University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Masters Theses 1911 - February 2014

2009

Task Partitioning and Mapping Algorithms for Multi-core Packet Processing Systems

Wei Chen
University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/theses

Chen, Wei, "Task Partitioning and Mapping Algorithms for Multi-core Packet Processing Systems" (2009). *Masters Theses 1911 - February 2014*. 255.
Retrieved from https://scholarworks.umass.edu/theses/255

This thesis is brought to you for free and open access by ScholarWorks@UMass Amherst. It has been accepted for inclusion in Masters Theses 1911 - February 2014 by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

TASK PARTITIONING AND MAPPING ALGORITHMS FOR MULTI-CORE PACKET PROCESSING SYSTEMS

A Thesis Presented

by

WEI CHEN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 2009

Electrical and Computer Engineering

TASK PARTITIONING AND MAPPING ALGORITHMS FOR MULTI-CORE PACKET PROCESSING SYSTEMS

A Thesis Presented

by

WEI CHEN

Approved as to style and content by:

____

Tilman Wolf, Chair

____

Wayne Burleson, Member

____

Russell Tessier, Member

____

C.V.Hollot, Department Head
Electrical and Computer Engineering

TABLE OF CONTENTS

Page

LIST OF FIGURES . . . . **v**

CHAPTER

1. INTRODUCTION **1**

2. RELATED WORK **4**

2.1 Classes of Parallel Computers 4

2.1.1 Cluster 5
2.1.2 Multiprocessor 5
2.1.3 Multicore Computing 6
2.1.4 Network Processor 6

2.2 Task Partitioning Algorithms 7

2.2.1 Analysis Scheme 8
2.2.2 Construction Scheme 9

2.3 Task Mapping Algorithms 10

2.3.1 Graph Theoretic Algorithms 10
2.3.2 Mathematical Programming 10
2.3.3 Heuristic Algorithms 11

3. SYSTEM CONFIGURATION PROCESS **12**

3.1 Application Partitioning 12
3.2 Task Mapping 13

3.2.1 Task Mapping Problem Statement 14
3.2.2 Runtime Profiling 15

3.3 Dynamic Adaptation 16

4. APPLICATION MAPPING **18**

4.1 Task Duplication 18
4.2 UDFS Mapping Algorithm 21
4.3 KL algorithm 22
4.4 Extended KL Algorithm 23
4.5 Simulated Annealing Algorithm 25
4.6 Merging and Duplication 26

5. EVALUATION OF ALGORITHMS **30**

5.1 Simulation Environment 30
5.2 Profiling 32
5.3 Duplication 36
5.4 Mapping:UDFS 37
5.5 Mapping: KL Algorithm 40
5.6 Mapping: Extended KL 40
5.7 Mapping: Simulated Annealing Algorithm 42
5.8 Merging and Duplication 45
5.9 Architecture Exploration 51

6. IMPLEMENTATION CONSIDERATIONS ON INTEL IXP SYSTEM **56**

6.1 System Architecture 56
6.2 Model Implementation 56

6.2.1 Processing Units 57
6.2.2 Inter-processor Communication 58

6.3 Applicability and Limitation of Task Mapping Model 59

7. CONCLUSIONS **61**

BIBLIOGRAPHY **62**

LIST OF FIGURES

Figure **Page**

2.1 Generic Network Processor Architecture 7

2.2 Parallel program development flow 8

3.1 Application Graph 12

3.2 Task graph 14

4.1 Task Duplication Example 19

4.2 Workload distribution comparison 20

4.3 Situations where Nodes A and B can not be merged 28

4.4 Situations where Nodes A and B can be merged 28

4.5 Comparison of duplicate and merge-then-duplicate schemes 29

5.1 Overall Simulation Flow 31

5.2 Experimental application. 33

5.3 Workload for Trace 1. 34

5.4 Workload for Trace 2. 34

5.5 Utilization of tasks for Trace 1. 35

5.6 Utilization of tasks for Trace 2. 35

5.7 Distribution of Work w_i' per Task Instance Before and After Duplication for Trace 1. 36

5.8 Distribution of Work w_i' per Task Instance Before and After Duplication for Trace 2. 37

5.9 Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different Mapping Algorithms. . . . 39

5.10 Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different Mapping Algorithms. . . . 41

5.11 Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different α. . . . 43

5.12 Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different L. . . . 44

5.13 Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different Mapping Algorithms. . . . 46

5.14 Distribution of Work w_i' per Task Instance with and without merging for Trace 1. . . . 47

5.15 Distribution of Work w_i' per Task Instance with and without merging for Trace 2. . . . 47

5.16 Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$. . . . 48

5.17 Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$. . . . 49

5.18 Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$. . . . 50

5.19 Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for All Mapping Algorithms . . . 51

5.20 Architecture Exploration: Catalog 1 . . . 53

5.21 Architecture Exploration: Catalog 2 . . . 54

5.22 Architecture Exploration: Catalog 3 . . . 55

6.1 IXP 2400 network processor data path architecture . . . 57

CHAPTER 1

INTRODUCTION

Routers are the devices that connect scattered networks and create a unified Internet which keeps changing every aspect of our lives. Since 1970's, the Internet has never stopped evolving and there is no sign that it will slow down. Therefore, routers, which serve as the key devices of Internet technology, also need to keep pace with the development of the Internet. Although routers were originally designed as simple store-and-forward "dumb" devices, nowadays, researchers and designers are trying to put more intelligence into them to meet the requirements of high performance, security, and flexibility etc. [13] Applications such as firewall, NAT, encrytion/decryption for VPN are integrated into these devices. With the evolution of the Internet, more applications, protocols and services are expected to push the network into one that will require routers to be stronger and more flexible.

Router applications could be implemented either in software or in hardware. In most cases, hardware designs are faster but require longer design cycles. While for software implementations, it is much easier for developers to build and debug their codes, but the performance of their application will be limited by the ability of target hardware platform. Therefore, router designers will always have to find a balance between performance and development time. To meet the need for rapid high performance network application development, packet processing engines are widely accepted nowadays. In a typical packet processing system such as Intel IXP2400, there is a general processor which serves as a control plane, and tens of simpler packet processing units which in general have limited instruction set but are optimized for network packets processing applications. These packet

processing processors form the data plane in a router, and are able to download new applications at any time. The packet processing engine architecture also reflects technology progress in multi-core architecture. Unlike ASICs, where everything is hard-wired into the chip, these multi-core, programmable processors are more flexible. Every time network designers have new ideas about applications, protocols or algorithms, they can program the device and get the router running. In the ASIC world, this is not the case. ASIC designers normally need much longer develop time to implement the network designer's idea on the chip. What is worse, as the Internet technology changes so fast, when the ASIC is done, probably the originally "new" idea is already outdated. In addition, these multi-core, programmable processors have far more processing power than general purpose processors. This processing power comes from the fact that network application has inherent parallelism and different packets can be processed by multiple cores at the same time. This greatly improves the throughput of the packet processing system. We can see that specially design network processors are more suitable for network application than a general purpose processor while more flexible than ASIC chips [18].

While multi-core, programmable systems are good candidates for network applications, there is an important problem not solved yet. It is not easy to program these systems to make full use of their processing power. As these systems have multiple cores and diverse shared resources, the problem of how to balance workload among multiple cores comes when we try to assign tasks to cores. Since the workload of system is determined not only by applications that run on processing cores, but also by the content of network traffic. It is not obvious how to reasonably assign tasks to each core so that there will not be any bottleneck that might compromise the performance of the system. In this thesis, I am trying to find a method that can effectively program the multi-core devices to unleash their processing power.

My method involves the following steps.

- Application partitioning. This step involves partitioning the network application graph into more detailed task graph. Nodes in task graph are the basic computation steps in the network application. This step is necessary because we need to divide the application into pieces so that each one of them can be processed by one of the processors.

- Task mapping. Task graph is annotated by profiling information. The annotated task graph is then mapped to the multi-core, packet processing systems using mapping algorithms. This step is crucial because it determines how effectively those computing resources can be utilized and how much contention occurs on shared resources.

- Dynamic Adaptation. This step involves the dynamic changing of the mapping after some time interval. This step is important because network is a dynamic system. A single fixed mapping will not create an efficient network processing system, so we need to monitor the online traffic and adjust the mapping accordingly.

CHAPTER 2

RELATED WORK

Parallel Computing is a form of computation in which many instructions are carried out simultaneously [15], operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently. There are several different forms of parallel computing: bit-level parallelism [12], instruction-level parallelism [21], data parallelism [16] and task parallelism [19]. Bit-level parallelism is achieved by increasing the word size of the computer. Instruction-level parallelism can be done in many ways, such as by reordering of the program so that a program can be combined into different groups that can be executed in parallel without changing the result. Data parallelism means distributing data across different computing nodes to be processed in parallel. Task parallelism targets the program that entirely different calculations can be performed on either the same or different sets of data. Parallel computers can be roughly classified according to the level at which the hardware supports parallelism - multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters and grids use multiple computers to work on the same task.The following section discusses the classes of parallel computers. Then we present the previous work on the utilization of the parallel computers including task partitioning algorithms and task mapping algorithms.

2.1 Classes of Parallel Computers

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. The classification reflects the difference between computing nodes, the memory organization and the connecting medium.

2.1.1 Cluster

A cluster [11] is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer.Clusters are composed of multiple stand-alone machines connected by a network such as fast local area network. So each computing node is a single computer. It is not necessary that computers in the cluster be symmetric, load balancing will be easier to achieve if they are. The most common type of cluster is the Beowulf cluster [20], which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network. Beowulf technology was originally developed by Thomas Sterling and Donald Becker. The vast majority of the TOP500 supercomputers are clusters.

It is worth to mention grid computing, a special type of cluster computing system [7]. Grid computing makes use of computers communicating over the Internet to work on a given problem. Because the grid computing nodes communicate through Internet, the cost of communication is relatively high. So it is optimized for workloads which consist of many independent jobs or packets of work, which do not have to share data between the jobs during the computation process. Grids serve to manage the allocation of jobs to computers which will perform the work independently of the rest of the grid cluster. Resources such as storage may be shared by all the nodes, but intermediate results of one job do not affect other jobs in progress on other nodes of the grid.

2.1.2 Multiprocessor

A multiprocessor system has multiple processors on the same motherboard. These processors can be symmetric (SMP) [22] or asymmetric(ASMP) [1]. The most common type of multiprocessor is symmetric multiprocessors. A symmetric multiprocessor is a computer system with multiple identical processors that share memory and connect via a bus. These systems allow any processor to work on any task no matter where the data for that task are located in memory. Bus contention prevents bus architectures from scaling. As a

result, this kind of systems generally does not comprise more than 32 processors.Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists. An asymmetric multiprocessor is comprised of multiple unique processors, normally with a master processor and multiple slave processors that are designed for specific tasks. Examples of asymmetric multiprocessing include many media processor chips that are a relatively slow base processor assisted by a number of hardware accelerator cores.

2.1.3 Multicore Computing

A multicore processor is a processor that includes multiple execution units on the same chip [3]. A multicore processor can issue multiple instructions per cycle from multiple instruction streams. Cores in a multicore device may share a single coherent cache at the highest on-device cache level (e.g. L2 for the Intel Core 2) or may have separate caches (e.g. current AMD dual-core processors).The processors also share the same interconnect to the rest of the system. The proximity of multiple CPU cores on the same die allows the cache coherency circuitry to operate at a much higher clock rate than is possible if the signals have to travel off-chip. Multi-core systems are very popular nowadays, the representative systems include Core, Core 2 and Xeon from Intel etc.

2.1.4 Network Processor

A network processor is an integrated circuit which has a feature set specifically designed for the networking application domain [24]. The generic network processor has the architecture shown in figure 2.1. We can see that a network processor normally includes multiple RISC cores. It also has dedicated hardware for common networking operations, high-speed memory interfaces, high-speed IO interfaces,interface to general purpose CPU etc. Network processor designers from different companies have made vastly different decisions about I/O interfaces, memory interfaces, and programming models, system archi-

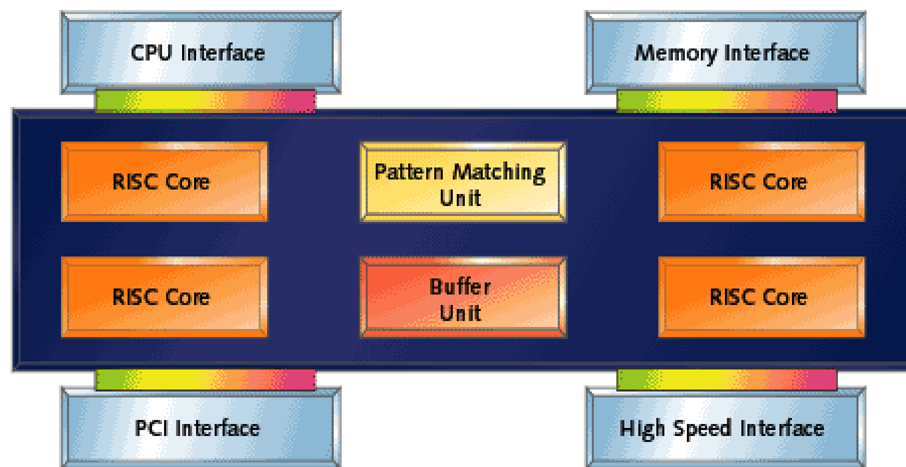


Figure 2.1. Generic Network Processor Architecture

tecture and the type of hardware acceleration to include. The examples of existing network processors include C-5 digital communication processor [17], Intel IXP2400 [18], Lucent network processor, Sitera network processor etc [23].

