Automated Task Allocation on Single Chip, Hardware Multithreaded, Multiprocessor Systems

William Plishker, Kaushik Ravindran, Niraj Shah, Kurt Keutzer
University of California, Berkeley
{plishker, kaushikr, niraj, keutzer}@eecs.berkeley.edu

Abstract

The mapping of application functionality onto multiple multithreaded processing elements of a high performance embedded system is currently a slow and arduous task for application developers. Previous attempts at automation have either ignored hardware support for multithreading and focused on scheduling, or have overlooked the architectural peculiarities of these systems. This work attempts to fill the void by formulating and solving the mapping problem for these architectures. In particular, the task allocation problem for a popular multithreaded, multiprocessor embedded system, the Intel IXP1200 network processor, is encoded into a 0-1 Integer Linear Programming problem. This method proves to be computationally efficient and produces results that are within 5% of aggregate egress bandwidths achieved by hand-tuned implementations on two representative applications: IPv4 Forwarding and Differentiated Services.

1. Introduction

For a number of years, computer architects have tried many techniques to improve the performance of application-specific programmable processors. One of the most effective is multithreading with hardware support, of which there are many flavors including fine-grain multithreading, coarse-grained multithreading [1], and simultaneous multithreading [2]. Each approach attempts to maintain high processor utilization by having hardware dynamically schedule multiple threads. Growing silicon capability has now made it possible to incorporate multiple multithreaded processors on a single die. However, since programming environments for these architectures have not advanced as rapidly, software developers face the daunting challenge of efficiently allocating the tasks of their application on to hardware multithreaded, multiprocessor architectures. While some automated solutions exist for solving this problem generically (e.g. dynamic partitioning in operating systems, offline scheduling algorithms), for high performance embedded systems the most common solution is simply a manual partitioning of the design across threads and processors. Partitioning may be done either informally, by the designer's intuition and experience, or more methodically, by architecting the design to be flexible, and then changing the task allocation based on profiling feedback. In either case it is a time-consuming and challenging problem, largely due to a huge and irregular design space further exacerbated by resource constraints. This work automates the task allocation process for hardware multithreaded, multiprocessor architectures. We use simplified models of the application and the architecture and leverage recent advances in 0-1 *integer linear programming* (ILP) to solve it efficiently. To demonstrate our approach, we map the data plane of an IPv4 router and a Differentiated Services interior node onto the Intel IXP1200, a hardware multithreaded, multiprocessor designed for network applications. For both examples, the runtime of our approach is less than one second, with resulting implementations performing within 5% aggregate data rate of hand partitioned designs.

The remainder of this paper is organized as follows: Section 2 gives some background on prior work and an overview of the IXP1200. The problem formulation is described in Section 3. In Section 4, we present the results of our approach for two network applications. Section 5 concludes and summarizes this work.

2. Background

In this section we review prior approaches to solving similar problems and also overview the Intel IXP1200 architecture, which is used as a demonstration vehicle for these ideas.

2.1. Related work

The mapping of application tasks onto an embedded multiprocessor architecture is typically conducted in two steps: task allocation and scheduling. Approximation algorithms have been extensively studied to solve these

problems for general multiprocessor models [3,4]. However, such generalized solutions are not suitable for modern embedded architectures since they fail to consider practical resource constraints. In particular, the approximation schemes do not take into account thread and storage limitations, which are critical factors that affect the quality of the mapping to multithreaded architectures. This simplification substantially limits the design space that would be explored for some embedded systems. Therefore, existing approximation algorithms are not appropriate to solve the task allocation and scheduling problems for hardware multithreaded multiprocessors.

We utilize the framework of ILP to solve our variant of the mapping problem. The use of ILP in high-level synthesis is not new. Hwang, et al. [5] presented an ILP model for resource-constrained scheduling and developed techniques to reduce the complexity of the constraint system. Extending this concept, *mixed integer linear programming* (MILP) based task allocation schemes for heterogeneous multiprocessor platforms have been advanced [6]. These formulations determine a mapping of application tasks to hardware resources that optimizes a trade-off function between execution time, processor and communication costs. The advantage of ILP is the natural flexibility to express diverse constraints and its potential to compute optimal solutions with reference to the problem model. However, ILP approaches are typically infeasible, since most ILP solvers suffer due to large run times even for simple problem instances. To counter this issue, we use a modern 0-1 ILP solver with improved search heuristics and additionally introduce special constraints to restrict the search space. Thus we exploit the flexible framework of ILP to generate optimal solutions to the mapping problem within fractions of a second.

