-performance storage servers [12], and encryption for virtual private networks (VPN).

To handle the increasing functional and performance requirements, router designs have moved away from hard-wired ASIC forwarding engines. Instead, software-programmable network processors (NPs) have been developed in recent years. These NPs are typically single-chip multiprocessors with high-performance I/O components. A network processor is usually located on each input port of a router. Packet processing tasks are performed on the network processor before the packets are passed through the router switching fabric and on to the next network link. This is illustrated in Figure 1. Commercial examples for such systems are the Intel IXP2800 [15], IBM PowerNP [1], and EZchip NP-1 [10].

Due to the performance demands on NP systems, not only general-purpose RISC processor cores are used, but also a number of specialized co-processors. It is quite common to find coprocessors for checksum computation, address lookup, hash computation, encryption and authentication, and memory management functions. This leads to network processor architectures with a number of different processing resources. The trend towards more heterogeneous NP architectures will continue with the advances in CMOS technology as an increasing number of processing resources can be put on an NP chip. This allows for NP architectures with more co-processors; particularly those which implement more specialized, less frequently used functions.

The heterogeneity of NP platforms poses a particularly difficult problem for application development. Current software development environments (SDKs) are already difficult to use and require an in-depth understanding of the hardware architecture of the NP system (something that traditionally has been abstracted by SDKs). Emerging NP systems with a large number of heterogeneous processing resources will make this problem increasingly difficult as the program developer will have to make choices on which hardware units to use for which tasks. Such decisions can have significant impact on the overall performance of the system as poor choices can cause contention on resources. One way to alleviate this problem is to profile and analyze the NP applications and make static or run-time deci-

sions on how to assign processing tasks to a particular NP architecture. This process of identifying and mapping processing tasks to resources is the topic of this paper.

Our approach to this problem is to analyze the run-time characteristics of NP applications and develop an abstract representation of the processing steps and their dependencies. This creates an "annotated acyclic directed graph" (ADAG), which is an architecture-independent representation of an application. The annotations indicate the processing requirements of each block and the strength of the dependency between blocks. The basic idea is that we build the application representation "bottom-up." We consider each individual data and control dependency between instructions and group them into larger clusters, which make up the ADAG. The ADAG can then be used to determine an optimal allocation of processing blocks to any arbitrary NP architecture. Our contributions are:

1. A methodology for automatically identifying processing blocks from a run-time analysis of NP applications.

2. An algorithm to group "cohesive" processing blocks into processing clusters and a heuristic to efficiently approximate this NP-complete problem. The result is an application graph (ADAG) that is an architecture-independent description of the processing requirements.

3. A mapping (or "scheduling") algorithm to dynamically allocate processing clusters to processing resources on arbitrary network processor architectures.

We present the results for all these points using four realistic applications. One of the key points of this work is that the ADAG can be created *completely automatically* from a run-time instruction trace of the program on a uni-processor system.

The results from this work can help in a number of ways. The ability to identify cohesive processing blocks in a program is crucial to support high-level programming abstractions on heterogeneous NP platforms. Also, quantitative descriptions of the processing steps in terms of processing complexity and amount of communication between processing steps are the basis for any efficient scheduling. Further, the ADAG representation of an application gives very intuitive insights into the type of parallelism present in the application (e.g., multiprocessing vs. pipelining). Finally, the proposed scheduling algorithm can map packets at run-time to processing resources, which is superior to static approaches, which are dependent on an a-priori knowledge of traffic patterns.

In Section 2, we briefly discuss related work. Section 3 discusses the run-time analysis of NP applications and how we obtain the necessary profiling information. To get from the profiling information to an ADAG representation, we use a clustering algorithm that is discusses in Section 4. The results of the application analysis and the resulting ADAGs are presented for four applications in Section 5. The scheduling algorithm that maps the ADAGs to processing resources is presented in Section 6. Finally, Section 7 summarizes and concludes this paper.

2 Related Work

There has been some work in the area of application analysis and programming abstraction for network processors. Shah *et al.* have proposed NP-Click [29] as an extension to the Click modular router [18] that provides architecture and program abstractions for network processors. This and similar approaches require an a-priori understanding of application details in order to derive application modules. The problem with this approach is twofold:

- It assumes that the application developer can identify all application properties, which requires a deep understanding and careful analysis of the application. This will become an increasing problem as programming environments for network processors move towards higher-level and more complex abstractions. In such systems the application developer has less understanding on how a piece of code can be structured to match the underlying hardware infrastructure.

- No run-time information involved in application analysis. This is a crucial piece of information to use in making scheduling decisions. Using static application information only biases the results towards particular programming abstractions.

Therefore, we feel that it is crucial that application characteristics can be derived automatically as we propose in our work.

In NEPAL [21], Memik *et al.* propose a run-time system which controls the execution of applications on a network processor. Applications are separated into modules at the task level using the NEPAL API and modules are mapped to execution cores dynamically. There is an extra level of translation during which the application code is converted to modules. We avoid this translation and work directly with the dynamic instruction trace to generate basic blocks at a fine grained level.

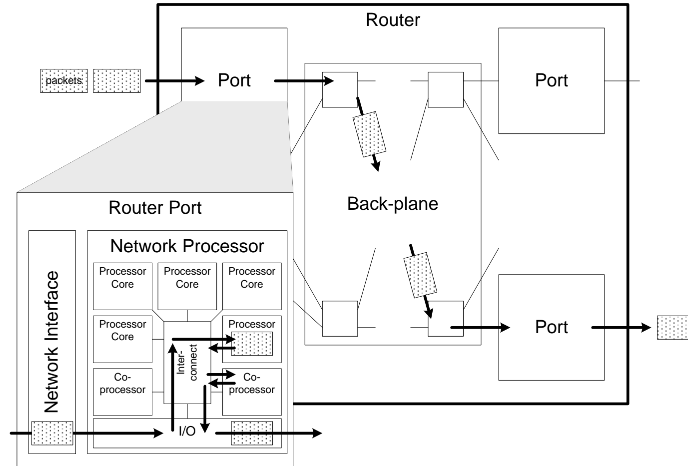


Figure 1: Packet Data Path on Network Router. Packets are shown as shaded boxes. Packet processing is performed on a network processor that is located at the input port of the system. The NP has a number of heterogeneous processing resources (processors cores and co-processors).

The problem of partitioning and mapping applications has been studied previously in the context of grid computing. Kenjiro *et al.* [30] present an approach that is similar to ours, but is used at a very high level to map an application on several heterogeneous computing resources on the Internet.

Our work is similar to some of the ideas used in trace scheduling [11] and superblocks [13], but is applied to a different domain. Trace scheduling aims to generate optimized code for VLIW architectures by exploiting more instruction level parallelism. Linear sequences of basic blocks are grouped to form larger basic blocks and dependencies are updated. We use a clustering algorithm for this purpose which works on a dynamic instruction trace after all basic blocks have been identified.

The exploration of application characteristics, modularization, and mapping to resources has also been studied extensively in the context of multi-processor systems. An example for scheduling of DAGs on multiprocessors is [20]. In the network processing environment, applications are much smaller and a more careful analysis can be performed. The bottom-up approach that we propose in Section 4 is not feasible for very large applications, but yields very detailed application properties for packet processing functions. Another difference between multiprocessors and NPs is that NPs aim at achieving high throughput rather than short delay in execution.