2.2 Task Partitioning Algorithms

Parallel program development includes four stags as shown in figure 2.2. The partitioning stage of a design is intended to expose opportunities for parallel execution. Hence, the focus is on defining a large number of small tasks in order to yield what is termed a fine-grained decomposition of a problem.The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks so as to allow computation to proceed. This information flow is specified in the communication phase of a design.In the third stage, we move from the abstract toward the concrete. We revisit decisions made in the partitioning and communication phases with a view to obtaining an algorithm that will execute efficiently on some class of parallel computer. In particular, we consider whether it is useful to combine, or agglomerate, tasks identified by the partitioning phase, so as to provide a smaller number

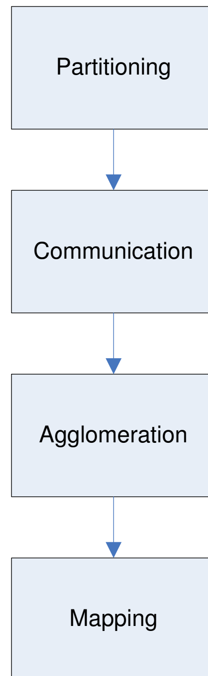


Figure 2.2. Parallel program development flow

of tasks, each of greater size. We also determine whether it is worthwhile to replicate data and/or computation.In the fourth and final stage of the parallel algorithm design process, we specify where each task is to execute [6].

In this section, we discuss the related work done in task partitioning area. Task partitioning is a crucial step for parallel computing application. If task is well partitioned and the dependency among modules is minimized then the parallel computing system is more possible to be fully utilized.

2.2.1 Analysis Scheme

The philosophy of analysis scheme is to take a program designed by an application designer in a traditional programming language such as C, C++ etc, analyze the program and partition it into multiple independent tasks. This one is in fact an ideal method for task partition since we can still follow our familiar sequential programming style and at the same time enjoy the power of parallel computing. The basic idea of this scheme is to extract the program dependency graph and partition this graph. There are several proposed

methods in this scheme. These methods are distinguished in the granularity at which the program is partitioned. In a coarse level, the original program is investigated for parts of program that can be executed in parallel by inserting queues [30]. In a refined level, the original program is compiled and the asm code is investigated and reordered then grouped together to achieve parallelism [29]. The granularity can also be adapted as needed as in [35]. Analysis scheme hides the parallel architectures from software designer, which can facilitate the development of the software. But this scheme has a big limitation since it is not obvious how to partition the program into parts that will not have communication and synchronization issues such data dependency, data consistency etc.

2.2.2 Construction Scheme

The construction scheme tackles the parallel problem in a different way. Instead of trying to partition the program, this scheme creates new programming models at the very beginning, the program is designed with parallelism in mind. These parallel programming languages make assumptions about the underlying memory architecture - shared memory, distributed memory, or shared distributed memory. Shared memory programming languages communicate by manipulating shared memory variables. Distributed memory uses message passing. POSIX Threads [9] and OpenMP [4] are two of most widely used shared memory APIs, whereas Message Passing Interface (MPI) [2]is the most widely used message-passing system API. One concept used in programming parallel programs is the future concept, where one part of a program promises to deliver a required datum to another part of a program at some future time. In network application area, Click module router toolkit [14] also follows this scheme. The building blocks are basic network processing steps. The network application is built by connecting these elements together.

2.3 Task Mapping Algorithms

Task mapping is another very crucial step in parallel computing paradigm. It directly affects the performance of the parallel computing system. The task mapping algorithms can be roughly classified as: graph theoretic algorithms, mathematical programming and heuristic algorithms. In this section we review these algorithms.

2.3.1 Graph Theoretic Algorithms

Graph theoretic algorithms are very popular algorithms for task mapping because task partitioning process can normally generate dependency graph of the program which fit right into the graph theoretic algorithms. The input to graph theoretic algorithms is a graph of partitioned tasks annotated by task execution time, communication cost or some other parameters. The graph theoretic algorithms are used to partition the annotated graph into multiple subgraphs and assign each one of them to the appropriate execution cores. Examples of graph theoretic algorithms for task mapping include network flow algorithm in [32] which uses Max Flow/Min Cut algorithm to find assignments which minimize total execution and communication costs, shortest tree algorithm in [8] which describes a shortest tree algorithm that minimize the sum of execution and communication costs for arbitrarily connected distributed systems with arbitrary number of processors provided the interconnection pattern of the modules forms a tree, A* algorithm [10] which describes a graph matching approach that match task graph with distributed system to achieve optimal task assignment.

2.3.2 Mathematical Programming

Mathematical programming [36] [34] [27] approaches the task mapping problem in another way. This method considers the resource constraints of the multiprocessor systems such as computation resource constraint, memory capacity constraint, communication constraint. The constraints are represented by mathematical inequalities and mathematical pro-

gramming is formulated. The different constraints such as computation, communication, memory lead to many different versions of mathematical programming.

2.3.3 Heuristic Algorithms

Since task mapping problem is NP problem, heuristic algorithms are often developed to tackle such problem. These algorithms include well-known simulated annealing (SA) algorithm [31] which recursively searches the mapping spaces and stop when the criterion is met, genetic algorithms [5] which simulate the evolution process and let the mapping evolves itself until a good mapping is obtained. Other customized heuristic algorithms include modified flow algorithm in [25] which augments the flow algorithm with additional parameters and objectives to achieve a better balance workload. Also some algorithms do some extra operations such as duplication to the task graph to achieve a better balance [26]. The decision of which tasks should be duplicated is derived from the profiling information of the task graph. Heuristic algorithms are where innovations can be made as long as we can pinpoint the key property of the problem.

CHAPTER 3

SYSTEM CONFIGURATION PROCESS

As we mentioned above, router design with multi-core, programmable device involves three steps as (1) Application partitioning (2) Task mapping and (3) Dynamic Adaptation. Here in this chapter, we are going to describe the design of each of these steps.

3.1 Application Partitioning

When we start our router design process, the first thing we can do is to write the source code for the application. Then we can construct an application graph according to the source code. In the application graph, we have nodes representing the processing steps we need in our packet processing system and the connections between nodes that specify the sequences of the processing steps. An application graph has the form as Figure 3.1. Each block node represents the processing step and a directed edge indicates that there may exist some packets that require a processing step from where the edge originates followed by the step to which the edge points.

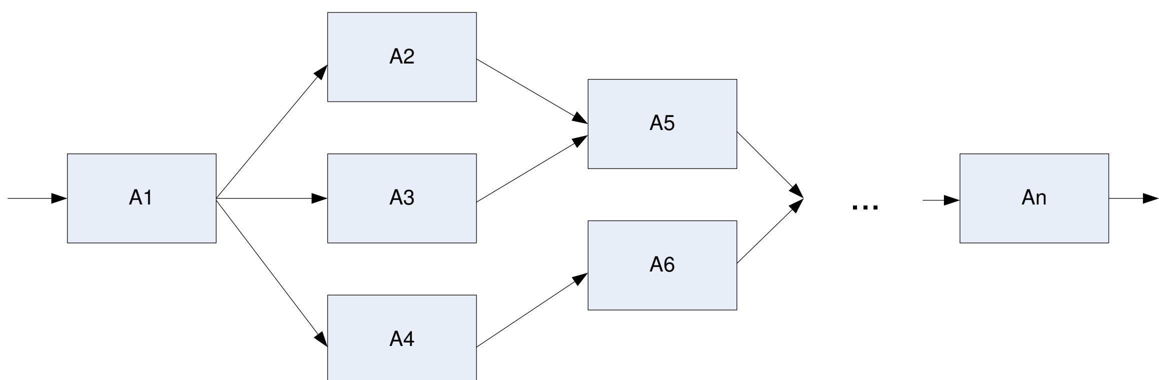


Figure 3.1. Application Graph

After an application graph has been constructed, we need to partition the applications to smaller pieces so that they can be processed by processing units in the processors. We call these smaller pieces "tasks", which are also the basic mapping units in our work. One important issue here is how to determine the granularity of the tasks. In the finest level, where each task represents each instruction, the outcome will be an incredibly large task graph and intractable number of tasks. In an ideal world, where we can have a super intelligent computer, this partition scheme maybe perfect because we can explore every possible parallelism in our application. But the reality is far from perfect, neither our computer is able to store such a large amount of information nor can it process them fast enough. In the other end, if the task is too coarse, then we will lose a lot of valuable information in the application. We will not be able to utilize the parallelism inside the application enough which will lead to a low-performance packet-processing system. So we should carefully set our task granularity to find a balance point between these two extremes.

In our work, we define the tasks by examining the source code and identifying major functions. So semantically, these tasks represent fundamental processing operations that occur in the context of packet processing (e.g. protocol header extraction, loop within router lookup algorithm, checksum computation etc.). If we use Click modules to design the router, then the Click modular router configuration is already a task graph itself. The partitioning result is illustrated by Figure 3.2.

3.2 Task Mapping

From application partitioning, we have constructed a task graph of our packet processing system. Our mission now is to assign each of these tasks to processing units. This process is the most critical step in our work because it directly determines the performance of our system. Generally speaking, this step is a graph partitioning problem which can be formalized as below.

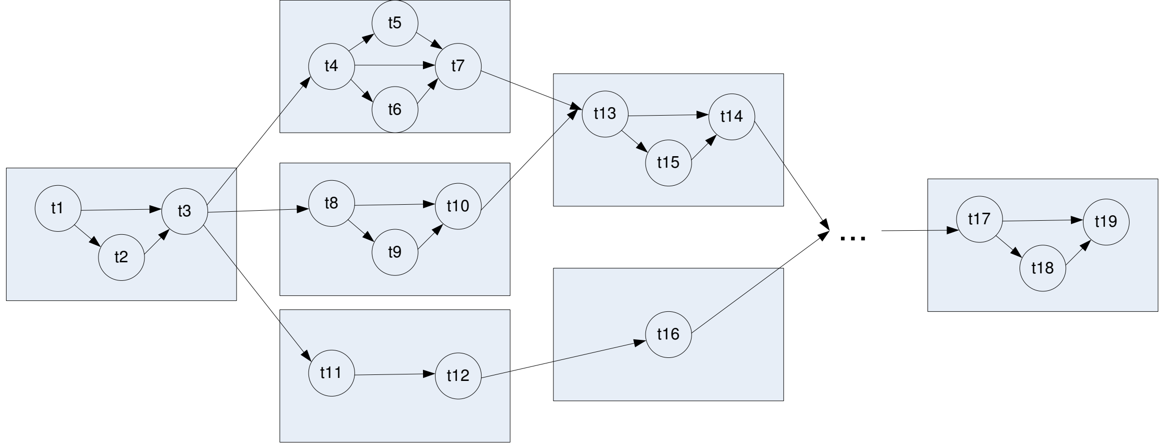


Figure 3.2. Task graph

3.2.1 Task Mapping Problem Statement

From application partitioning, we have a task graph with T nodes $t_1, t_2, ..., t_T$ and directed edges $e_{i,j}$ that represent processing dependencies between task t_i and t_j. As for our target processing system, we assume that there are N processors with M processing units on each one of them. Each unit can process one task at a time so the system can process $N \times M$ tasks concurrently. We also assume that processor interconnect provides connectivity from any processor to any other processor. The objective of our mission is to partition the T-node graph into N pieces with each piece has no more than M nodes. In mathematical format, we are going to find a mapping m that puts each of T tasks to one of N processors: $m : t_1, ..., t_T \longrightarrow [1, N]$. The mapping has to meet the constraint of resource limitations: $\forall j, 1 \leq j \leq N : |t_i|m(t_i) = j| \leq M$.

The objective is to find a mapping that can maximize the throughput or a mapping that can provide the most balanced processor utilization. The two goals are equivalent because such a mapping can provide the highest overall throughput without overloading any particular processor.

We need to mention here that tasks can be assigned to one or multiple processors or even not be assigned at all. The reason is that different tasks may have different computation

requirements in different situations. We will further discuss this issue when we talk about task duplication in the later section.

Now we have formalized the problem. But the task graph from application partitioning only gives us the functionality and connectivity of nodes. In order to maximize the throughput of our system, we need to get the information of the workload of each node so that we can do the mapping. So before we can actually do the mapping, we have an initialization step in which we collect the workload profile of the application.

3.2.2 Runtime Profiling

There are two profiling schemes called static and dynamic. In the static profiling scheme, workload information is collected off-line and used to do the mapping for the system while dynamic profiling collects the workload information when the system is operating. We know that the workload of the packet processing system is affected by two factors. One is the computation characteristics of the tasks in the systems. The other is the network traffic that exercises the processing system. While the computational characteristics of the tasks are fixed in a particular system, the network traffic is changing every minute during operation. Especially when more and more services are added to packet processing systems, the number of traffic types will increase quickly and processing requirement will become more data-dependant. So to accurately characterize the workload information, we need to use runtime profiling scheme.