The work that comes closest to our problem of mapping to hardware multithreaded architectures was published by Srinivasan et al. [7]. The authors consider the scheduling problem for the Intel IXP1200 and present a theoretical framework in order to provide service guarantees to applications. However, they do not consider practical resource constraints of the target architecture, nor do they test their methodology with real network applications. In contrast, our approach provides an efficient solution to the mapping problem, explicitly taking into account resource constraints of the hardware multithreaded multiprocessor.

2.2. Intel IXP1200 architecture

The IXP1200 [8] is one of Intel's first network processors based on their Internet Exchange Architecture. It has six identical RISC processors, called microengines, plus a StrongARM processor as shown in Figure 1. The StrongARM is used mostly to handle control and management plane operations. The microengines are geared for data plane processing and each has hardware support for four threads that share an instruction store. This is enabled by a hardware thread scheduler that permits fast context swapping. The memory architecture is divided into several regions: large off-chip SDRAM, faster external SRAM, internal scratchpad, and local register files for each microengine. Each region is under the direct control of the user and there is no hardware support for caching data from slower memory into smaller, faster memory (except for the small cache accessible only to the StrongARM). Currently there are several programming environments that may be used to design applications for the IXP1200 including NP-Click [9], Teja-C [10], Intel's ACE framework [11], and Intel's Microengine C [12]. We chose to use NP-Click as our implementation environment due to its modularity and ease of use. It is an efficient programming approach that provides visibility into salient architectural details that greatly affect performance.

3. Problem formulation

We approach the mapping problem in three steps: (1) construct a simplified model to capture only those salient application parameters and resource constraints that are most likely to influence the quality of the final solution, (2) encode the constraint system as a 0-1 ILP formulation, and (3) solve the optimization problem using an efficient solver to determine feasible configurations. We elaborate on these steps in the following sections.

3.1. Model

We view the multiprocessor as a symmetric shared memory architecture consisting of a memory hierarchy in which each region has uniform access time from any processing element (PE). This obviates the necessity to explicitly account for memory and communication metrics in the model. We incorporate instruction store limits per PE

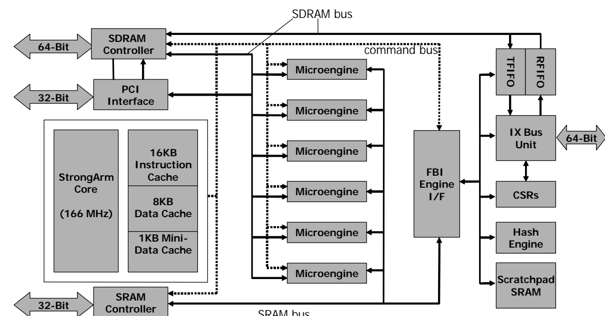

Figure 1. Intel IXP1200 microarchitecture

as a resource constraint. In our experience, applications implemented on the IXP1200 are often instruction store bound, severely complicating the load balancing process. Furthermore, the tradeoff between instruction store and execution cycles of individual tasks is critical to determining optimal performance.

In our application model, tasks are classified by *class*. Elements within a *class* are functionally the same. Tasks within a *class* can have different *implementations*, which may differ in their execution cycles, number of instructions, etc. In a multithreaded system, two tasks on the same PE may share part of the instruction store if they are of the same class and implementation. Such instructions are called *shareable instructions*. Conversely, a task that contains state may have an implementation in which instructions directly address specific state variables. Therefore these instructions may not be shared. However, since some multithreaded architectures allow for context relative addressing, they enable a logical separation of a single memory resource. As a result, instructions which utilize context relative addressing to reference state variables directly may be shared across threads, but not within one. Such instructions are called *quasi-shareable*. While direct references are faster, a designer may implement a task with state with *shareable* instead of *quasi-shareable* instructions by indirectly addressing state variables. Tasks written with shareable instructions incur additional cost of execution time and total instruction store, but allow a developer to tradeoff execution cycles and total instruction memory.