Mapping and scheduling of DAGs or task graphs has been surveyed by Kwok *et al.* in [19] and we propose a scheduling algorithm similar to that introduced by El-Rewini *et al.* in [9]. The main difference in both cases is that we are considering at a heterogeneous processing platform, where tasks can take different amounts of processing time (e.g., depending on the use of general-purpose processors vs. co-processors). Similar algorithms have also been used in the VLSI CAD area. Clustering approaches similar to ours have been surveyed by Alpert *et al.* in [2]. The methodology of clustering functionality proposed by Karypis *et al.* in [16] is similar to ours, but applied to the VLSI domain.

3 Application Analysis

Our goal is to generate workload models that can be used for network processor scheduling independent of the underlying hardware architecture. There are a number of approaches that have been developed for multiprocessor systems, real-time systems, and compiler optimization that characterize applications, and our approach uses some of these well-known concepts. The main difference is that network processing applications are very simple and execute a relatively small number of instructions (as compared to workstation applications). This allows us to use much more detailed analysis methods that would be infeasible for large programs. Also, there are a few issues that are specific to the NP domain and usually are not considered for workstation applications. When exploiting parallelism in NP applications, we do not necessarily have to use multiple parallel processors for one packet, but we can also use pipelining. Additionally, the heterogeneity of processing resources requires that we can identify the portions of an application that can be executed on specialized co-processors.

3.1 Static vs. Dynamic Analysis

One key question is whether to use a static or a dynamic application analysis as the basis for this work. With a static analysis, detailed information about every potential processing path can be derived. All processing blocks can be analyzed – even the ones that are not or hardly used during run-time. A static analysis typically results in a "call-graph," which shows the static control dependencies (i.e., which function can call which other function). This gives a good basic understanding of the application structure, but does not yield any information on its run-time behavior. But run-time behavior is exactly what is crucial for network processor performance.

A run-time analysis of the application (e.g., an instruction trace) shows exactly which instructions were executed and which instruction blocks were not used at all. In addition, all actual load and store addresses are available, which can be used for an accurate data dependency analysis. The drawbacks of run-time analysis is that each packet could potentially cause a different sequence of execution. In a few cases certain blocks are executed a different number of times depending on the size of the packet (we show this effect below in the context of packet encryption). Thus, the results are specific to a particular packet.

We have chosen to follow the path of run-time analysis due to the fact that it more accurately reflects the actual processing as well as provides actual load and store addresses, which are important when determining data dependencies. To address the issue of variations in network traffic processing even within the same application (e.g., different packet services or number of loop executions), there are several solutions. One is to assume the packet uses the most common execution path and if not an exception is raised and processing is continued on the control processor. This is currently done on some network processors (e.g., if IP options are detected in an IP forwarding application). Another approach is to analyze a large number of packets and find the union of all execution paths. By scheduling the union on the network processor system it is guaranteed that all packets can be processed, but the drawback is a lower system utilization as not all components will be used at all times. In this work, we focus on the analysis of a single packet for each application with the understanding that the work can be extended to consider a range of network traffic.

Another issue that arises from a run-time analysis is that it necessarily is done on compiled code. This introduces a certain bias towards a particular compiler and instruction set architecture. In our analysis, we only used compiler optimizations that are independent of the target system (e.g., no loop unrolling). Together with the use of a general RISC instruction set, the assumption is that the analysis yields results that are generally applicable to most processing engines in current network processors.

3.2 Annotated Acyclic Directed Graphs

The result of the application analysis needs to yield an application representation that is independent of the underlying architecture and can later be used for mapping and scheduling. For this purpose, we use an annotated directed acyclic graph, which we call ADAG. The ADAG represents processing steps (or blocks) as vertices and dependencies as edges. The processing steps are *dynamic* processing steps, i.e., they represent the instructions that are actually executed and loops cause the generation of processing blocks for each iteration. Only by considering the dynamic instances of each processing block can we obtain an acyclic graph. Also, it is desirable to only consider the instructions that are actually executed rather than any "dead code." The dependencies that we consider are data dependencies as well as control dependencies. There are two key parameters in an ADAG:

- Node weights: These indicate the amount of processing that is performed on each node (e.g., number of RISC instructions).
- Edge weights: These indicate the amount of state that is transferred between processing blocks.

Such an ADAG fully describes the processing and communication relationship for all processing blocks of an application.

3.3 Application Parallelism and Dependencies

In order to make use of the parallelism in a network processor architecture, the ADAG should have as few dependencies between processing blocks as possible. This increases the potential for parallelizing and pipelining processing tasks. At the same time, dependencies that are inherent to the application must not be left out to assure a correct representation.

We consider the following dependencies in our run-time analysis:

- Data dependencies: If an instruction reads a certain data location then it becomes a dependent to the most recent instruction that wrote to this location. Note that any type of memory, including registers, needs to be considered.

- Control dependencies: If a branch occurs due to a computation (i.e., a conditional branch) then the branch target is dependent on the instructions that compute the condition for the branch. Note that unconditional branches do not cause dependencies as they are static and any potential dependencies between two blocks would be covered by data dependencies.

Note that the above dependencies do not include anything related to resources. Resource conflicts are results of the underlying hardware and not a property of the application and thus not considered. Also other "hazards" (e.g., write-after-read (WAR)) do not need to be considered, because the run-time trace is a correct execution of the application where all hazards have been resolved ahead of time.

The result of the dependency analysis is an annotated run-time trace as shown in Figure 2. The trace sample is taken from an IPv4 forwarding implementation (details can be found below in the Section 5). As is shown in Figure 2, data dependencies are tracked across registers as well as memory locations. Also, control dependencies between basic blocks are shown. Note that the resulting graph is directed and acyclic since dependencies can only "point downward" (i.e., no instruction can ever depend on a later instruction).

Since the dependencies are limited to the absolute necessary dependencies (i.e., data and control), we get a DAG that is as sparse as possible and thus exhibits the maximum amount of parallelism. By focusing on these dependencies only, it is possible to find parallelism in the application despite the serialization that was introduced by running it on a uni-processor simulator.

Note that the analysis is done as a post-processing step of an instruction trace from simulation. This trace contains effective memory addresses and information about all status bits that are changed during execution. This means that there is no need for memory disambiguation.

3.4 ADAG Reduction

A practical concern of this methodology is that the number of processing blocks is large (in the order of the total instructions executed) and the representation of the DAG becomes unwieldy. Therefore, we take a simplifying step that significantly reduces the number of processing steps: instead of considering individual instructions, we consider basic blocks. A basic block is a group of instructions that are executed in sequence and has no internal control flow change. That is, the execution of a program can only jump to the beginning of a basic block and cannot jump somewhere else until the end of the basic block is reached. Still, all necessary dependencies are considered, but the smallest code fragment that can be parallelized or pipelined is a basic block. In Figure 2, basic blocks are separated by dashed lines.

Even with a reduction to basic blocks, the resulting ADAG is not a suitable representation of an application, because it does not capture any higher-level application properties. The dependency between processing blocks can be very different depending on the nature of the application. Most applications show a "natural" separation between parts of the application (e.g., checksum verification and destination address lookups in IP forwarding), while showing a strong dependency within a particular part (e.g., all basic blocks of checksum computation). In order to consider such "clustering," we further reduce the ADAG with the algorithm described in the following section.

4 ADAG Clustering Using Maximum Local Ratio Cut

When assigning processing steps to a network processor architecture, there are several points that need to be considered. Most of all, there is a tradeoff between the cost of processing (or the speedup that is gained by using a co-processor) and the cost of communication. This implies that it is not desirable to offload small processing blocks to co-processors, especially when this requires a large amount of communication.