During runtime, we collect the following information:

1. Task service time s_i: For each task t_i, we determine the service time s_i measured in number of instructions executed per packet. Since this value may be different for each packet, we consider s_i as the expected value from a random variable S_i.

2. Edge utilization $u(e_{i,j})$: At the completion of each processing tasks, we observe where the packet is processed next. This transition from task t_i to task t_j is denoted as utilization $u(e_{i,j})$ of edge $e_{i,j}$.

3. Task utilization $u(t_i)$: Based on edge utilization, we can derive the utilization of a particular task t_i which is denoted by $u(t_i)$.

Based on these values, we can derive the values to annotate the task graph.

After constructing this annotated workload graph, we can develop our task mapping algorithm to do the actual mapping. Algorithms are described in the following two chapters.

3.3 Dynamic Adaptation

In order to make the packet processing system always run in an optimal setting, we need to dynamically change the mapping. We call this process dynamic adaptation. The step is important because the processing workload required by network traffic cannot be known in advance because end-systems may send packets to any arbitrary destination using any protocol in any time. Generally, we have two ways to tackle this problem. One is to over-provision for any possible traffic scenario. Using this measure, we need to predict all the possible traffic and set our parameters to meet the worse case scenario. The measure in one hand can not produce best performance since it pessimistically estimates the situation, in the other hand, it is getting harder to predict the traffic before hand since more and more services are added on the packet processing systems. Because of these shortcomings, we decided to take the second method, which is to dynamically adapt our system.

To collect the real time traffic information, we need to monitor the dynamic trends of the processing workload. We collect the run-time utilization parameter $u(t_i)$ and $u(e_{i,j})$. These values can be directly used in the next mapping process. The next problem we need to answer is how frequently we should do the re-mapping. If we do the re-mapping in too short a time period, then the re-mapping cost will be too high and affect the overall system performance. Also it can generate re-mapping that is affected by traffic bursts that are not representative of the overall workload. But if we do the re-mapping in an extended period of time, the system may also suffer from inferior performance because of unsuitable mapping. So we need to carefully find a mapping interval that can balance these two

situations. Generally, the interval should depend on the workload change patterns. If the workload changes a lot, that means that we need to do the mapping again. In our current work, we use a fixed mapping interval according to our experience.

CHAPTER 4

APPLICATION MAPPING

From last chapter, we know that the system configuration involves three steps as (1) application partitioning (2) task mapping (3) runtime adaptation. We also had formalized the task mapping problem. In this chapter, we describe algorithms for task mapping.

4.1 Task Duplication

From runtime profiling step, we have collected three runtime parameters of the system. Using these parameters we can find the values that can closely represent the real workload that a task can place on processing resources. In order to model how computationally demanding a task is we need to consider both its expected computation time and its frequency of being used. So we assign to each task node t_i the weight:

$$w_i = u(t_i) \cdot E[S_i]. \tag{4.1}$$

During our runtime-profiling phase, we found that $u(t_i)$ for some tasks can be very large in some periods of time which lead to a large w_i value. This phenomenon is not good for our mapping phase because task computation complexities are so different from each other. So we need to try to evenly distribute the values of w_i. The measure we take is to duplicate those heavy-duty tasks. That is to create additional instances for those heavy-duty tasks. These duplicated instances are fully connected to the same predecessor and successor tasks as the original task. We assume that the predecessor distributes packets uniformly among all task instances and thus effectively reduces the utilization of each task instance. This procedure can be illustrated by Figure 4.1.

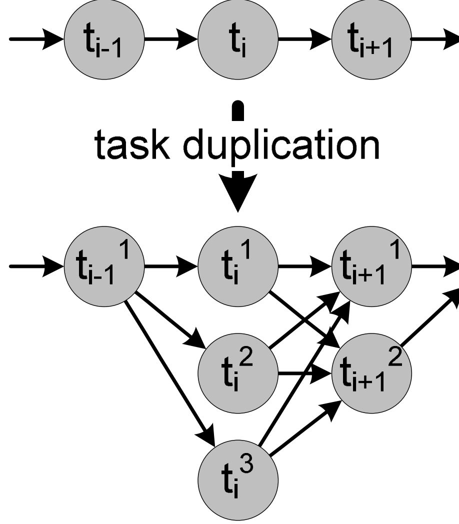


Figure 4.1. Task Duplication Example

Now here comes the problem of how to duplicate the task. We need to determine which task to duplicate and how many instances should be created. The intuition is that we should duplicate the most heavy-duty ones first and balance the amount of work that each task performs in order to simplify the mapping process.At the same time, we also need to meet the constraint that the total number of task instances is not more than the number of computing units. So we can use a greedy scheme to generate our duplication and continuously check the constraints. To better describe our duplication scheme, we use some new notations. We use parameter d_i to indicate the number of duplicated instances for task t_i. These instances are named $t_i^1, ..., t_i^{d_i}$. Any incoming edge $e_{j,i}$ from task t_j to t_i is duplicated: $e_{j,i^1}, ..., e_{j,i^{d_i}}$. Outgoing edges are also duplicated in the similar way. Due to the reduced edge utilization of $u(e_{j,i})/d_i$, fewer packets are processed by each task instance and the task utilization decreases to $u(t_i)/d_i$. So the amount of work required by each task instance is denoted as w_i':

$$w_i' = \frac{u(t_i)}{d_i} \cdot E[S_i]. \tag{4.2}$$

The algorithm is described by Algorithm 1. Here $argmax_i(w_i')$ is a function that returns the ID of node with maximum workload.

Algorithm 1 Task Duplication Algorithm.

1: **while** $\sum_{i=1}^{T} d_i < N \cdot M$ **do**
2: $\quad j \leftarrow \text{argmax}_i \, w'_i$
3: $\quad d_j \leftarrow d_j + 1$
4: **end while**

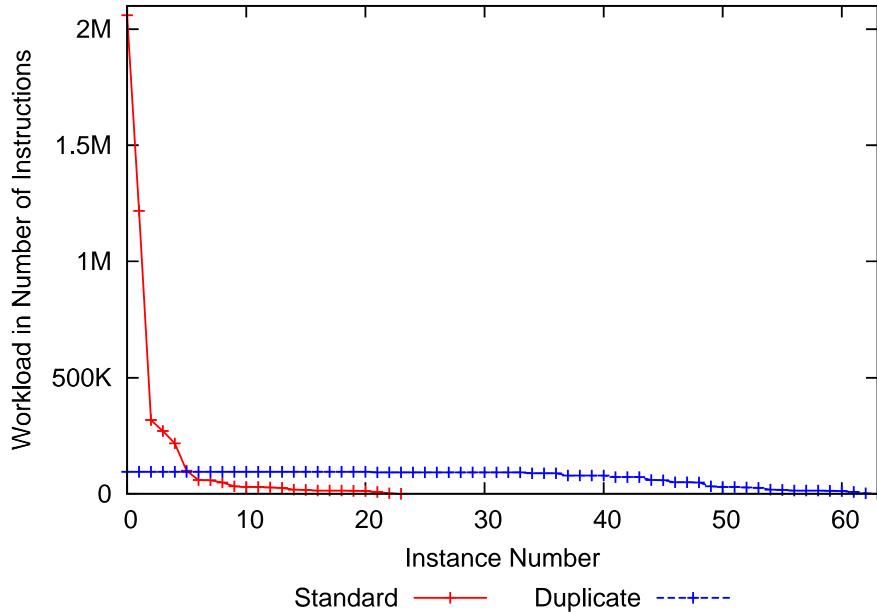


Figure 4.2. Workload distribution comparison

This algorithm can produce a more balanced workload. This is verified by our experimental results. One impressive result is shown by Figure 4.2.From the figure, we can see that before duplication, only 25 task instances exist and processing requirements differ by several orders of magnitude. After duplication, we have 64 task instances (since $N \cdot M = 64$ in our experimental setup) with much more balanced workload. This result also illustrates how difficult it would be to find a balanced mapping when using tasks without duplication. A single task with large processing requirements would represent a bottleneck in the packet processing system.

4.2 UDFS Mapping Algorithm

A mapping problem has been formalized in the last chapter. The objectives of mapping are to evenly distribute the tasks to processing units and minimize the communication between different processors. In our duplication phase, we have sliced the big tasks into smaller ones to effectively generate tasks with similar workload requirement. So when we do the mapping, the workload of each processor can be evenly distributed considering that each processor has the same number of processing units and each one of these processing units can process one task at a time. As for minimizing the communication between processors, this is crucial because communication resource is very limited in current multi-core, programmable system. Over frequent communication between processors requires more storage resource and communication links and will also cause long queue etc. which greatly depreciate the overall system performance. So we should try to keep adjacent tasks in the same core, in this way, when a task passes packets to another task, the state can efficiently be transferred through local registers. So our guideline is that we should try to cluster adjacent tasks together and we should also put those high utilization edges in one core instead of letting them cross two cores. Following this guideline, we can construct our intuitive algorithm which we called utilization-based depth-first(UDFS) algorithm. The algorithm greedily clusters tasks on a processor until all processing units are fully utilized. The key aspect of the algorithm is the order in which the task graph is traversed. High-utilization edges are traversed first to increase task locality and reduce interconnect usage. The algorithm is shown in Algorithm 2.

A more detailed description of this algorithm is as follows: We initially map node t_1, which is assumed to be the ingress node for all traffic, to the first processor. Then, using the *map_next* function, we search among all outgoing edges to find that with the highest utilization. If there are still resources available on the same processor, the task that is pointed to by this edge is mapped to the same processor. Otherwise it is mapped to the next processor. This process is repeated recursively to achieve depth-first mapping. The

Algorithm 2 UDFS Task Mapping Algorithm.

```
1:  function map_next(i,p)
2:  while ∃ e_{i,j} with t_j unmapped do
3:      k ← argmax_j(u(e_{i,j})) //find the node connected by the heaviest utilized edge
4:      if tasks_allocated_to(p) ≤ M then
5:          //if there are still available processing units on core p.
6:          m(t_k) ← p //assign t_k to core p
7:          p ← map_next(k,p) //map the next node
8:      else
9:          //if there are not available processing units left on core p
10:         m(t_k) ← p + 1 //assign t_k to core p + 1
11:         p ← map_next(k,p + 1) //map the next node
12:     end if
13: end while
14: return p
15:
16: function map()
17: m(t_1) ← 1
18: map_next(1,1)
19: return m
```

recursion terminates when a node has no outgoing edges to unmapped tasks.The variable p keeps track of which processor is currently being used for task allocation.

We should also note that (1) The algorithm maps tasks and their duplicates. To simplify notation, only tasks are mentioned. (2) If the ingress task is different from t_1, the algorithm can be easily adapted. (3) We assume that a packet transfer between processors is the basic unit of interconnect usage. In some cases, it may be possible that the interconnect usage is variable. This can occur when different amounts of processing state needs to be sent between processors. In such a scenario, the algorithm would use a different function inside the argmax function.

4.3 KL algorithm

UDFS is simple and intuitive. It can produce decent mappings for the packet processing system but it, by no means, is the best mapping algorithm. We learn that task mapping is essentially a graph partitioning process. So we can explore in the well researched graph

partitioning field and pick some existing algorithms and extend them for our purposes. One of the good candidates is Kernighan/Lin Algorithm (KL) algorithm.KL algorithm is an iterative graph partitioning algorithm. Given a graph $G = (N, E, W_E)$ with nodes and weighted edges and an initial partitioning of the graph that $G = G_1 + G_2$ and $|G_1| = |G_2|$. Here $|G|$ is the number of nodes in G. Now let $C = cost(G_1, G_2) = \sum W_E \forall E(G_1, G_2)$, that is the cost of the partitioning is equal to the weights of all the edges that cross the partitioning. The goal is to minimize C for a given G. To do that, let X be a subset of nodes of G_1 and Y be a subset of nodes in G_2, such that $|X| = |Y|$. If we were to switch X and Y, we would not change the number of nodes in each of the two subgraphs. However we could then calculate a new cost of partitioning with $(G_1 - X) \bigcup Y$ and $(G_2 - Y) \bigcup X$; if the cost of the new subgraphs is less than the cost of the old subgraphs, then we should accept the new subgraphs in place of the old subgraphs. The trick of KL algorithm is efficiently finding subsets of nodes X and Y to swap. Let $Ex(n)$ equal the external cost of leaving node n in subgraph G_1 (i.e. $\sum W_E \forall E(n, G_2)$) and $In(n)$ equal the internal savings of leaving node n in subgraph G_1 (i.e. $\sum W_E \forall E(n, G_1)$). The value of switching node n into subgraph G_2 is $D(n) = Ex(n) - In(n)$. $D(n)$ can be similarly calculated for all nodes in G_2. With these $D(n)$ values assigned to each node, the comparison of two subgroups becomes simple. The value of switching two nodes X and Y between G_1 and G_2 is: $gain(X, Y) = D(X) + D(Y) - 2 * W_E(X, Y)$. Note that since X and Y remain in different subgroups, the benefit of removing $W_E(X, Y)$ disappears for switching. The Kernighan/Lin Algorithm thus steps through the problem of improving a partitioning as described by Algorithm 3.