Further, we assume that the application consists of independently executing tasks. The queues in the data plane of a typical network application decouple tasks from execution dependencies. In other words, while there is a graph that represents dataflow through the application, the individual tasks may be considered as executing independently. Currently, we assume tasks have the same periodicity, but our formulation could be easily extended to accommodate multiple execution rates. The utilization of a PE by a task is measured by the number of execution cycles it consumes (execution time less long latency events). Our goal is to allocate tasks onto PEs with the objective of minimizing the average *makespan* – the maximum execution cycles of all tasks running on the system. We acknowledge that these assumptions sacrifice some accuracy, but they were carefully chosen to maximize the exploration of critical parts of the design space while still keeping the problem tractable.

3.2. Problem

We attempt to solve the following resource-constrained optimization problem: given a set of independent tasks and a set of PEs, find a feasible implementation for each task and a mapping of tasks to PEs so that the makespan is minimized. Formally, we have a collection of task classes Y , and each class $y \in Y$ consists of a set of tasks $T(y) = \{t_{y_1}, t_{y_2}, ...\}$ and a set of implementations $M(y) = \{m_{y_1}, m_{y_2}, ...\}$. An individual task in $T(y)$ can operate in any one implementation from $M(y)$. Each $m_{y_i} \in M(y)$ is characterized by a tuple, $(e_{y,m_{y_i}}, s_{y,m_{y_i}}, q_{y,m_{y_i}})$ denoting the number of execution cycles, the number of shareable instructions, and the number of quasi-shareable instructions, respectively. All tasks of the same class and implementation may share a part of the program instructions in a PE denoted by $s_{y,m_{y_i}}$. The quasi-shareable instructions denoted by $q_{y,m_{y_i}}$ can only be shared by at most N tasks of the same class and implementation, where N is the number of hardware threads in a PE. Once more than N tasks are assigned to a PE, extra instructions must be used to accommodate additional state registers. By choosing the appropriate implementation for each task, the optimization problem attempts to minimize the makespan, while satisfying constraints on the instruction store. The set P consists of the available PEs in our system. An instruction store limit S_{limit} is enforced for each PE. The parameter E_{cycle} denotes the bound on makespan.

We encode the resource constrained decision problem as a 0-1 ILP; the variables in our constraint system are the following:

(1) $x_{y,t,m,p}$: a 0-1 variable which indicates whether task $t \in T(y)$ belonging to task class $y \in Y$ with implementation $m \in M(y)$ is assigned to PE $p \in P$. In a feasible configuration, a variable with value 1 denotes a selection of implementation and PE for each task.

(2) $a_{y,m,p,k}$: a 0-1 variable which indicates whether k or more tasks from class $y \in Y$ with implementation $m \in M(y)$ are assigned to PE $p \in P$, where $k \in \{1, 1+N, 1+2N, \cdots, 1+N \cdot \lceil (|T(y)| - N)/N \rceil\}$. This counts the number of tasks from class y and implementation m in PE p in increments of N .

The constraints in our system are the following:

$$\sum_{p \in P} \sum_{m \in M(y)} x_{y,t,m,p} = 1 \quad \forall y \in Y,\ \forall t \in T(y) \tag{1}$$

$$\sum_{y \in Y} \sum_{t \in T(y)} \sum_{m \in M(y)} e_{y,m} \cdot x_{y,t,m,p} \le E_{cycle} \quad \forall p \in P \tag{2}$$

$$a_{y,m,p,k} = 1 \Leftrightarrow \sum_{t \in T(y)} x_{y,t,m,p} \geq k \quad \forall p \in P, \forall y \in Y,$$
$$\forall m \in M(y), \forall k \in \{1, 1+N, 1+2N, \cdots, 1+N \cdot \lceil (|T(y)| - N)/N \rceil\} \quad (3)$$

$$\sum_{y \in Y} \sum_{m \in M(y)} \sum_{k \in \{1,\cdots,1+N \cdot \lceil (|T(y)| - N)/N \rceil\}} q_{y,m} \cdot a_{y,m,p,k} + \sum_{y \in Y} \sum_{m \in M(y)} s_{y,m} \cdot a_{y,m,p,1} \leq S_{limit} \quad \forall p \in P \quad (4)$$