The ADAG generation above results in a graph with thousands of basic blocks. The dependencies between them can cause a large amount of communication if the basic blocks were to be distributed to different computational resources. Thus, using the above ADAG directly for workload mapping is not suitable. Instead, we want to reduce the number of processing components in the ADAG to yield a more natural, tractable grouping of processing instructions. For this purpose, we use a clustering technique called "ratio cut" [32]. Ratio cut has the nice property of identifying "natural" clusters within a graph without the need for a-priori knowledge of the final number of clusters.

The ratio cut algorithm is unfortunately NP-complete and thus not tractable for ADAGs with the number of nodes that we need to consider here. Therefore, we propose a heuristic that is based on ratio cut and reduces the computational complexity while still achieving good results. Our heuristic is called "maximum local ratio cut" (MLRC).

```
Inst.-#   Address      Instruction          Effective Address
...
129       33557096 : ldrb  r3,[r4,#8]         : 0x33977912
130       33557100 : cmp   r3,#0              : 0x--------
131       33557104 : bne   0x2000aa4          : 0x--------
132       33557156 : sub   r3,r3,#1           : 0x--------
133       33557160 : strb  r3,[r4,#8]         : 0x33977912
134       33557164 : mov   r2,#65280          : 0x--------
135       33557168 : ldr   r3,[r4,#8]         : 0x33977912
136       33557172 : add   r2,r2,#254         : 0x--------
137       33557176 : mov   r3,r3, lsr #16     : 0x--------
138       33557180 : cmp   r3,r2              : 0x--------
139       33557184 : mov   r2,#1              : 0x--------
...
```

Figure 2: Instruction Trace Analysis Example. Data dependencies between writes and reads in registers and memory locations are shown. Also, control dependencies for conditional branches are shown.

4.1 Clustering Problem Statement

Before discussing the algorithm, let us formalize the problem that we address here. The ADAG, $A_n = (P_n, D_n)$, consists of a "processing vector," P_n, and a "dependency matrix," D_n. The processing vector contains the number of instructions that are executed in each of the n processing blocks. The dependency matrix contains the data values that need to be transferred between each pair of blocks. If d_{ij} is non-zero, the block i depends on block j because it reads d_{ij} data values that are written by block j. (Control dependencies are considered to be one-value dependencies.) The n blocks in A_n are ordered in such a way that the upper right of the dependency matrix is zero, which ensures that A_n is a directed acyclic graph. Thus, we have:

$$A_n = (P_n, D_n) \quad with \quad P_n = (p_1, \ldots, p_n) \quad and$$
$$D_n = \begin{pmatrix} 0 & \cdots & \cdots & 0 \\ d_{21} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ d_{n1} & \cdots & d_{nn-1} & 0 \end{pmatrix} \quad (1)$$

The goal of the clustering process is to generate a new ADAG, $A_{n'}$, which is based on A_n, but smaller ($n' < n$). Sets of nodes from A_n can be combined to clusters, which then become nodes in $A_{n'}$. If m nodes $i_1 \ldots i_m$ are combined to a cluster node j, then $p_j = \sum_{k=1}^{m} p_{i_k}$. The nodes x on which j depends are updated such that $d_{jx} = \sum_{k=\{i_1 \ldots i_m\}} d_{kx}$. The dependents y of j are updated accordingly to $d_{yj} = \sum_{l=\{i_1 \ldots i_m\}} d_{yk}$. Basically, if nodes are clustered, then the new cluster combines the properties of all its nodes: the processing costs are added together and the dependencies are combined. A clustering step can only be performed if the resulting graph is still a directed acyclic graph (i.e., the dependency matrix can be reordered to have the upper right be zero).

This clustering can be performed repeatedly in order to reduce n to the desired number of total clusters. Next, we discuss ratio cut, which is an algorithm to determine *which* nodes should be clustered and how many clusters the final solution contains.

4.2 Ratio Cut

The basic concept of ratio cut is to cluster together nodes that show some natural "cohesiveness," as described in detail in [32]. In our context, ratio cut clusters instruction blocks together such that

- clusters perform a significant amount of processing and
- clusters have little dependencies between them.

The metric to determine these properties is the ratio cut, r_{ij}, for two clusters i and j, which is defined as[1]:

$$r_{ij} = \frac{d_{ij} + d_{ji}}{p_i \times p_j}. \quad (2)$$

The ratio cut algorithm will cluster the graph such that r_{ij} is *minimized*. This means the dependencies between i and j are small and the amount of processing that is done i and j is large. Note that either d_{ij} or d_{ji} has to be zero due to the acyclic property of an ADAG.

Ratio cut operates in a top-down fashion. Starting from one cluster that contains all nodes (i.e., A_1), the ratio cut is applied to find two groups that minimize r_{ij}. Then this process is applied recursively within each group. With each recursion step, the minimum ratio cut value will increase (because the clusters will have less and less clear separations). The clustering process can be terminated when the ratio cut exceeds

[1] The ratio cut described in [32] uses nodes of uniform size and thus $r_{ij} = \frac{d_{ij}+d_{ji}}{|i| \times |j|}$. We are not interested in the number of blocks that are in each cluster, but the amount of processing that is performed. Thus, we adapted the definition of r_{ij} accordingly.

a certain threshold ($r_{ij} > t_{terminate}$). The value of this threshold determines how "tightly" clustered the result is. If $t_{terminate}$ is small, then only few clusters will be found, but the separations between them will be very clear (i.e., little dependency). If $t_{terminate}$ is large, then many clusters will be found (all n blocks in the limit) and the dependencies between them can be significant (i.e., requiring large data transfers). In neither case, the exact number of clusters is predetermined. This is why ratio cut is considered an algorithm that finds a "natural" clustering that depends on the properties of the graph (i.e., r_{ij}).

While ratio cut is an ideal algorithm for our purposes, it has one major flaw. It is NP-complete (for proof see [32]). Basically, it is necessary to consider an exponential number of potential clusterings in each step. This makes a practical implementation infeasible. The heuristic that have been proposed in [32] are also not suitable as they assume and require the graph to undirected, which is not the case for our ADAG.

To address this problem, we propose a heuristic that uses the ratio cut metric, but is less computationally complex.

4.3 Maximum Local Ratio Cut

Instead of using the top-down approach that requires the exploration of a number of possible clusterings that grows exponentially with n, we propose to use bottom-up approach in our heuristic, which we call "maximum local ratio cut" (MLRC). It is called "local" ratio cut, because MLRC makes a local decision when merging nodes. MLRC operates as follows:

1. Start with ADAG, $A_i = A_n$, that has all nodes separated.
2. For each pair (i, j) compute the local ratio cut r_{ij}.
3. Find the pair (i_{max}, j_{max}) that has the maximum local ratio cut.
4. If the maximum ratio cut drops below the threshold ($r_{i_{max}j_{max}} < t_{terminate}$) stop the algorithm. A_i is the final result.
5. Merge i and j into a cluster resulting in $A_{i_{new}} = A_{i-1}$.
6. Set $A_i = A_{i_{new}}$ and repeat steps (2) through (6).

The intuition behind MLRC is to find the pair of nodes that should be least separated (i.e., one that has a lot of dependency and does little processing). This pair is then merged and the process applied recursively. As a result, clusters will form that show a lot of internal dependencies and little dependencies with other clusters.

Of course, this is a heuristics and therefore cannot find the best solution for all possible ADAGs. The following intuition argues why MLRC performs well:

- If two nodes show a large ratio between them, it is likely that they belong to the same cluster in the optimal ratio cut solution.
- By merging two nodes that exhibit a high local ratio cut, the overall ratio cut of A is reduced (in most cases), which leads to a better solution overall.
- The termination criterion is similar to that of ratio cut and leads to a similarly "natural" clustering.