4.4 Extended KL Algorithm

From the description of KL algorithm in the last section, we can see that KL algorithm has limitations when applied to our mapping problem. KL algorithm requires that all the nodes have the same amount of weight and the initial partitions have the same number of nodes in each partition. These two conditions are not satisfied in our problem. Our

Algorithm 3 KL Task Mapping Algorithm.

1: **repeat**
2: Compute $D(n)$ for all nodes n in graph.
3: Unmark all nodes in the graph.
4: **while** Unmarked nodes exist **do**
5: Find two unmarked nodes X and Y that maximizes $gain(X, Y)$
6: Add X, Y and $gain(X, Y)$ to ordered list.
7: Mark nodes X and Y.
8: Update $D(n)$ for all unmarked nodes as if X and Y had switched.
9: **end while**
10: Pick j maximizing Gain, the sum of the first j gains on the ordered list.
11: **if** Gain > 0 **then**
12: Update $G_1 = G_1 - X + Y$.
13: Update $G_2 = G_2 - Y + X$.
14: Update $cost(G_1, G_2) = cost_{old}(G_1, G_2) - Gain$.
15: **end if**
16: **until** $Gain \leq 0$

task nodes have different workloads thus different weights and the initial partitions are not well balanced either. When these conditions are not satisfied, this KL algorithm, which is designed for min-cut problem, may give us low communication partitions but with inferior workload balance.

In order to overcome this problem, we decided to modify the gain function in original KL algorithm. In original KL algorithm, only edgecut gains are considered. In our modified gain function, we also consider the balance gain, that is the decrease in workload difference between two partitions. With balance in mind, our new gain function is formulated as 4.3, where $gain_{edgecut}$ is formulated as 4.4 and $gain_{balance}$ is formulated as 4.6. We can see that $\Delta_{edgecut}$ is the original $gain(X, Y)$ in KL algorithm. Parameter α is used to set the percentages of gains from edgecut and workload balance to the total gain. The bigger is the α, the higher percentage of edgecut gain contributes to the total gain. When α is equal to 1, this algorithm becomes the original KL algorithm.

$$gain(X, Y) = \alpha \cdot gain_{edgecut} + (1 - \alpha) \cdot gain_{balance} \tag{4.3}$$

$$gain_{edgecut} = \Delta_{edgecut} / edgecut_{old} \quad (4.4)$$

$$\Delta_{edgecut} = edgecut_{old} - edgecut_{new} \quad (4.5)$$

$$gain_{balance} = \Delta_{workloadDiff} / workloadDiff_{old} \quad (4.6)$$

$$workloadDiff = |workload(X) - workload(Y)| \quad (4.7)$$

$$\Delta_{workloadDiff} = workloadDiff_{old} - workloadDiff_{new} \quad (4.8)$$

4.5 Simulated Annealing Algorithm

Simulated annealing(SA) is a probabilistic and iterative algorithm. It simulates the metallic annealing process. During this process, the metal is first heated to a very high temperature so the atoms gain enough energy to break chemical bonds and become free to move. The metal is then slowly cooled down to a lower internal energy. The metal is then heated again and again to get the atoms out of local minimum internal energy and give them a chance to find the global minimum internal energy state.

When applying this technique to our problem, we can see that the lowest energy we are going to get here is the inter-processor communication cost or the workload difference between the processors or both. So it is easy to come up with algorithm 4. The difficulties are in how to tune the algorithm to get the best results. In simulated annealing algorithm, parameters such as initial temperature, temperature changing scheme and number of iterations under each temperature greatly affect the effectiveness of the algorithm.

In our experiment, the initial temperature T_0 is determined by first pairwise swapping the nodes in two initial random partitions until all the nodes have been swapped to the other partition. In each swapping, the energy which is the inter-processor communication cost of the resulting partitions is computed. The initial temperature is then set to 20 times the standard deviation of the energy for these swaps. This scheme can generate an initial temperature that accepts high percentage of swaps in the initial stages of annealing algorithm [33].

In our experiment, the temperature T is updated by $T \leftarrow k \cdot T$, where $0 < k < 1$ is an update factor. The adjustment of temperature T can involve complex procedure [33]. But in our experiment, we just keep it simple and use a constant number k to update the temperature.

The number of iterations under each temperature has a huge impact on the quality on the partition. Our guideline is to find a number that can give us a decent result but does not cost a lot of computation time. In our experiment, we use $L \cdot NumberofNodes$ as the number. L is determined by doing experiments and pick the smallest L that can satisfy our needs.

Algorithm 4 SA Task Mapping Algorithm.

```
1:  T ← T_0
2:  generate a starting solution s
3:  while T > T_stop do
4:      for i = 1 to i = L · Number of Nodes do
5:          generate a new solution t in the neighbor of s
6:          ΔE ← E(t) − E(s)
7:          if ΔE < 0 then
8:              s ← t
9:          else if exp(−ΔE/T_k) > random[0, 1] then
10:             s ← t
11:         end if
12:     end for
13:     T ← k · T
14: end while
```

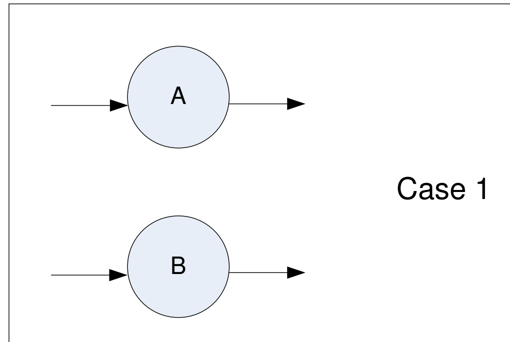
4.6 Merging and Duplication

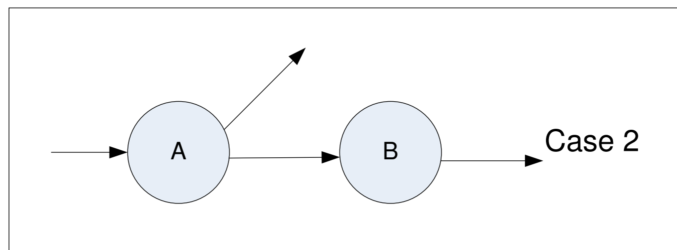
In the last chapter, duplication method is proposed. The duplication process slices the big nodes to multiple smaller nodes. This process greatly improves the balance of overall workload distribution. But after we do the duplication, from the resulting figure of node workload distribution, we can still find variations in workload distribution. The problem is caused by the smallest nodes. Those smallest nodes require little computation power but they still seize processors for themselves. The duplication process is not able to tackle this

problem so we need to do the inverse operation “merging” to further improve the balance of workload distribution. The idea is that we merge those small nodes together to produce large one or merge those small nodes to its bigger neighbors. In this way, it will yield spare computing units so we can have more processing units to do the duplication to improve the workload distribution. But the merge process is not as straightforward as duplication process. We can not arbitrarily merge any two small nodes. The rule we need to follow is that, when we merge two nodes we simply combine these two nodes together, we don’t add any new functionalities to the merged node. Figure 4.3 depicts three situations where nodes *A* and *B* are not able to be merged. In case 1, node A and B are not neighbors. In case 2, node A has more than one outgoing edge. In case 3, node B has more than one incoming edge. These three situations are where nodes can not be merged. In situation depicted by Figure 4.4, node A and B can be safely merged. In this situation, node A has only one outgoing edge and node B has only one incoming edge. In this situation, we call node A and node B are eligible to be merged. This is one of the situations that our merge algorithm will try to identify and use.

Following these rules, we can develop our merge-and-duplicate algorithm. There are three possible schemes depending on the order we do the merge and duplication. We can do the merge first then do the duplication. Or we can do the duplication first and then do the merge operation. Or we can do the duplication and merge alternatively. Here we explore the first scheme. We first do the merge and then we do the duplication. The process goes as follows. We first compute the optimal balanced workload for each processing unit. Then we search our task graph to find the nodes that have workload less than the optimal workload. We then try to merge the nodes together or to their neighbors. The process can be depicted by the algorithm 5.

Our experimental result shows that this merge-then-duplicate algorithm can produce a better workload distribution then the original duplication algorithm. This is verified by the Figure 4.5.





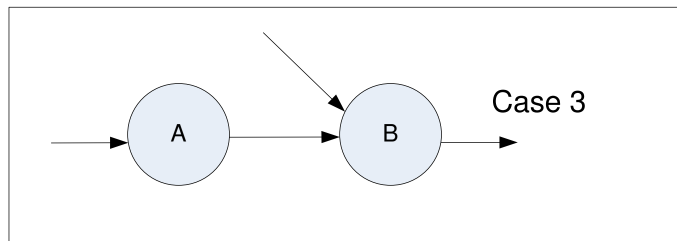


Figure 4.3. Situations where Nodes A and B can not be merged

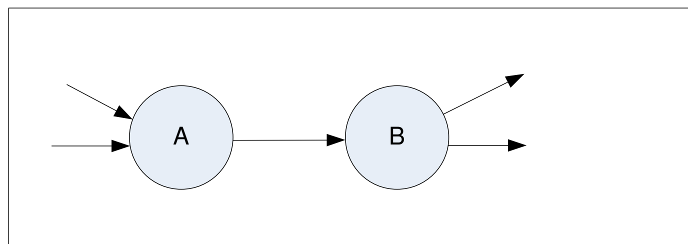


Figure 4.4. Situations where Nodes A and B can be merged

Algorithm 5 Task Merge-then-duplicate Algorithm.

1: unmark all nodes
2: $w_{opt} = W_{total}/N \cdot M$
3: **while** $\exists$ *unmarked* $node_i$ *with* $w(node_i) < w_{opt}$ **do**
4: find the unmarked $node_i$ with the smallest w($node_i$)
5: check with its neighbors
6: **if** mergeable **then**
7: merge the node to its neighbor
8: **else**
9: mark $node_i$
10: **end if**
11: **end while**
12: do the duplication

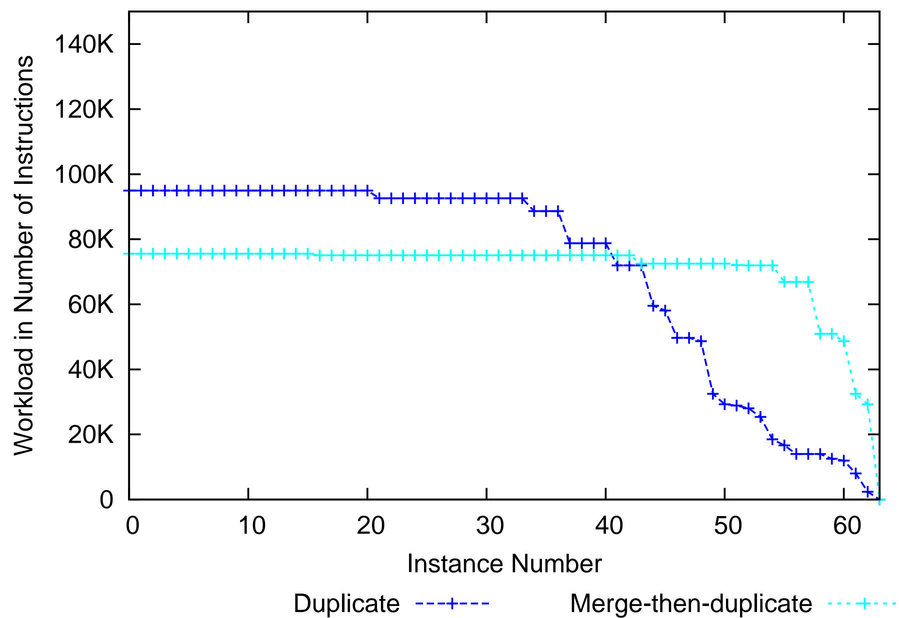


Figure 4.5. Comparison of duplicate and merge-then-duplicate schemes

CHAPTER 5

EVALUATION OF ALGORITHMS

In this chapter, we implement the algorithms described in the last chapter, generate the results, plot them and do the comparison. We also search for the suitable parameters for extended KL algorithm and simulated annealing algorithm. Finally we do the explorations in an architectural perspective. We apply our algorithms to different multi-core architectures and plot and analyze the results.

5.1 Simulation Environment

Our simulation can be divided into two phases: Workload Profiling and Task Mapping. During workload profiling, we use PacketBench [28] to evaluate the processing requirements of each packet in a trace of network traffic. PacketBench provides an instruction trace of each processor instruction executed and thus allows us to accurately determine utilization parameters $u^{\tau}(t_i)$ and $u^{\tau}(e_{i,j})$ for each interval τ and the distribution of service time S_i (measured in instructions executed). During task mapping, tasks are duplicated/merged and mapped as described above. This process is repeated for each interval τ. The overall process is depicted by the flow graph 5.1. In figure 5.1, diamonds represent files and rectangles represent programs. First PacketBench takes packet trace file and network application source file and produces instruction trace file. Graph generator then takes network application file and instruction trace file and produces task graph. Then task graph and instruction trace file are put in profile generator to get a annotated task graph. Then our mapping algorithm is used to generate mapping for our system.