Constraint (1) is the exclusionary constraint that specifies that each task must be executed in exactly one implementation and assigned to exactly one PE. The total execution time of all tasks in their selected implementations in each PE must be less than E_{cycle} and this is ensured by constraint (2). Constraint (3) determines the values of the $a_{y,m,p,k}$ variables, which are then used in (4) to stipulate a bound on the combined instruction store of all tasks assigned to a PE, accounting separately for the shareable and quasi-shareable parts. The total number of variables in our constraint system is of order $O\left(max\{|T(y)| \mid y \in Y\} \cdot max\{|M(y)| \mid y \in Y\} \cdot |P| \cdot |Y|\right)$. The number of constraints is linear in the number of variables.

3.3. Solver

The search strategy is to perform a binary search on E_{cycle} to find the optimum possible execution time and a corresponding implementation and PE assignment for each task. We note that the decision problem formulated above is a reduction of the basic bin-packing problem and hence is NP-complete. Though encoding the problem as ILP allows us the flexibility to specify varied constraints, solving such problems in the general case is inefficient for reasonably sized instances. However, we take advantage of recent advancements in search algorithms and heuristics for solving 0-1 ILP formulations to efficiently compute solutions that are optimal with respect to our problem model. We use GALENA [13], a fast pseudo-Boolean SAT solver, to solve the constraint system.

Additionally, we can introduce specialized constraints to prune the solution space and remarkably speed up the ILP search procedure. For instance, tasks from the same task class and implementation are identical (since they are characterized by the same number of instructions and execution cycles), and hence lead to symmetric configurations. Accordingly, introducing symmetry-breaking constraints eliminates all redundant configurations that are distinguished only by a permutation of tasks from the same class. To generate these constraints, a total ordering is introduced over all tasks $(T(y), \leq)$ and implementations $(M(y), \leq)$ in each task class $y \in Y$ and PEs $(P, \leq)$ in the system. Intuitively, let $\boldsymbol{r}(t_{y_i})$ and $\boldsymbol{w}(t_{y_i})$ denote a selection of PE and implementation, respectively, for some task $t_{y_i} \in T(y)$. In order to break symmetry, we enforce the constraint that if t_{y_i} and t_{y_j} are two tasks in the same task class $y \in Y$ with $t_{y_i} \leq t_{y_j}$, then $\boldsymbol{r}(t_{y_i}) \leq \boldsymbol{r}(t_{y_j}) \wedge \boldsymbol{r}(t_{y_i}) = \boldsymbol{r}(t_{y_j}) \Rightarrow \boldsymbol{w}(t_{y_i}) \leq \boldsymbol{w}(t_{y_j})$. In this way, we avoid symmetric configurations that arise due to a permutation of tasks. This directs the ILP search towards useful configurations and achieves significant run time speedups.

Similarly, we can exploit symmetries that arise because the PEs in the system are non-distinct. We may also include restrictions on the placement or pairing of certain tasks to expedite the search. The ILP framework is sufficiently general to accommodate various other user-specified constraints based on knowledge of the problem instance. Thus, the combination of a modern 0-1 ILP solver with specialized constraints to restrict the search space provides a powerful mechanism to solve the task allocation problem efficiently.

4. Results

To demonstrate the validity of the model, we used the IXP1200 and two common and representative network applications, internet protocol packet forwarding in IPv4 and a Differentiated Services (DiffServ) interior node. IPv4 packet forwarding [14] is a common kernel of many network processor applications. The major features of this benchmark are: (1) Receiving the packet (2) checking its validity, (3) determining the egress port of the packet by a longest prefix match route table lookup, (4) decrementing the time-to-live (TTL), and finally (5) transmitting the packet on the appropriate port. We use egress aggregate bandwidth as a proxy for performance of IPv4 forwarding. The Differentiated Services architecture [15] is a method of facilitating end-to-end quality of service (QoS) over an existing IP network. In contrast to other QoS methodologies, it is a provisioned model, not a signaled one. This implies network resources are provisioned for broad categories of traffic instead of employing signaling mechanisms to temporarily reserve network resources per flow. Interior nodes apply different per hop behaviors (PHBs) to various classes of traffic. The classes of PHBs recommended by IETF include: Best Effort (no guarantees of packet loss, latency,