We show results for four applications in Section 5 that show the performance of the algorithm for realistic inputs.

4.4 MLRC Complexity

The maximum local ratio cut algorithm has a complexity that is tractable and feasible to implement. The algorithm runs over at most n iterations (in case $t_{terminate}$ is not reached until the last step). In each iteration the ratio cut for $i^2/2$ pairs needs to be computed (which takes $O(1)$). Finding the maximum can easily be done during the computation. Thus, the total computational complexity is

$$\sum_{i=1}^{n} \frac{i^2}{2} \times O(1) = O(n^3). \tag{3}$$

The space requirement for MLRC is $O(n^2)$, which is the same complexity that is required to represent A_n. Thus, MLRC is a feasible solution to the NP-complete ratio cut algorithm. In the following section, we show the performance of MLRC on a set of network processing applications.

5 ADAG Results

To illustrate the behavior and results of the application analysis, we use a set of four network processing applications. We briefly discuss the tool that we use to derive run-time traces and the details of the applications. Then we show the clustering process for one application and the final results for all four applications.

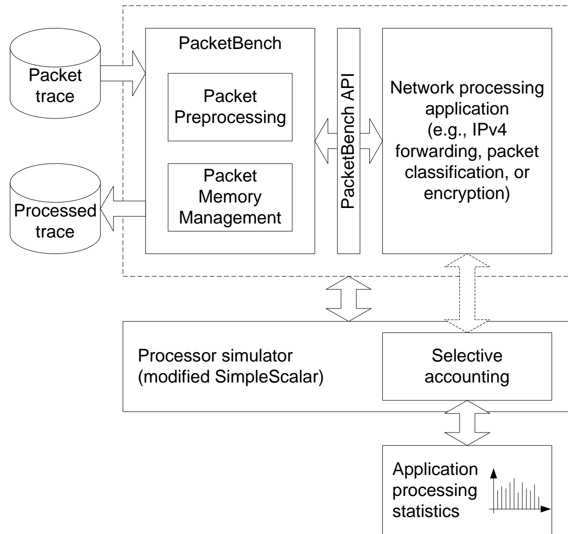


Figure 3: PacketBench Architecture. The application implements the packet processing functionality that is measured. PacketBench provide support functions for packet and memory management. The simulator generates an instruction trace for the application (and not the framework) through selective accounting.

5.1 The PacketBench Tool

In order to obtain runtime analysis of application processing, we use a tool called "PacketBench" that we have developed [28]. The goal of PacketBench is to emulate the functionality of a network processor and provide an easy to use environment for implementing packet processing functionality. The conceptual outline of the tool is shown in Figure 3. The main components are:

- **PacketBench Framework.** The framework provides functions that are necessary to read and write packets, and manage memory. This involves reading and writing trace files and placing packets into the memory data structures used internally by PacketBench. On a network processor, many of these functions are implemented by specialized hardware components and therefore should not be considered part of the application.

- **PacketBench API.** PacketBench provides an interface for applications to receive, send, or drop packets as well as doing other high-level operations. Using this clearly defined interface makes it possible to distinguish between PacketBench and application operations during simulation.

- **Network Processing Application.** The application implements the actual processing of the packets. This is the processing that we are interested in as it is the main contributor to the processing delay on a router (e.g., packet classification for firewalling or encryption for VPN tunneling). The workload characteristics of the application needs to be collected separately from the workload generated by the PacketBench framework.

- **Processor Simulator.** To get instruction-level workload statistics, we use a full processor simulator. In our current prototype we use SimpleScalar [6], but in principle any processor simulator could be used. Since we want to limit the workload statistics to the application and not the framework, we modified the simulator to distinguish operations accordingly. The Selective Accounting component does that and thereby generates workload statistics as if the application had run by itself on the processor. This corresponds to the actual operation of a network processor, where the application runs by itself on one of the processor cores. Additionally, it is possible to distinguish between accesses to various types of memory (instruction, packet data, and application state), which is useful for a detailed processing analysis.

The key point about this system design is that the application and the framework can be clearly distinguished – even though both components are compiled into a single executable in order to be simulated. This is done by analyzing the instruction addresses and sequence of API calls. This separation allows us to adjust the simulator to generate statistics for the application processing and ignore the framework functions. This is particularly important as network processing consists of simple tasks that execute only a few hundred instructions per packet [33]. Also, in real network systems the packet management functions are implemented in dedicated hardware and not by the network processor and thus should not be considered part of the workload.

Another key benefit of PacketBench is the ease of implementing new applications. The architecture is modular and the interface between the application and the framework is well defined. New applications can be developed in C, plugged into the framework, and run on the simulator to obtain processing characteristics.

In our prototype, the PacketBench executable is simulated on a typical processor simulator to get statistics of the number of instructions executed and the number of memory accesses made. We use the ARM [4] target of the SimpleScalar [6] simulator, to analyze our applications. This simulator was chosen because the ARM architecture is very similar to the architecture of the core processor and the microengines found in the Intel

IXP1200 network processor [14], which is used commonly in academia and industry. The tools were setup to work on an Intel x86 workstation running RedHat Linux 7.3. PacketBench supports packet traces in the *tcpdump* [31] format and the Time Sequenced Header (TSH) format from NLANR [24]. The latter trace format does not contain packet payloads, so we have the option of generating dummy payloads of the size specified in the packet header. For the experiments that we perform in this work, the actual content of the payload is not relevant as no data-dependent computations are performed.

The run-time traces that we obtain from PacketBench contain the instructions that are executed, the registers and memory locations that are accessed, and an indication of any potential control transfer. Using these traces we build an ADAG that considers dependencies among instructions as well as allows us to discover any potential parallelism. Since we make no assumption on the processing order other than the dependencies between data (see next subsection), we are able to represent the application almost independently from a particular system.

5.2 Applications

The four network processing applications that we evaluate range from simple forwarding to complex packet payload modifications. The first two applications are IP forwarding according to current Internet standards using two different implementations for the routing table lookup. The third application implements packet classification, which is commonly used in firewalls and monitoring systems. The fourth application implements encryption, which is a function that actually modifies the entire packet payload and is used in VPNs. The specific applications are:

- **IPv4-radix.** IPv4-radix is an application that performs RFC1812-compliant packet forwarding [5] and uses a radix tree structure to store entries of the routing table. The routing table is accessed to find the interface to which the packet must be sent, depending on its destination IP address. The radix tree data structure is based on an implementation in the BSD operating system [25].

- **IPv4-trie.** IPv4-trie is similar to IPv4-radix and also performs RFC1812-based packet forwarding. This implementation uses a trie structure with combined level and path compression for the routing table lookup. The depth of the structure increases very slowly with the number of entries in the routing table. More details can be found in [27].

- **Flow Classification.** Flow classification is a common part of various applications such as firewalling, NAT, and network monitoring. The packets passing through the network processor are classified into flows which are defined by a 5-tuple consisting of the IP source and destination addresses, source and destination port numbers, and transport protocol identifier. The 5-tuple is used to compute a hash index into a hash data structure that uses linked lists to resolve collisions.

- **IPSec Encryption.**IPSec is an implementation of the IP Security Protocol [17], where the packet payload is encrypted using the Rijndael algorithm [7], which is the new Advanced Encryption Standard (AES) [23]. This algorithm is used in many commercial VPN routers. This is the only application where the packet payload is read and modified. It should be noted that the encryption processing for AES shows almost identical characteristics as the decryption processing. We do not further distinguish between the two steps.