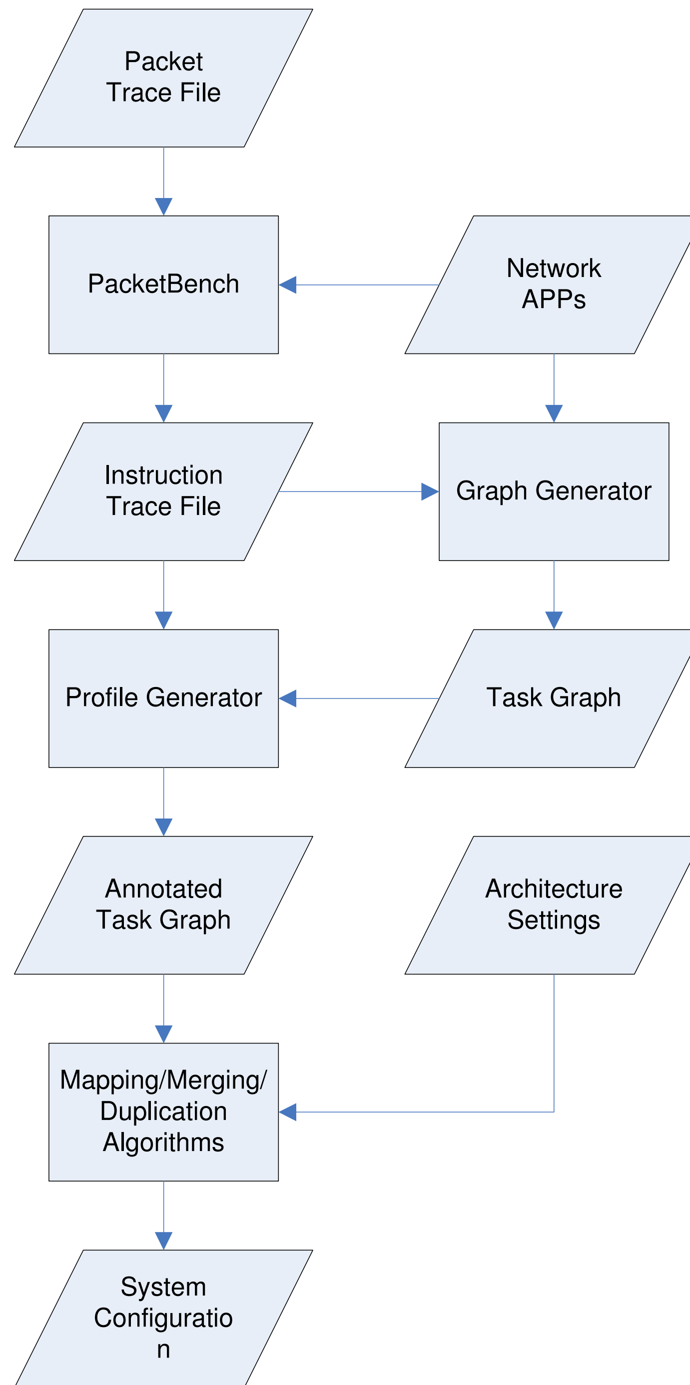


Figure 5.1. Overall Simulation Flow

In our simulation, we first assume a packet processing system with $N = 8$ processors with $M = 8$ threads each. The processor interconnection provides connectivity from any processor to any other processor. More architecture settings are examined in the architecture exploration section. We assume that re-mapping takes place at intervals of 1000 packets.

We use two different packet traces in our experiments in order to exercise the system with network traffic that exhibits different levels of workload dynamics:

- Trace 1: This trace is obtained from the Internet uplink of our institutional network. It represents real network traffic and exhibits a low amount of dynamic variation. The trace is 100 intervals long.

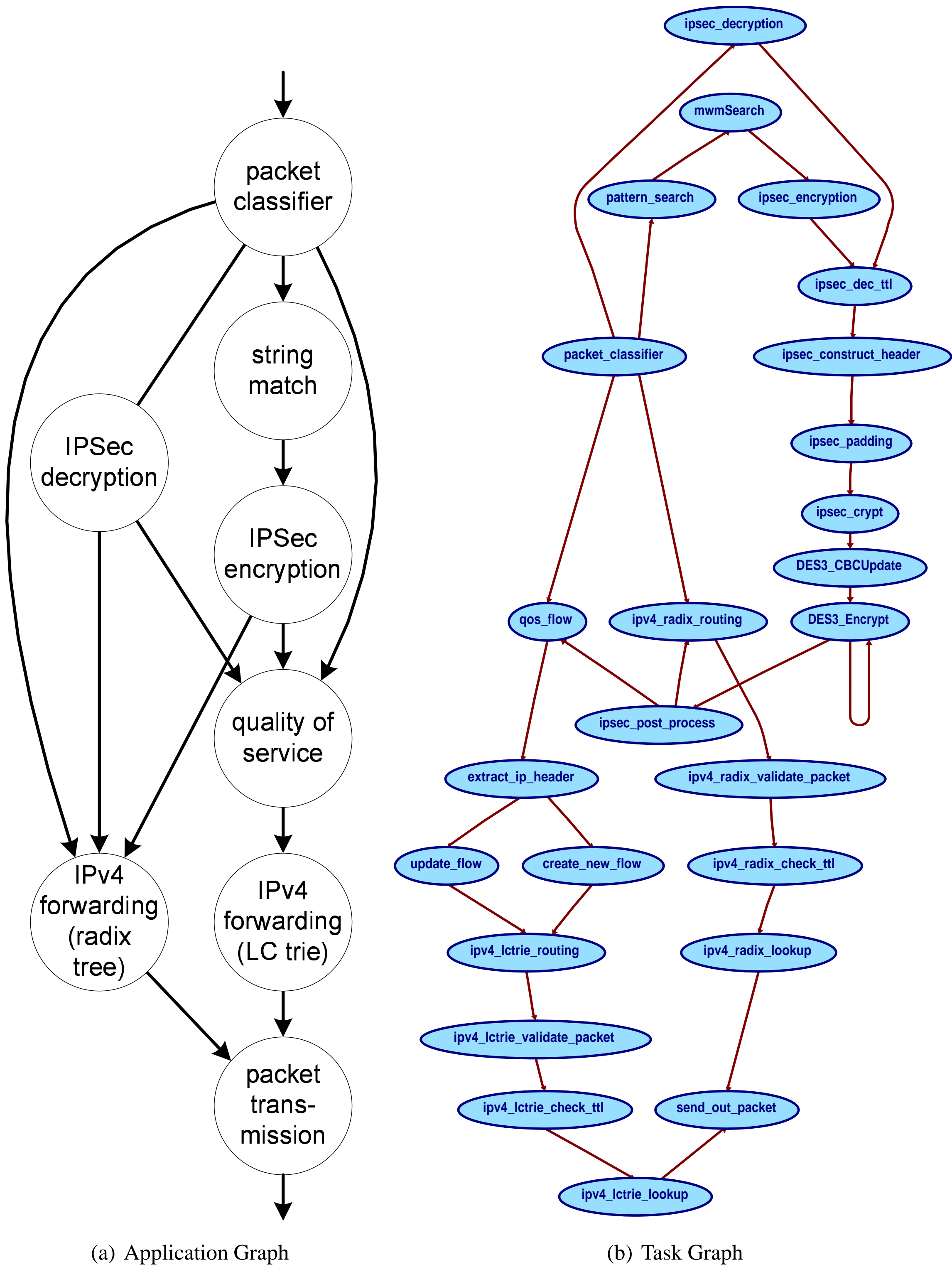
- Trace 2: This trace was generated synthetically by splicing several different traces together. The resulting workload changes dramatically every 10 intervals so require a drastic change in allocated processing tasks. The trace is 40 intervals long.

The processing applications in our workload are shown in Figure 5.2(a) with their respective dependencies. By partitioning these eight applications, we obtain the task graph shown in Figure 5.2(b). The 25 tasks shown in Figure 5.2(b) are labeled with their functional descriptions. Edges illustrate the possible paths of packets through the system.

5.2 Profiling

The results of the profiling phase are shown in Figure 5.3 and Figure 5.4. For each processing task from Figure 5.2(b), we show the amount of processing work, w_i, that is necessary. Recall that this value depends on the processing complexity of the task and its utilization. The utilization of tasks for both traces are also shown in Figure 5.5 and Figure 5.6

From the workload profiling figures, we could observe that there is a very large difference between tasks in terms of processing requirements. The variation of w_i for any given



(a) Application Graph

(b) Task Graph

Figure 5.2. Experimental application.

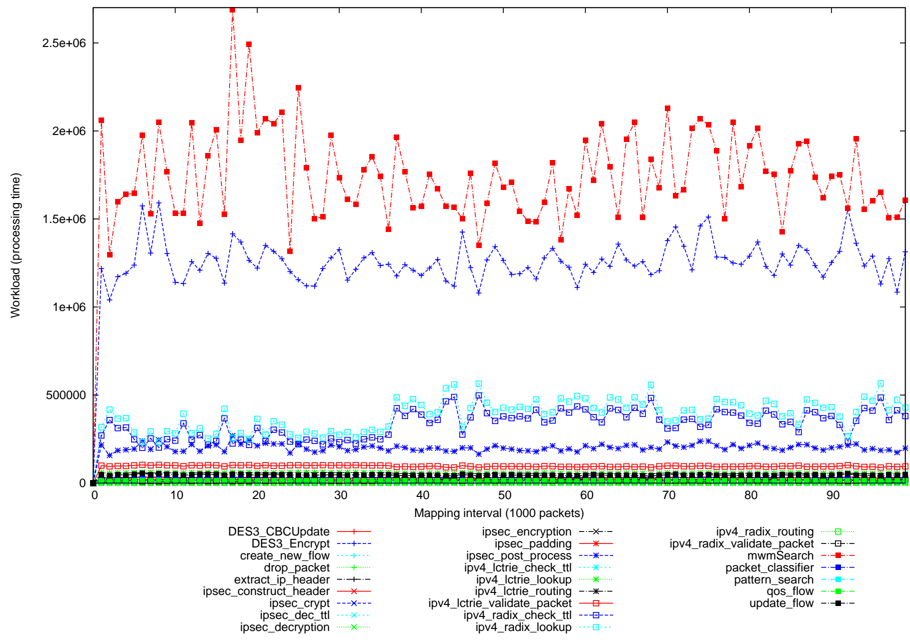


Figure 5.3. Workload for Trace 1.

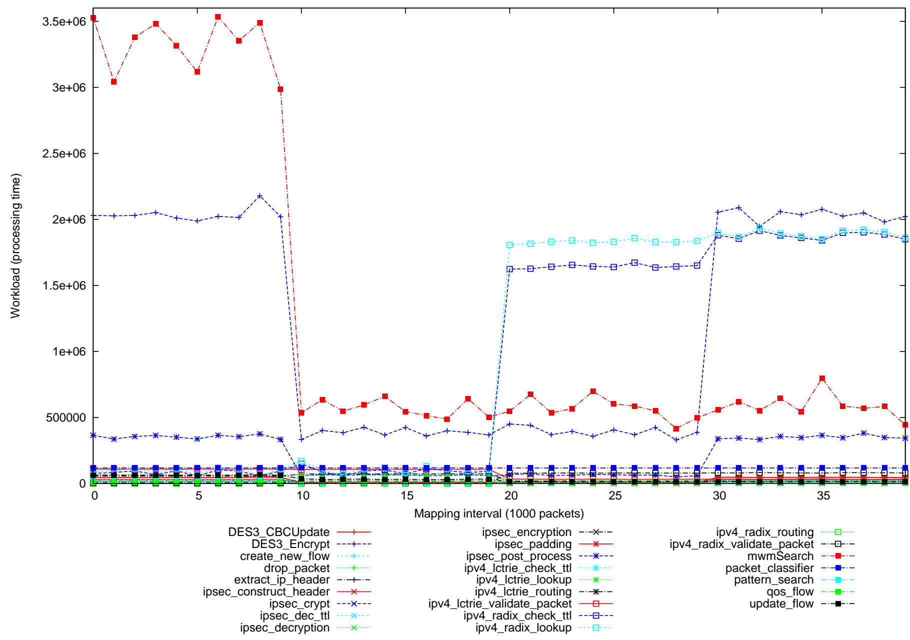


Figure 5.4. Workload for Trace 2.