Table 1. IPv4 Forwarding Task Characteristics

Class	Number of Tasks	Execution Cycles	Shareable	Quasi-shareable
Receive	16	337	801	0
Transmit (Impl 1)	16	160	348	0
Transmit (Impl 2)	16	140	5	285

Table 2. DiffServ Task Characteristics

Class	Number of Tasks	Execution Cycles	Shareable Instructions
Receive	4	99	462
Lookup	4	134	218
DSBlock	4	320	1800
Transmit	4	296	985

or jitter), Assured Forwarding (4 classes of traffic, each with varying degrees of packet loss, latency and jitter), and Expedited Forwarding (low packet loss, latency and jitter).

In both applications, a developer supplied the tasks to be used, and each class and implementation was profiled in a cycle accurate simulation environment. To obtain average execution cycles per task, the application was tested with worst case input traffic. An instance of each task class and implementation was run on a PE by itself in a functionally correct configuration so that it could be profiled with the appropriate traffic. We note that different configurations may cause a task to perform differently, but those effects have not yet been substantial. We have seen at most a 10% change in execution cycles consumed between a task compiled alone and in the presence of other tasks. For shareable and quasi-shareable instructions, the application was complied with varying task configurations. We implemented a 16 port Fast Ethernet (16x100Mbps) IPv4 router consisting of 16 Receive class and 16 Transmit class tasks with characteristics shown in Table 1. A Transmit task has state and can be written with either shareable or quasi-shareable instructions to reference its shared variables. For this application, we targeted a version of the IXP1200 for which the bound on the instruction store was 1024 instructions. The result is optimal with respect to the model: the instruction store constraints preclude the two task classes from coexisting on any PE. The problem then degenerates into bin packing, the only wrinkle being that Transmit class tasks may exist in either implementation. Implementation 2 is preferred for all Transmit tasks since it is faster of the two and fits within instruction store limits. The resulting configuration shown in Table 3 is exactly the same partition that was arrived at from hand tuning.

Our DiffServ application supported 4 Fast Ethernet ports (4x100Mbps) targeting the 2K instruction store version of the IXP1200. The corresponding task configurations are presented in Table 2. Since there are only 4 tasks in each class and the IXP1200 has 4 hardware threads per PE, the quasi-shareable instructions are omitted and all instruction store used is represented by the sharable component. After testing with various mixes of traffic flows, we found the egress bandwidth of each packet class in the automatically generated design to be within 2% of the hand-tuned design, except for Best Effort which occasionally transmitted at only one-third the datarate of the hand-tuned. This is because there is a strict priority scheduler between Best Effort traffic and all other traffic, and that the latency of Transmit tasks is higher due to the simplifications inherent to the model. When the Transmit tasks service the various flows and fail to keep up with ingress, Best Effort is the first to suffer. Overall the automated partition is within 5% of the hand-tuned aggregate bandwidth for all data points, and was generated in less than a second while the hand-tuned design took days to arrive at.

The principal reason the hand-tuned design performs better than the automatically generated one is that the designer's internal model accounts for more aspects of the mapping problem than the one proposed here. Empirically, the major difference between these two models can be accounted for by the consideration of execution cycles consumed by polling. In our setup, tasks poll to determine whether a packet is ready to be processed, so there are execution cycles consumed by these polling loops. The execution cycles shown in Table 1 and Table 2 do not include this polling since it depends greatly on the final

Table 3. Final Partitioning

Application	PE1	PE2	PE3	PE4	PE5	PE6
IPv4 forwarding (hand & auto)	4 Receive	4 Receive	4 Receive	4 Receive	8 Transmit (Impl 2)	8 Transmit (Impl 2)
DiffServ (hand)	4 Receive	4 Lookup	2 DSBlock	2 DSBlock	2 Transmit	2 Transmit
DiffServ (auto)	1 Receive 1 Lookup 1 Transmit	1 Receive 1 Lookup 1 Transmit	2 DSBlock	2 DSBlock	1 Receive 1 Lookup 1 Transmit	1 Receive 1 Lookup 1 Transmit