The selected applications cover a broad space of typical network processing. IPv4-radix and IPv4-trie are realistic, full-fledged packet forwarding applications, which perform all required IP forwarding steps (header checksum verification, decrementing TTL, etc.). IPv4-radix represents a straight-forward unoptimized implementation, while IPv4-trie performs a more efficient IP lookup. The applications can also be distinguished between header processing applications (HPA) and payload processing applications (PPA) (as defined in [33]). HPA process a limited amount of data in the packet headers and their processing requirements are independent of packet size. PPA perform computations over the payload portion of the packet and are therefore more demanding in terms of computational power as well as memory bandwidth. IPSec is a payload processing application and the others are header processing applications. The applications also vary significantly in the amount of data memory that is required. Encryption only needs to store a key and small amounts of state, but the routing tables of the IP forwarding applications are very large.

Altogether, the four applications chosen in this work are good representatives of different types of network processing. They display a variety of processing characteristics as is shown below.

To characterize workloads accurately, it is important to have realistic packet traces that are representative of the traffic that would occur in a real network. We use several traces from the NLANR repository [24] and our local intranet. The routing table for the IP lookup applications is MAE-WEST [26].

Application	Number of Basic Blocks (n)	Number of Unique Basic Blocks	Maximum Processing ($max(p_i)$)	Maximum Dependency ($max(d_{ij})$)
IPv4-radix	2340	375	29	40
IPv4-trie	37	28	13	11
Flow Class.	36	35	35	29
IPSec	267	93	89	82

Table 1: Results from Application Analysis.

5.3 Basic Block Results

The initial analysis of basic blocks and their dependencies yields the results shown in Table 1. Ipv4-radix executes the most number of instructions and has by far the most basic blocks. Note that the number of unique basic blocks is much smaller. This is due to the fact that many basic blocks are executed repeatedly during runtime. For Flow Classification almost all basic blocks are different indicating that there are no loops.

5.4 Clustering Results

Using the MLRC algorithm, the basic block ADAG is step-by-step decreased in size. Figure 4 shows the last 10 ($A_{10} \dots A_1$) steps of this process for the Flow Classification application. In each cluster, the name of the cluster (e.g., $c0$) and the processing cost (e.g. 25) are shown. The edges show the dependency between clusters (number of data transfers). Note that the cluster names change across figures due to the necessary renaming to maintain DAG properties (zeros in upper right of dependency matrix). The start nodes (i.e., nodes that are not dependent on any other nodes) are shown as squares. The end nodes (i.e., nodes that have no dependents) are shown with a thick border. The following can be observed:

- Aggregation of nodes causes the resulting cluster to have a processing cost equal to the sum of the nodes.
- Edges are merged during the aggregation.
- The number of "parallel" nodes decreases as the number of clusters decreases.

The first two observations follow the expected behavior of MLRC. The third observation is more interesting. The reduction in parallelism means that an application that has been clustered "too much" cannot be processed efficiently and in parallel on a network processor system. Therefore it is crucial to determine when to stop the clustering process.

In Figure 5, the progress of two metrics in the MLRC is shown for all four application. The plots show the value of the maximum local ratio cut ("local ratio cut") and the number of "parallel" nodes. The local ratio cut value decreases with fewer clusters – as is expected. In a few cases, the local ratio cut value increases after a merging step. This is due to MLRC being a heuristic and not an optimal algorithm. The initial local ratio cut value is 1. For our applications, this is the worst case (e.g., occurring when there are two one-instruction blocks with one dependency) since there cannot be more dependencies than instructions. The number of parallel nodes is derived by counting the number of nodes that have at least one other node in parallel (i.e., there is no direct or transitive dependency). These nodes could potentially be processed in parallel on an NP system. Eventually this value drops to zero. For IPv4-trie, and Flow Classification, this happens at around 5 clusters, for IPv4-radix at 20 clusters, and for IPSec at around 50 clusters. This indicates that IPv4-radix and IPSec are applications that lend themselves more towards pipelining than towards parallel processing.

5.5 Applications ADAGs

Figure 6 shows the ADAGs A_{20} for all four applications (independent of $t_{terminate}$. We can observe the following application characteristics:

- IPv4-radix is dominated by the lookup of the destination address using the radix tree data structure. This traversal of the radix tree causes the same loop to execute several times. Since we consider run-time behavior, each loop instance is considered individually. The patterns of processing blocks with 330, 181, 195, and 136 instructions in A_{20} show these instruction blocks. Another observation is that the lack of parallelism between blocks is indicative of the serial nature of an IP lookup. Even though the same code is executed, there are data dependencies in the prefix lookup, which are reflected in the one-data-value depen-

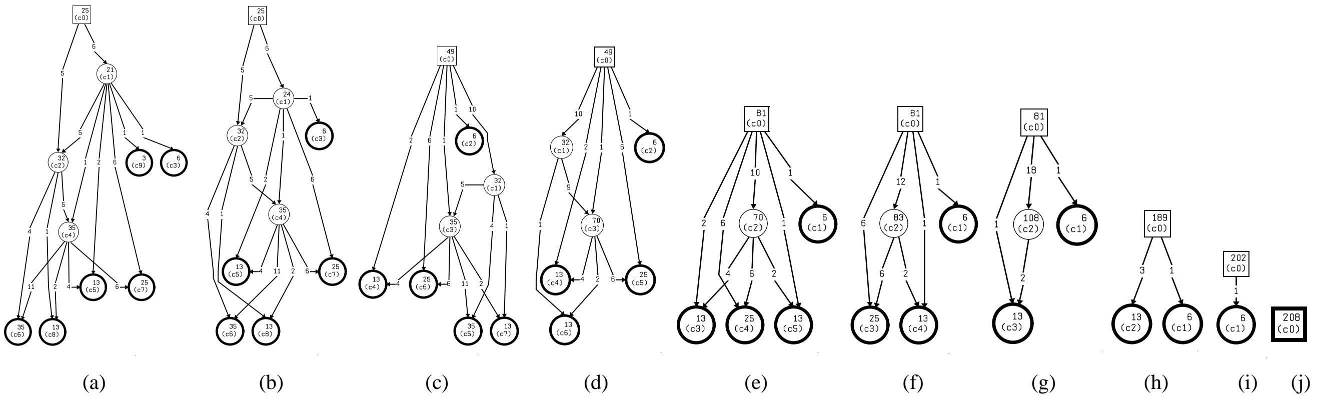


Figure 4: Sequence of ADAG Clustering. ADAG for Flow Classification is shown for 10 to 1 clusters. Each node shows the processing cost of the cluster and its name (e.g. 25 instructions for cluster $c0$ in (a)). Note that the clusters are renamed with each merging step.