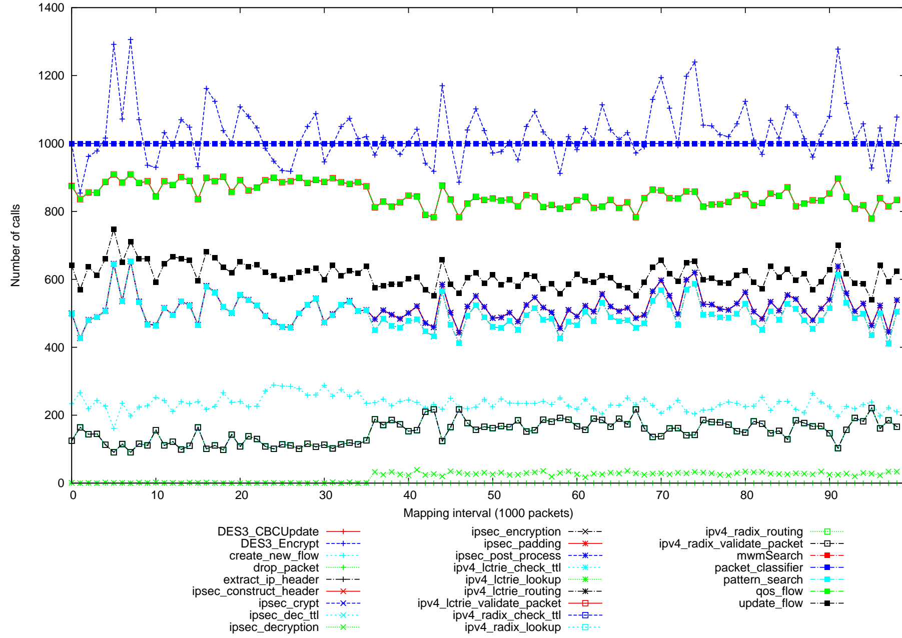


Figure 5.5. Utilization of tasks for Trace 1.

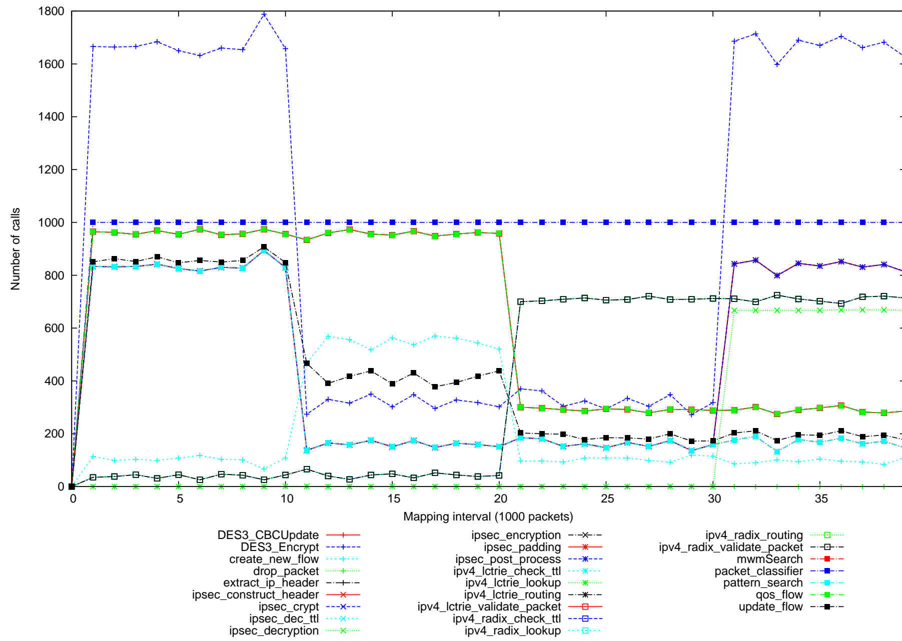


Figure 5.6. Utilization of tasks for Trace 2.

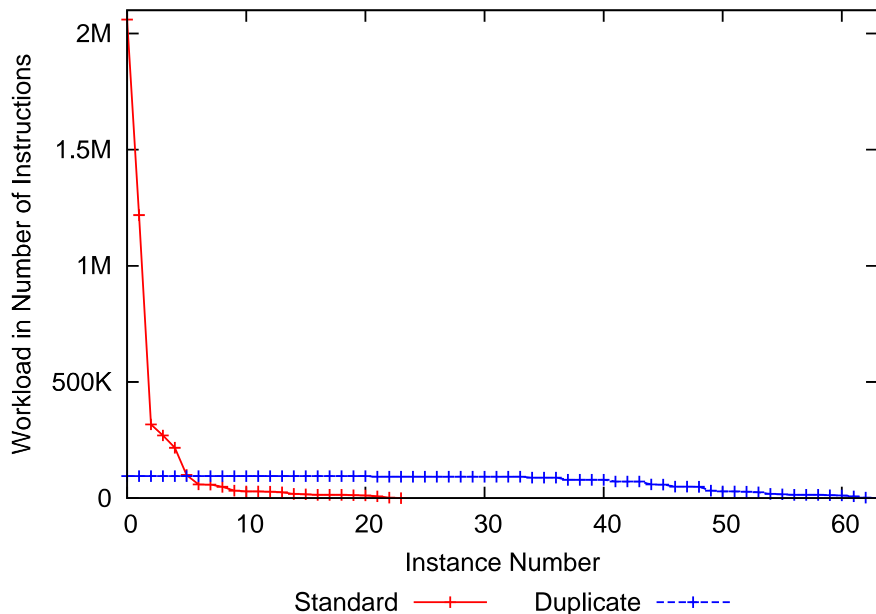


Figure 5.7. Distribution of Work w'_i per Task Instance Before and After Duplication for Trace 1.

task is low for Trace 1 (Figure 5.3). In contrast, Figure 5.4 shows high variations due to the changes in network traffic every 10 intervals.

For the utilization of tasks graphs, we could also observe that utilization of tasks are highly dependent on the content of traffic.

These profiling results provide evidence for two observations we have made earlier: (1) there is a big difference in processing requirements among tasks and (2) these requirements change dynamically as network traffic changes.

5.3 Duplication

From the previous chapters, we know that duplication is necessary to obtain a balanced workload distribution. So we first use duplication of selected tasks to obtain a more balanced workload. The resulting work w'_i (from Equation 4.2) is shown in Figure 5.7 and Figure 5.8. These figures show the amount of work per task instance before and after duplication for one interval from Trace 1 and Trace 2 respectively. Before duplication, only

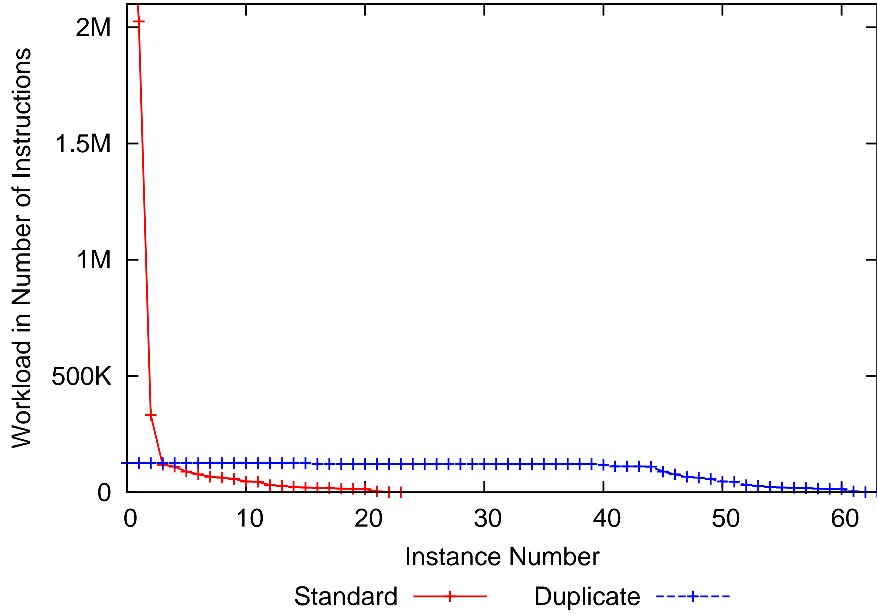


Figure 5.8. Distribution of Work w'_i per Task Instance Before and After Duplication for Trace 2.

25 task instances exist and their processing requirements differ by several orders of magnitude. After duplication, we have 64 task instances (since $N \cdot M = 64$ in our experimental setup) with very balanced w'_i (except for the smallest tasks).

5.4 Mapping:UDFS

This section presents the results from previously designed UDFS algorithm. To evaluate the quality of the mapping algorithm, we consider two metrics:

- Average Processor Utilization $\overline{u}$: The average utilization $\overline{u}$ of all processors is the sum of all work allocated to each processor divided by N times the maximum allocation:

$$\overline{u} = \frac{\sum_{j=1}^{N} \left(\sum_{\{i|m(t_i)=j\}} w'_i \right)}{N \cdot \max_j \left(\sum_{\{i|m(t_i)=j\}} w'_i \right)}. \tag{5.1}$$

When each processor's work allocation is close to the maximum, then the overall average utilization is high. Higher utilization implies that more work gets done and

more packets get processed (since the total amount of work $\sum_{i=1}^{T} w_i$ is constant for any mapping result). Thus, utilization is directly related to the maximum line rate (i.e., throughput) R of the packet processing system: $R \sim \overline{u}$. Thus, higher utilization $\overline{u}$ indicates higher system performance.

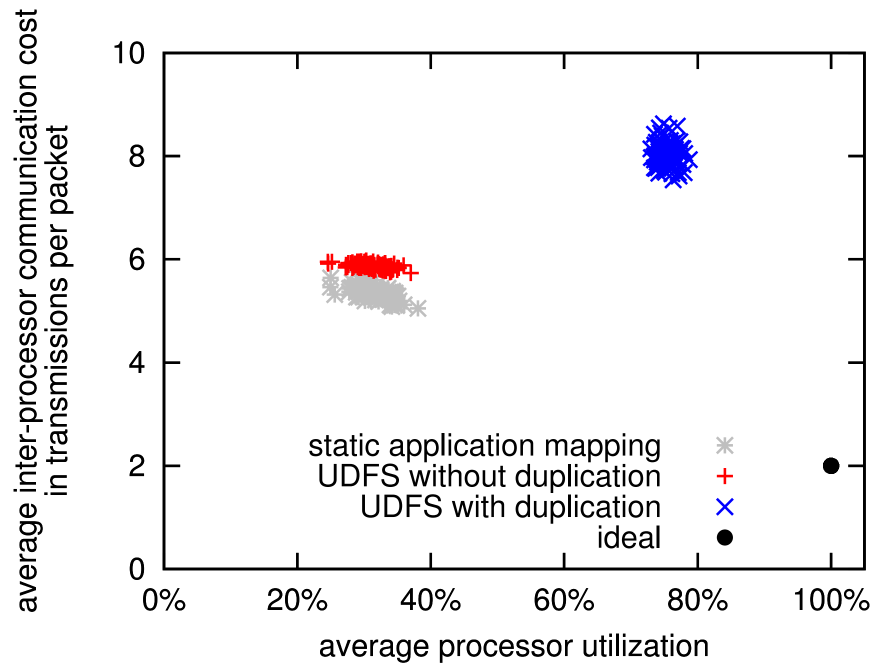
- Average Inter-Processor Communication Cost $\overline{c}$: The average communication cost $\overline{c}$ represents the number of times a packet has to be sent across the processor interconnect:

  $$\overline{c} = \sum_{\{i,j|m(t_i)\neq m(t_j)\}} u(e_{ij}). \tag{5.2}$$

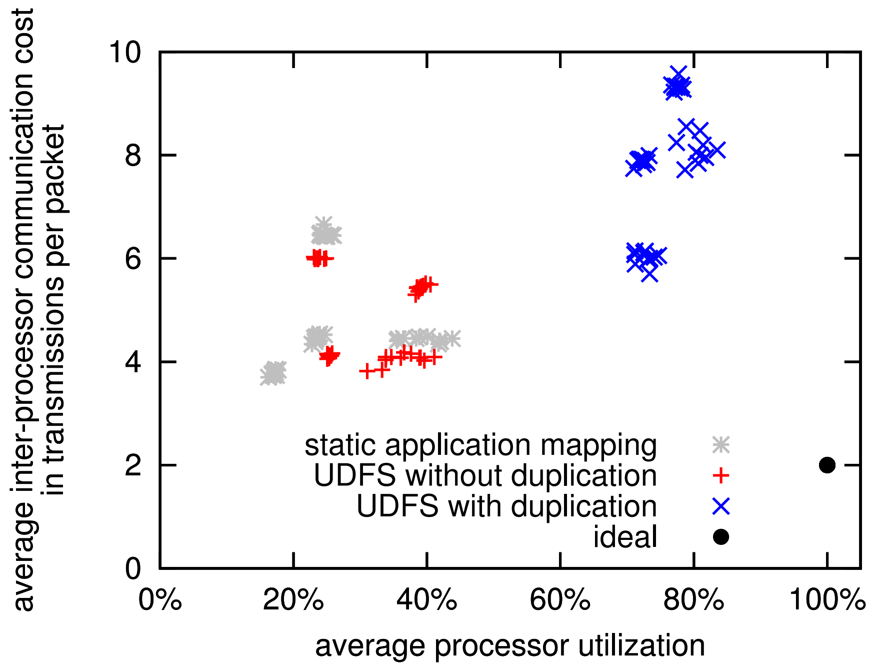
  At a minimum, each packet has to be sent once from the incoming interface to a processor and once from the processor to the outgoing interface. Thus, $\overline{c} \geq 2$. Higher values for $\overline{c}$ imply more load on the interconnect. Therefore, lower values of $\overline{c}$ are desirable.