configuration, however it can impact design decisions. Consider the DiffServ allocation problem once the 4 DSBlocks are assigned to 2 PEs by themselves (a decision that both approaches agree on). We are left with a sub-problem of allocating 4 Receive, 4 Lookup, and 4 Transmit tasks to the 4 remaining PEs. Based on the model presented, since there is ample instruction store in the sub-problem, the optimal result is to put one of each task class into each PE. But since the tasks with shorter execution times (Receive and Lookup) spend execution cycles in their polling loops, a significant number of additional execution cycles are consumed: a fact which is not considered by the model. A knowledgeable developer would note that pairing an execution-heavy Transmit task with only one other execution-heavy task (one additional polling loop) might be better than putting it with the two execution-light tasks (two additional polling loops). In this new configuration, the Transmit tasks are paired together and Receive and Lookup are given their own PEs. After trying the configuration, the results indicate that the two polling loops Receive and Lookup introduce mitigate any apparent savings from being execution-light.

5. Summary and conclusions

As the functionality expected from single chip, high performance, multiprocessor systems continues to increase, the task of distributing that functionality becomes more critical. Designers are already faced with a large and non-intuitive set of tradeoffs for task partitioning and mapping. To cope with this, we formulate the mapping problem for one instance of such resource constrained embedded systems. By encoding the mapping problem as a 0-1 ILP, we allow for flexibility and extensibility while still utilizing a high performance back end. Problems are solved in less than a second with results that are optimal with respect to the model. The model has proved itself for two representative network application producing results within 5% aggregate bandwidth of hand balanced designs. While more applications need to be tested, we feel that this approach is robust and fast enough to be used as a tool by designers when developing software for these systems. It is one piece in an overall design process that will enable designers to explore different task implementations and identify optimal mappings. This in turn expedites the overall application design flow for multiprocessor, hardware multithreaded embedded systems.

We aim to extend and improve this approach in a number of ways. First, we plan to address some limitations discussed earlier and incorporate effects like contention and excessive polling. Second, to further automate the design process, we plan to look at generating efficient task configurations from high-level application descriptions that will result in better designs. We also intend to test this approach on other more complex architectures such as Intel's IXP2400 as they become available to universities. It is our belief that as systems incur additional resource constraints, processors, and tasks, the advantages of an automated approach to the mapping problem will be more apparent.

6. References

[1] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach." 3rd Ed. Morgan Kaufmann Publishers, CA, 2003.

[2] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," Proceedings of the 22rd Annual International Symposium on Computer Architecture, pg 392-403, June 1995.

[3] C. Chekuri, "Approximation Algorithms for Scheduling Problems," Technical Report CS-TR-98-1611, Computer Science Department, Stanford University. August 1998.

[4] H. Shachnai and T. Tamir, "Polynomial time approximation schemes for class-constrained packing problems," Proceedings of Workshop on Approximation Algorithms, pg 238-249, 2000.

[5] C-T. Hwang, J-H. Lee, and Y-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume: 10, Issue: 4, April 1991, pg 464 -475.

[6] A. Bender, "MILP Based Task Mapping for Heterogeneous Multiprocessor Systems," Proceedings of EDAC, 1996, pg 283-288.

[7] A. Srinivasan, et al., "Multiprocessor Scheduling in Processor-based Router Platforms: Issues and Ideas," Network Processor Design: Issues and Practices, Volume 2, November 2003.

[8] Intel Corp., "Intel IXP1200 Network Processor," Product Datasheet, December 2001.

[9] N. Shah, W. Plishker, and K. Keutzer, "Programming Models for Network Processors," Network Processor Design: Issues and Practices, Volume 2, November 2003.

[10] Teja Technologies, Inc., "Teja C: A C-based Programming language for Multiprocessor Architectures," to appear in Network Processor Design: Issues and Practices, Volume 2, November 2003

[11] Intel Corp., "Intel IXA SDK ACE Programming Framework Developer's Guide," June 2001.

[12] Intel Corp., "Intel Microengine C Compiler Language Support Reference Manual," March 2002.

[13] D. Chai and A. Kuehlmann: "A fast pseudo-boolean constraint solver." DAC 2003: 830-835

[14] F. Baker, "Requirements for IP Version 4 Routers," Request for Comments - 1812, Network Working Group, June 1995

[15] S. Blake, et al., "An Architecture for Differentiated Services," Request for Comments - 2475, Internet Engineering Task Force (IETF), December 1998.