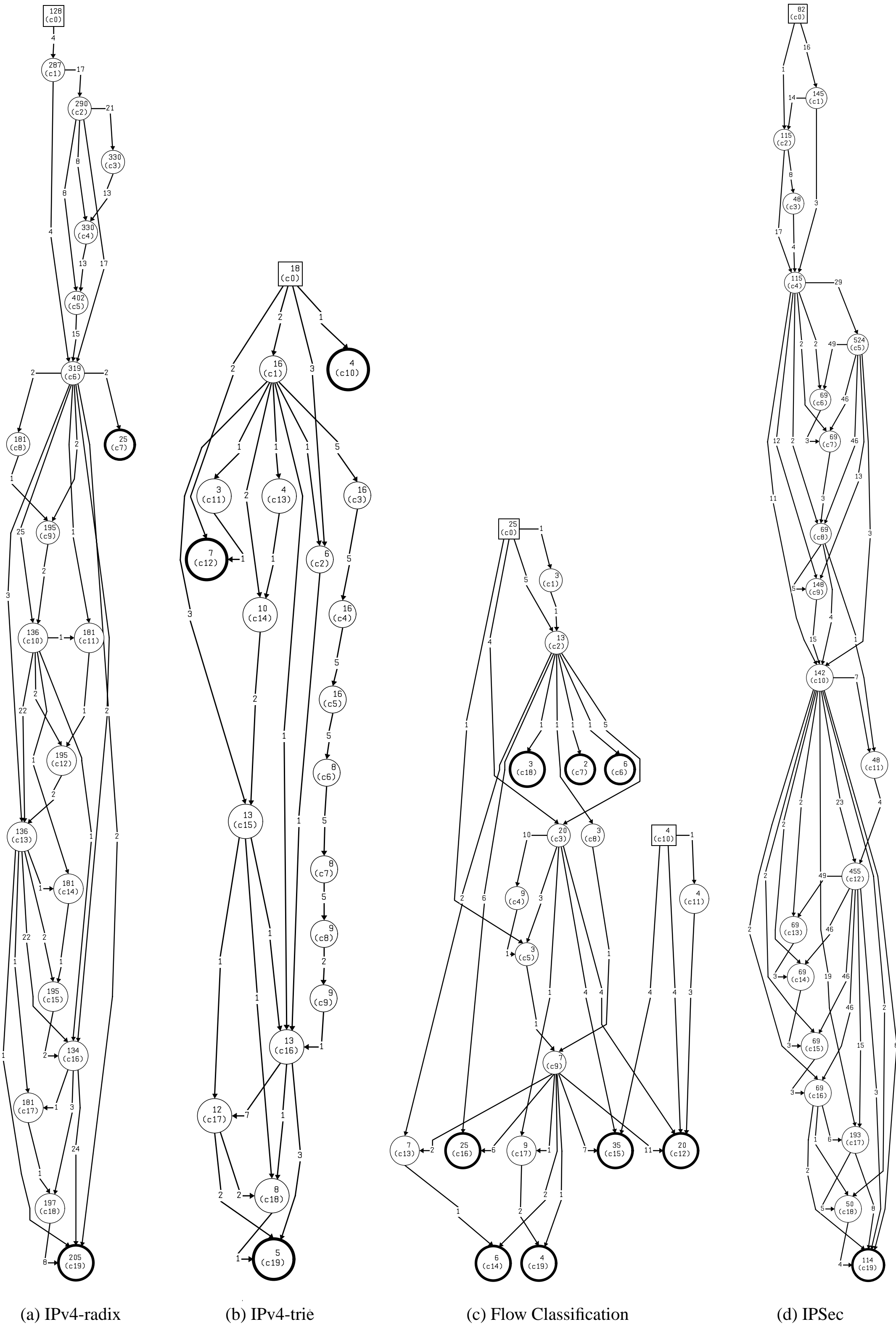
(a) IPv4-radix

(b) IPv4-trie

(c) Flow Classification

(d) IPSec

Figure 5: Local Ratio Cut Algorithm Behavior.



(a) IPv4-radix (b) IPv4-trie (c) Flow Classification (d) IPSec

Figure 6: ADAGs for Workload Applications.

dencies shown in Figure 6.

- IPv4-trie implements a simpler IP lookup than IPv4-radix. The lookup is represented by the sequence of clusters three to nine with mostly five-data-value dependencies. IPv4-trie exhibits more parallelism, but still is dominated by the serial lookup.

- Flow Classification has two start nodes and a number of end-nodes, where processing does not have any further dependents. These are write updates to the flow classification data structure. Altogether, there is a good amount of parallelism and less serial behavior than in the other applications.

- IPSec is extremely serial and the encryption processing repeatedly executes the same processing instructions, which are represented by the blocks with 69 instructions and 49 or 46 data dependencies going into the block. This particular example executes the encryption of *two* 32-byte blocks. The transition from the first to the second block happens in cluster 4. This application shows no parallelism as is expected for encryption.

5.6 Identification of Co-Processer Functions

The final question for application analysis is how to identify processing blocks that lend themselves for co-processor implementations. There are some functions which by default are ideal for co-processing that can be identified by the programmer (e.g., checksum computation due to its simplicity and streaming data access nature). We want to take a different look at the problem and attempt to identify such functions without a-priori understanding of the application.

The ADAGs only show how many instructions are executed by a processing block, but not which instructions. In order to identify if there are instruction blocks that are heavily used in an application, we use the plots shown in Figure 7. The x-axis shows the instructions that are executed during packet processing. The y-axis shows each unique instruction address observed in the trace. For example, in IPSec, the 400th unique instruction is executed sixteen time (eight times between instruction 500 and 1000 and eight times between 1500 and 2000).

Figure 7 is a good indicator for repetitive instruction execution. For Flow Classification, there are almost no repeated instructions. In IPSec, however, there are several instruction blocks that are executed multiple times (sixteen times for instructions with unique address 350 to 450). If these instructions can be implemented in dedicated hardware, a significant speed-up can be achieved due to the high utilization of this function.

Again, this method of co-processing identification requires no knowledge or deep understanding of the application. Instead the presented methodology extracts all this information from a simple instruction run-time trace. One problem with this methodology is that processing blocks that execute non-repetitive functions are not identified as suitable for co-processors, even though they could be (as it is the case for Flow Classification). Such functions still need to be identified manually by the programmer.

6 Mapping Application DAGs to NP Architectures

Once application ADAGs have been derived, they can be used in multiple ways. One way of employing the information from ADAGs is for network processor design. With a clear description of the workload and its parallelism and pipelining characteristics, a matching system architecture can be derived. Another example is the use of application ADAGs to map instruction blocks to NP processing resources. In this section, we discuss this mapping and scheduling in more detail.

6.1 Problem Statement

The mapping and scheduling problem is the following: Given a packet processing application and a heterogeneous network processor architecture, which processing task should be assigned to which processing resource (mapping) and at what time should the processing be performed (scheduling)?

For this problem, we assume that a network processor has m different processing resources $r_1 \ldots r_m$. These processors can be all of the same kind (e.g., all general-purpose processors) or can be a mix of general-purpose processors and co-processors. All processing resources are connected with each other over an interconnect. Transferring data via the interconnect incurs a delay proportional to the amount of data transferred. Thus, if the application uses multiple resources in parallel, the communication cost for the data transfer needs to be considered.