Figure 5.9 produced by previously designed algorithms shows a comparison of the performance of three different algorithms using metrics $\overline{u}$ and $\overline{c}$. As baseline, *static application mapping* is shown, which represents the conventional approach to task management on multi-core packet processing systems. Each application a_i is allocated to a different processor. The UDFS algorithm is shown in two instances – without duplication and with duplication. The prior is an intermediate result to illustrate the importance of task duplication. The ideal scenario of full utilization and a two packet transmission (one ingress, one egress) is also shown for comparison. The data in Figure 5.9 show clearly that UDFS mapping with task duplication achieves the highest system utilization $\overline{u}$ and thus the highest data rate R. UDFS mapping without task duplication is practically equivalent to static mapping since the imbalance in the amount of work w_i per task prevents an effective utilization of processors.

The overall performance improvement of UDFS (with duplication) over conventional static application mapping is shown in Table 5.1. An increase in throughput (due to $R \sim \overline{u}$)



(a) Trace 1



(b) Trace 2

Figure 5.9. Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different Mapping Algorithms.

Table 5.1. Comparison of UDSF Mapping to Static Application Mapping.

	Communication Cost $\overline{c}$	Throughput R
Trace 1	1.49×	2.39×
Trace 2	1.64×	2.89×

Table 5.2. Comparison of KL Mapping to UDFS Mapping.

	Communication Cost $\overline{c}$	Throughput R
Trace 1	0.81×	1.01×
Trace 2	0.80×	1.00×

of 2.39–2.89× can be achieved at a cost of 1.49–1.64× higher inter-processor communication.

5.5 Mapping: KL Algorithm

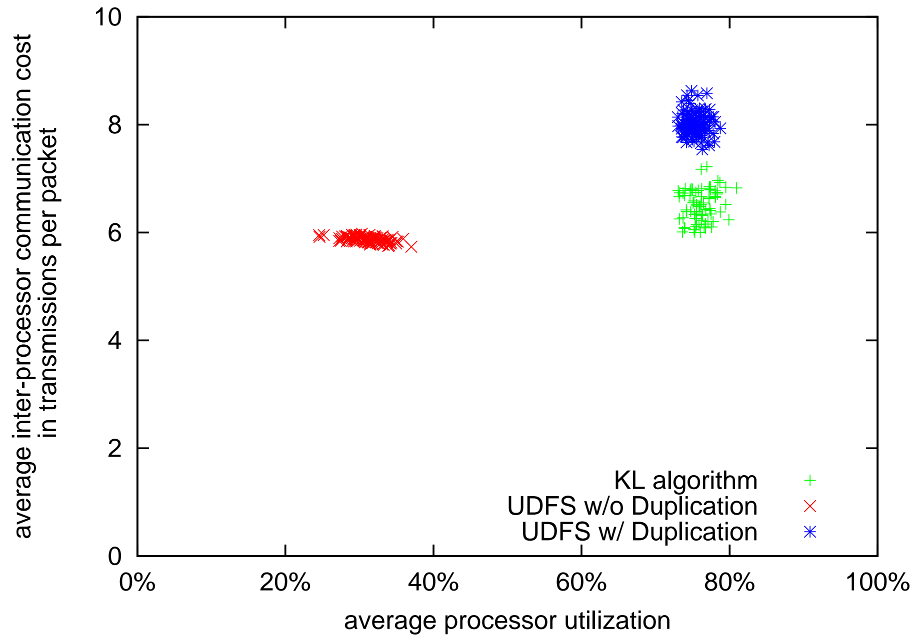
In this section, we implement the KL algorithm and plot the results. We also do the comparison between KL algorithm and previously designed algorithms.

With our improved algorithm, we can get even better results compared to UDFS algorithm. We show the results of KL algorithm in Figure 5.10. We can see that KL algorithm produces mappings that require less inter-processor communication cost while maintaining the similar throughput as UDFS algorithm.

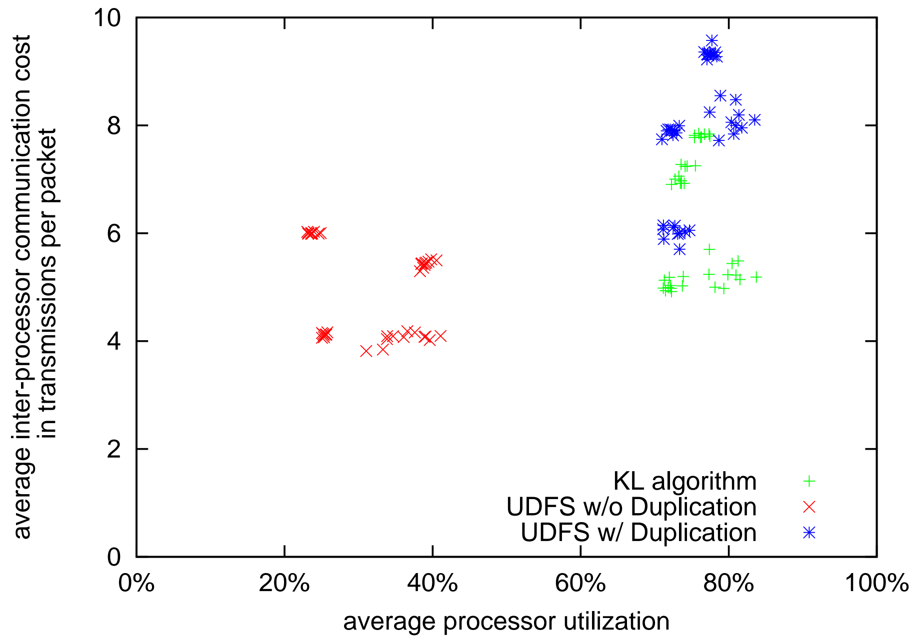
So the overall performance improvement of KL algorithm over conventional static application mapping is better than that of UDFS algorithm. Table 5.2 shows that KL algorithm only requires 0.80× of the communication cost as UDFS algorithm while obtaining the same or even better throughput.

5.6 Mapping: Extended KL

As we mentioned in the previous chapters, KL algorithm has some limitations when applied to our mapping problem. So we modify the basic KL algorithm and come up with this extended version of KL algorithm. In this extended KL algorithm, the gain function is



(a) Trace 1



(b) Trace 2

Figure 5.10. Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different Mapping Algorithms.

modified to not only consider the edgecut gain but also workload balance. The new gain function is described by function 4.3. We plot the results for different α in figure 5.11.

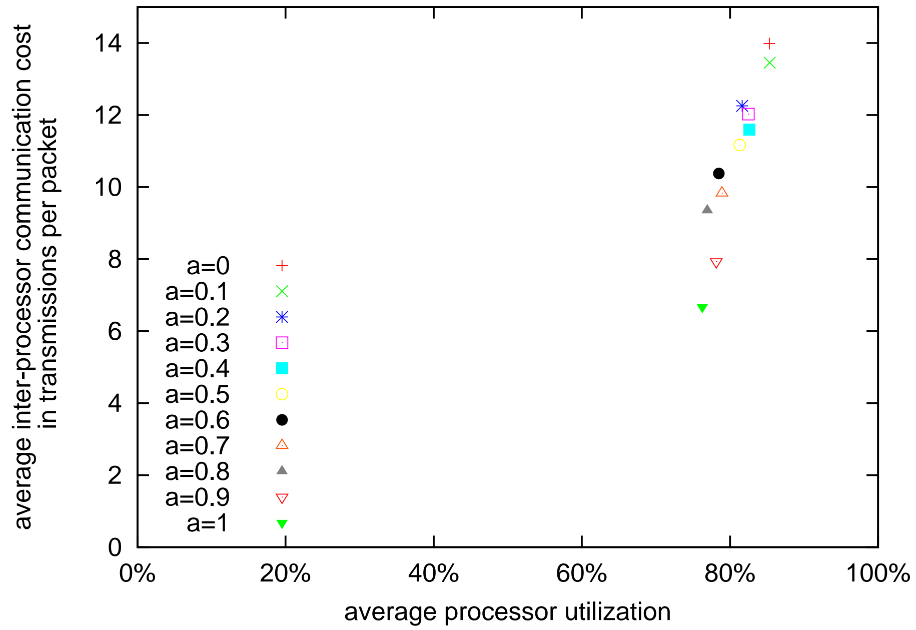
From the figures, we can see that as we increase the α, the average processor utilization decreases and the inter-processor communication decreases too. This is because when α is small, we put more effort to optimize the processor utilization, while α is approaching 1, we put more effort on inter-processor communication. To find the best parameter α for this algorithm, we need to consider the particular system. In the case where inter-processor communication bandwidth is well sufficient, we may want to pick a smaller α so that we can get a better utilization. While we have limited inter-processor communication but sufficient computing power, we will want a bigger α to get a better overall performance.

5.7 Mapping: Simulated Annealing Algorithm

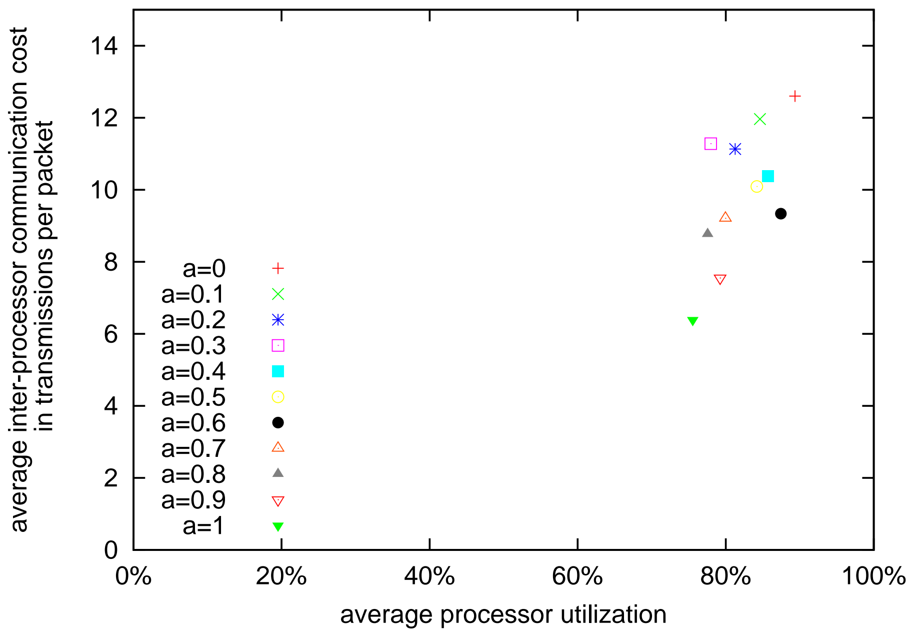
As mentioned in the previous chapters, the difficulty of simulated annealing algorithm is to find a good set of parameters that can give us good partitions. We already explained the ways to obtain some of the parameters. In this section, we show the way to obtain parameter L, the factor for the number of iterations in each temperature. Then we show the results of simulated annealing algorithms and compare them to results from KL algorithm.

First, we use inter-communication cost as energy function and do the following experiments. To search for the suitable value of L, we run the simulated annealing algorithm for L from 1 to 20 and choose the one that give us the best results in terms of communication cost and utilization. The results are plotted in the Figure 5.12

From the above figures, we can see that from $L = 1$ to $L = 20$, the inter-processor communication costs and utilizations have only small variations. So to speed up the algorithm, we use $L = 1$ as the parameter for all the following SA algorithm implementation. The results are plotted in the figure 5.13(a) and figure 5.10(b). From the figures, we can see that SA algorithm can produce the mappings that have the similar qualify as produced by KL

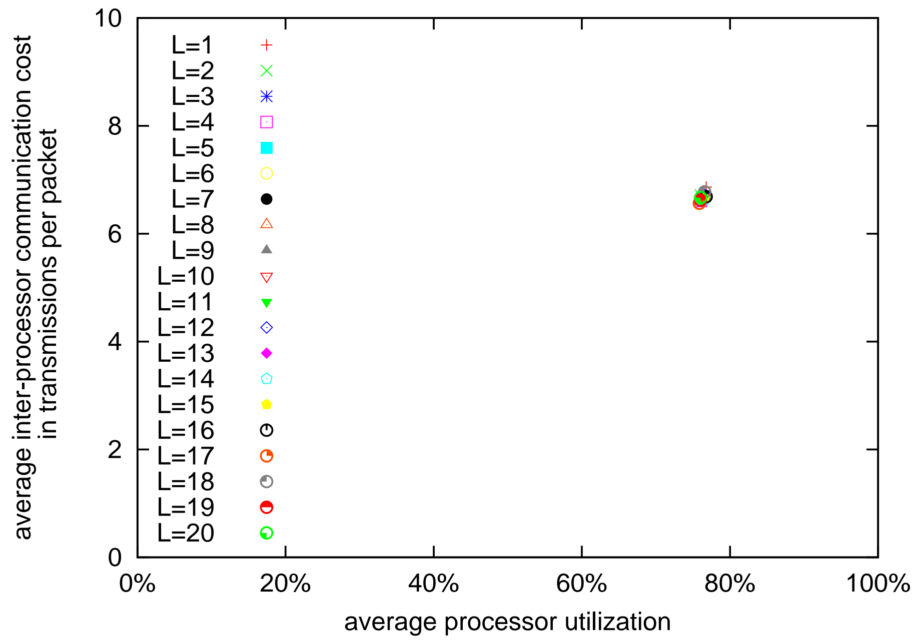


(a) Trace 1

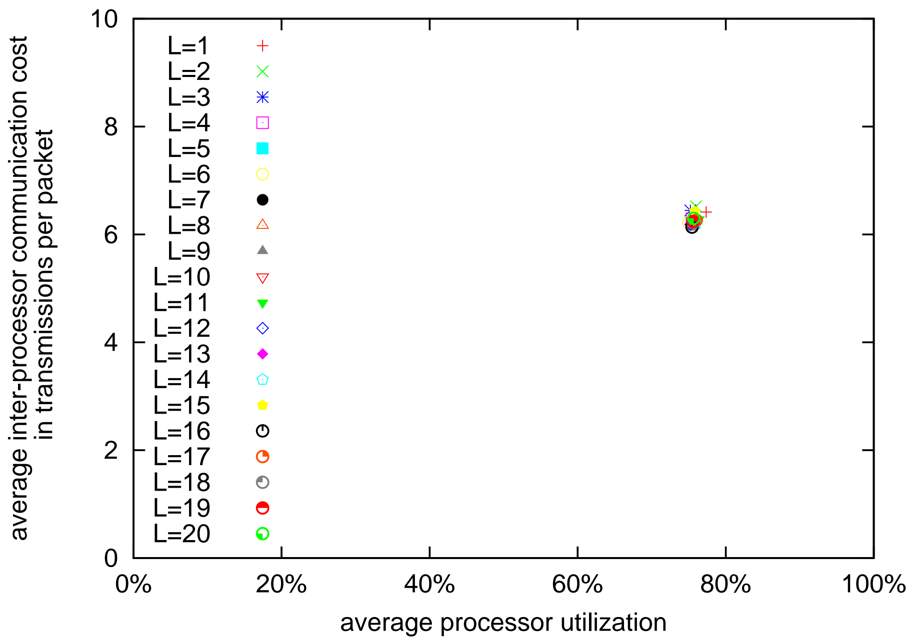


(b) Trace 2

Figure 5.11. Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different α.