An application is represented by an ADAG with n clusters $c_1 \ldots c_n$ and their processing costs and dependencies. Since we have processing resources with different performance characteristics, the processing cost for a cluster is represented by a vector $p_i = (p_i 1, \ldots p_i m)$. This vector contains the processing cost

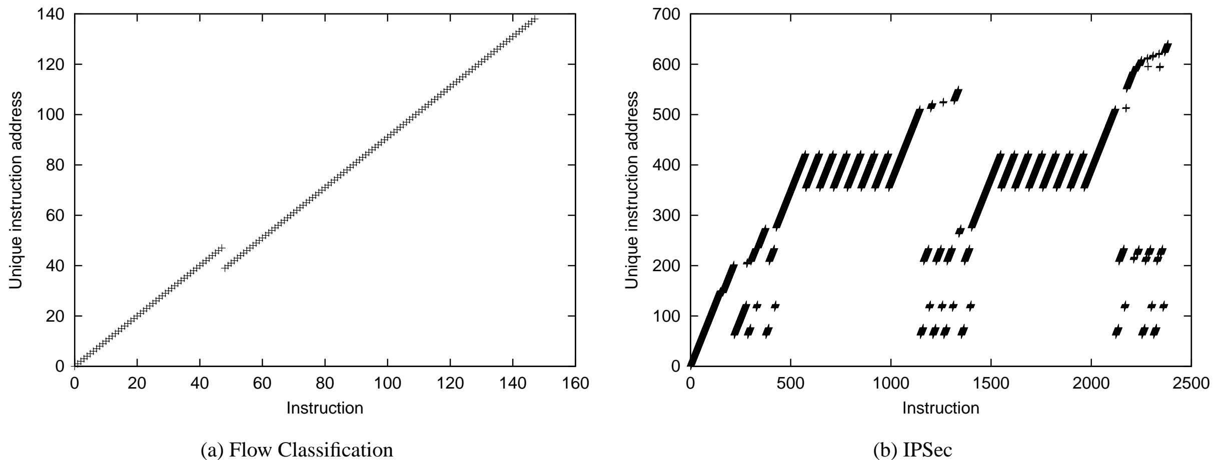


Figure 7: Detailed Instruction Access Patterns of a Single Packet for Flow Classification and IPSec from the MRA Trace.

for the cluster for each possible processing resource. If a cluster cannot be executed on a particular processing resource (e.g., checksum computation cannot be performed on a table-lookup co-processor, the processing cost is ∞).

The mapping solution, M, consists of n pairs that indicate the assignment of all clusters $c_1 \ldots c_n$ to a resource r_i: $M = ((c_1, r_{i_1}) \ldots (c_n, r_{i_n}))$. The schedule, S, is similar, except that is also contains a time t that indicates the start time of the execution of a cluster on a resource: $S = ((c_1, r_{i_1}, t_1) \ldots (c_n, r_{i_n}, t_n))$.

Finally, a performance criterion needs to be defined that is used to find the best solution. This could be shortest delay (i.e., earliest finish time of last cluster) or best resource usage (i.e., highest utilization of used resources) etc. We use minimum delay in our example.

Unfortunately, this problem, too, is NP complete. Malloy *et al.* established that producing a schedule for a system that includes both execution and communication cost is NP-complete, even if there are only 2 processing elements [20]. Therefore we need to develop a heuristic to find an approximate solution.

Mapping of task graphs to multiprocessors has been researched extensively and is surveyed by Kwok *et al.* in [19]. However, most of the previous work is targeted for homogeneous multiprocessor systems. Here, we consider the mapping of ADAGs onto a set of heterogeneous processing resources.

6.2 Mapping Algorithm

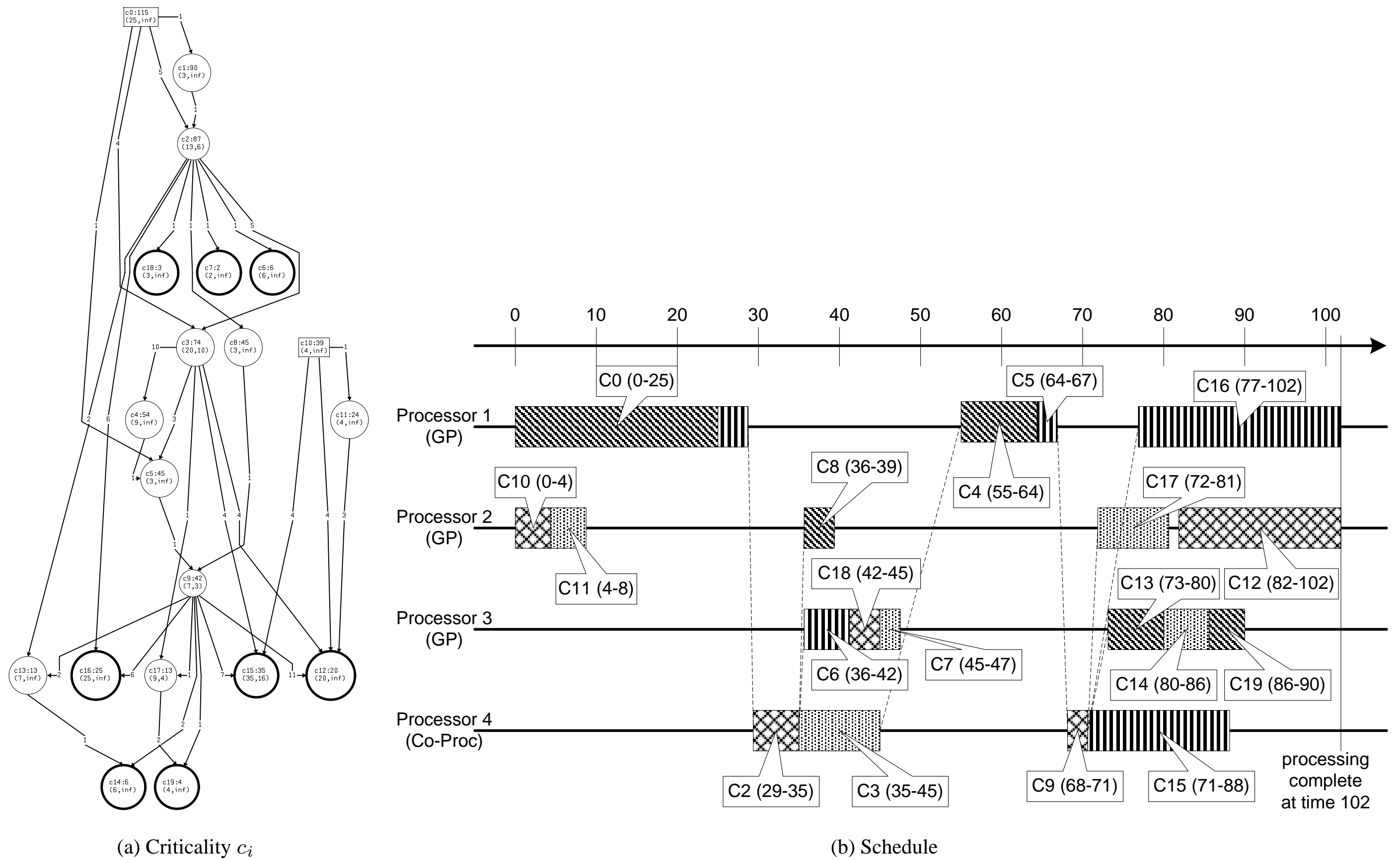
In our example, we consider the mapping of a single ADAG onto the processing resources. The goal is to map it in such a way as to minimize the overall finish time of the last cluster. This mapping also yields maximum use of the application's parallelism. We only consider one packet in this example, but the approach can easily be extended to consider the scheduling of multiple packets.

There are two parts to our mapping and scheduling algorithm. First, we identify the nodes that are most critical to the timely execution of the packet (i.e., the nodes that lie on the critical path). For this purpose we introduce a metric called the "criticality," c_i, of a node i. The criticality is determined by finding the critical path (bottom-up) in the ADAG. The criticality is determined by looking at the processing time of each cluster when using a general-purpose processor (we assume that this is resource 1). For each end node e (no children), the criticality is just its default processing time: $c_e = p_{e_1}$. For all other nodes i, the criticality is the maximum criticality of its children plus its own processing time: $c_i = \max c_j + p_i, \forall j : d_{ji} > 0$.

The clusters are then scheduled in order of their criticality such that each assignment achieves the minimum increase in the overall finish time. This requires that the current finish time of each node and resource has to be maintained. When determining the finish time of a cluster, f_i, the finish time of all its parents (on which it depends) needs to be considered as well as the delay due to data transfers between different processing resources over the interconnect.

Thus, the mapping and scheduling algorithm to heuristically find the earliest finish time of a processing application is:

1. Calculate each node's criticality c_i as defined above.



(a) Criticality c_i

(b) Schedule

Figure 8: Mapping and Scheduling Result for Flow Classification. The criticality graph shows the node name, the criticality c_i and the processing cost vector for general-purpose processors and the co-processor. The schedule shows which processing step is allocated to which processor and at what time the processing is performed.

2. Sort the nodes into a list L by decreasing criticality.

3. Dequeue node N with highest criticality from L

4. For each resource r_i determine the finish time for N by adding the maximum finish time of all parents (plus interconnect overhead) to the processing time, $p_N i$, of n on resource r_i. Assign N to the resources that minimizes the finish time.

5. Repeat steps (3) through (5) until L is empty.

The algorithm is developed based on the following observation: if one maps the critical path of the ADAG with minimal delay and all non-critical path nodes meet their deadlines, then the resulting schedule is optimal. So, the critical path gives us a global view of the ADAG, but mapping is done by local decisions to avoid exponential complexity.

This algorithm uses a greedy approach: given node N it tries to identify the processing element which yields the earliest finishing time by either (1) reducing communication cost and using the same resources as its parents or (2) by using a faster co-processor and paying for communication delay.

Our mapping and scheduling algorithm is based on the list scheduling techniques which are well explored under different assumptions and terminology. The criticality metric is similar to assigning a priority to the task as proposed by El-Rewini *et al.* [9] where is called "static bottom level." However, we use them for the heterogeneous system. Our metric, the early finishing time instead of early starting time, helps us explore the option of the fast processor when assigning one task to potential processing element.

6.3 Mapping and Scheduling Results

We show the results of this mapping and scheduling algorithm in Figure 8. It uses the A_{20} ADAG for Flow Classification and an NP architecture with four processors: three general purpose processors and one co-

processor that requires only half the instructions for some of the clusters (for illustration, these were picked randomly and don't reflect actual application behavior).

The schedule in Figure 8 completes the processing of the packet at time 102. This is shorter than the original criticality of the start node due to the use of the co-processor. Overall, it can be seen that the application parallelism is exploited and the processing resources of the network processor are used efficiently.

A further exploration of this algorithm and its impact on a scenario, where the optimization criterion is system throughput, is currently work in progress.

7 Summary and Conclusions

In this work, we have introduced an annotated directed acyclic graph to represent application characteristics and dependencies in an architecture-independent fashion. We have developed a methodology to automatically derive this ADAG from run-time instruction traces that can be obtained easily from simulations. To consider the natural clustering of instructions within an application, we have used maximum local ratio cut (MLRC) to group instruction blocks and reduce the overall ADAG size. For four network processing applications, we have presented such ADAGs and shown how the inherent parallelism (multiprocessing or pipelining) can be observed. Using the ADAG representation, processing steps can be allocated to processing resources using a heuristic that uses the node criticality as a metric. We have presented such a mapping and scheduling result to show its behavior.

We believe this is an important step towards automatically analyzing applications and mapping processing tasks to heterogeneous network processor architectures. For future work, we plan to further explore the issue of differences in run-time execution of packets and how it impacts the results from the analysis. We also want to compare the quality of the clustering obtained from Minimum Local Ratio Cut with that of other non-greedy ratio cut heuristics. Finally, it is necessary to develop a robust methodology for automatically identifying processing blocks for co-processors and hardware accelerators.

References

[1] J. Allen, B. Bass, C. Basso, R. Boivie, J. Calvignac, G. Davis, L. Frelechoux, M. Heddes, A. Herkersdorf, A. Kind, J. Logan, M. Peyravian, M. Rinaldi, R. Sabhikhi, M. Siegel, and M. Waldvogel. IBM PowerNP network processor: Hardware, software, and applications. *IBM Journal of Research and Development*, 47(2/3):177–194, 2003.

[2] C. Alpert and A. Kahng. Recent directions in netlist partitioning: A survey. *Integration: The VLSI Journal*, pages 1–81, 1995.

[3] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha. Design, implementation and performance of a content-based switch. In *Proc. of IEEE INFOCOM 2000*, pages 1117–1126, Tel Aviv, Israel, Mar. 2000.

[4] ARM Ltd. *ARM7 Datasheet*, 2003.

[5] F. Baker. Requirements for IP version 4 routers. RFC 1812, Network Working Group, June 1995.

[6] D. Burger and T. Austin. The SimpleScalar tool set version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[7] J. Daemen and V. Rijmen. The block cipher Rijndael. In *Lecture Notes in Computer Science*, volume 1820, pages 288–296. Springer-Verlag, 2000.

[8] K. B. Egevang and P. Francis. The IP network address translator (NAT). RFC 1631, Network Working Group, May 1994.

[9] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138–153, June 1990.

[10] EZchip Technologies Ltd., Yokneam, Israel. *NP-1 10-Gigabit 7-Layer Network Processor*, 2002. http://www.ezchip.com/html/pr_np-1.html.

[11] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[12] Hewlett-Packard Company. *Maximizing HP StorageWorks NAS Performance and Efficiency with TCP/IP offload engine (TOE) Accelerated Adapters*, Mar. 2003. http://www.alacritech.com.

[13] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Oullette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1–2):229–248, May 1993.

[14] Intel Corp. *Intel IXP1200 Network Processor*, 2000. http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[15] Intel Corp. *Intel IXP2800 Network Processor*, 2002. http://developer.intel.com/design/network/products/npfamily/ixp2800.htm.

[16] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *Proceedings ACM/IEEE Design Automation Conference*, pages 526–529, Anaheim, CA, June 1997.

[17] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, Network Working Group, Nov. 1998.

[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[19] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, Dec. 1999.

[20] B. A. Malloy, E. L. Lloyd, and M. L. Souffa. Scheduling DAG's for asynchronous multiprocessor execution. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):498–508, May 1994.

[21] G. Memik and W. H. Mangione-Smith. NEPAL: A framework for efficiently structuring applications for network processors. In *Proc. of Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, Feb. 2003.

[22] J. C. Mogul. Simple and flexible datagram access controls for UNIX-based gateways. In *USENIX Conference Proceedings*, pages 203–221, Baltimore, MD, June 1989.

[23] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*, Nov. 2001. FIPS 197.

[24] National Laboratory for Applied Network Research - Passive Measurement and Analysis. *Passive Measurement and Analysis*, 2003. http://pma.nlanr.net/PMA/.

[25] NetBSD Project. *NetBSD release 1.3.1*. http://www.netbsd.org/.

[26] Network Processor Forum. *Benchmarking Implementation Agreements*, 2003. http://www.npforum.org/benchmarking/bia.shtml.

[27] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, June 1999.

[28] R. Ramaswamy and T. Wolf. PacketBench: A tool for workload characterization of network processing. In *Proc. of IEEE 6th Annual Workshop on Workload Characterization (WWC-6)*, Austin, TX, Oct. 2003.

[29] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A programming model for the intel IXP1200. In *Proc. of Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 100–111, Feb. 2003.

[30] K. Taura and A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Heterogeneous Computing Workshop*, pages 102–115, Cancun, Mexico, May 2000.

[31] TCPDUMP Repository. *http://www.tcpdump.org*, 2003.

[32] Y.-C. Wei and C.-K. Cheng. Ratio cut partitioning for hierarchical designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(7):911–921, July 1991.

[33] T. Wolf and M. A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Austin, TX, Apr. 2000.