(a) Trace 1



(b) Trace 2

Figure 5.12. Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different L.

Table 5.3. Comparison of SA Mapping to UDFS Mapping.

	Communication Cost $\bar{c}$	Throughput R
Trace 1	0.83×	1.00×
Trace 2	0.81×	1.00×

Table 5.4. Comparison of UDFS algorithm with and without merging.

	Communication Cost $\bar{c}$	Throughput R
Trace 1	0.63×	1.22×
Trace 2	0.69×	1.20×

algorithm. The exact number is shown in the table 5.3. We can see that the mapping results from KL algorithm are slightly better than KL algorithms.

5.8 Merging and Duplication

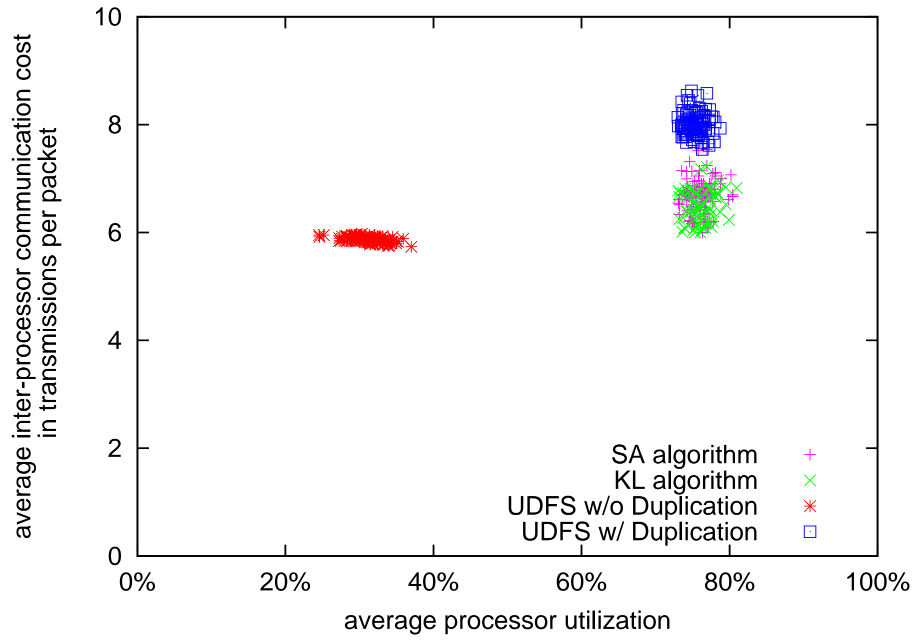
As mentioned in the last chapter, duplication can not fully evenly distribute the workload on processors. This is where merging comes to the rescue. We also show some trivial results in the last chapter. In this section, we will see if the new graph can actually improve the utilization and hopefully the inter-processor communication cost also.

First we show the workload distribution after merging process with original duplication only workload distribution in Figure 5.14 and Figure 5.15. We can see that with merging operation, the workload distribution becomes more even than with duplication only.

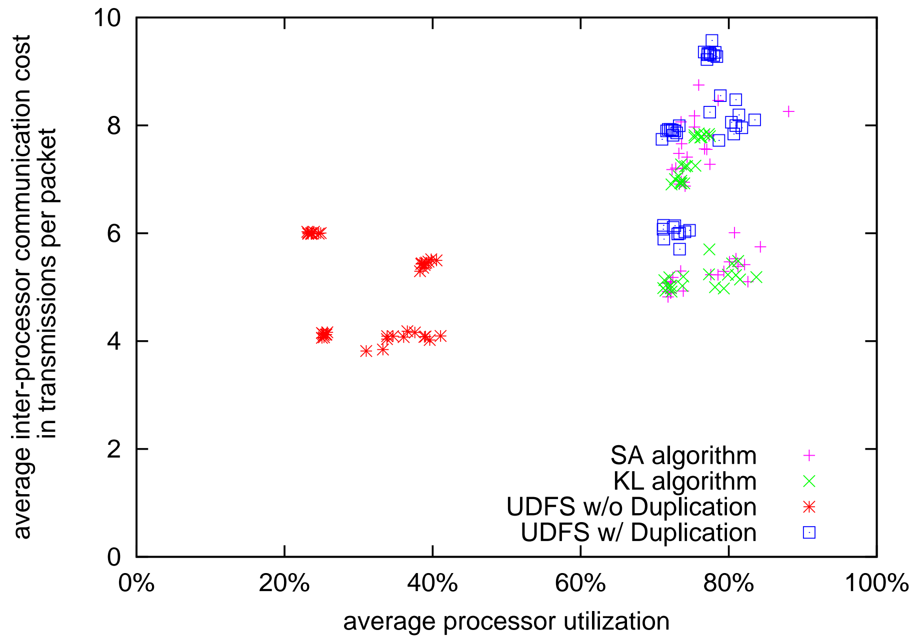
Using this new task graph, we run our algorithms again to obtain the new mappings. The results are depicted in the Figures 5.16, 5.17, 5.18. From the figures, we can see that merging and duplication process improves both inter-processor communication and utilization. The exact numbers are shown in the table 5.4, table 5.5 and table 5.6.

Table 5.5. Comparison of KL algorithm with and without merging.

	Communication Cost $\bar{c}$	Throughput R
Trace 1	0.80×	1.21×
Trace 2	0.82×	1.21×



(a) Trace 1



(b) Trace 2

Figure 5.13. Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for Different Mapping Algorithms.

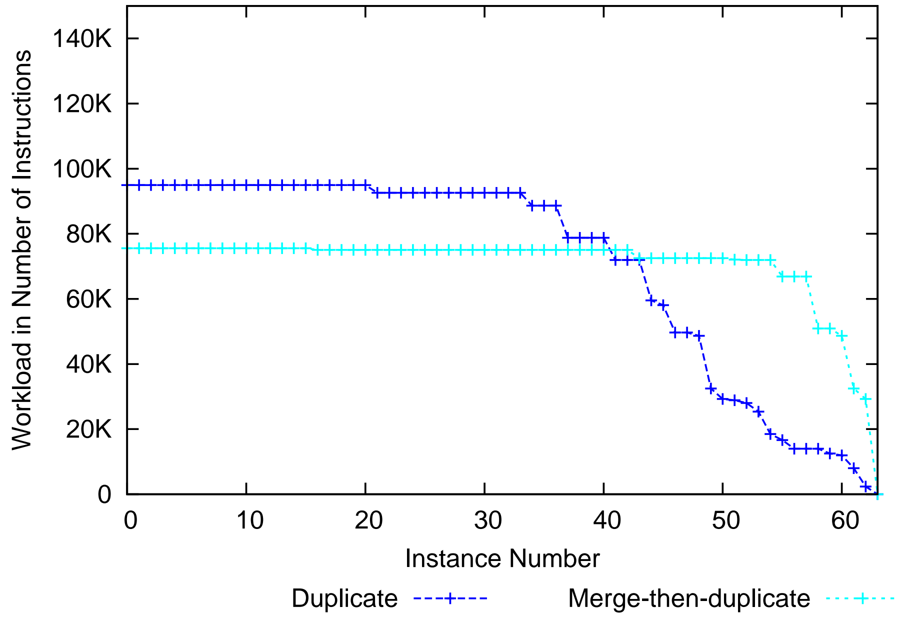


Figure 5.14. Distribution of Work w_i' per Task Instance with and without merging for Trace 1.

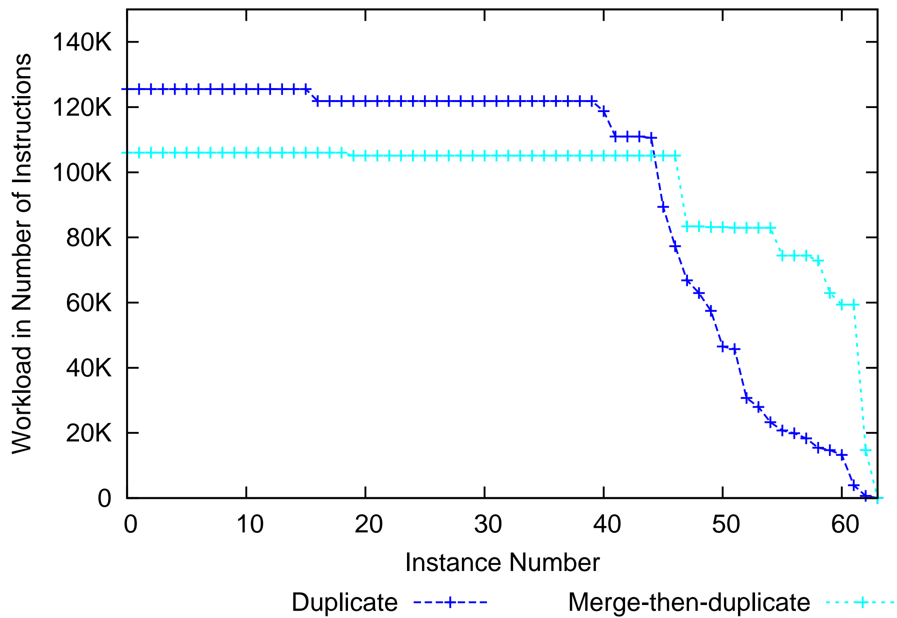
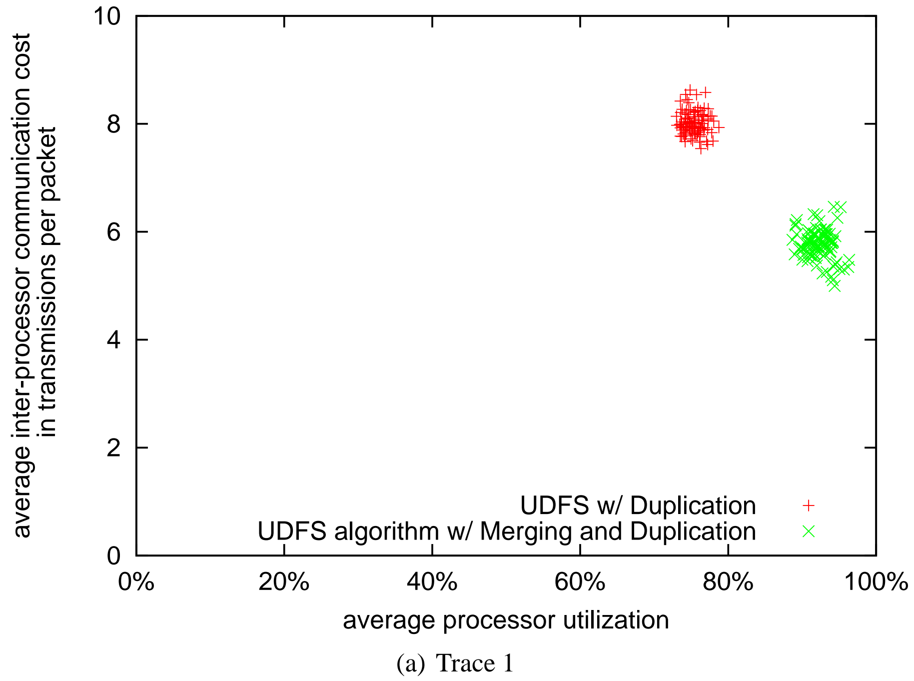
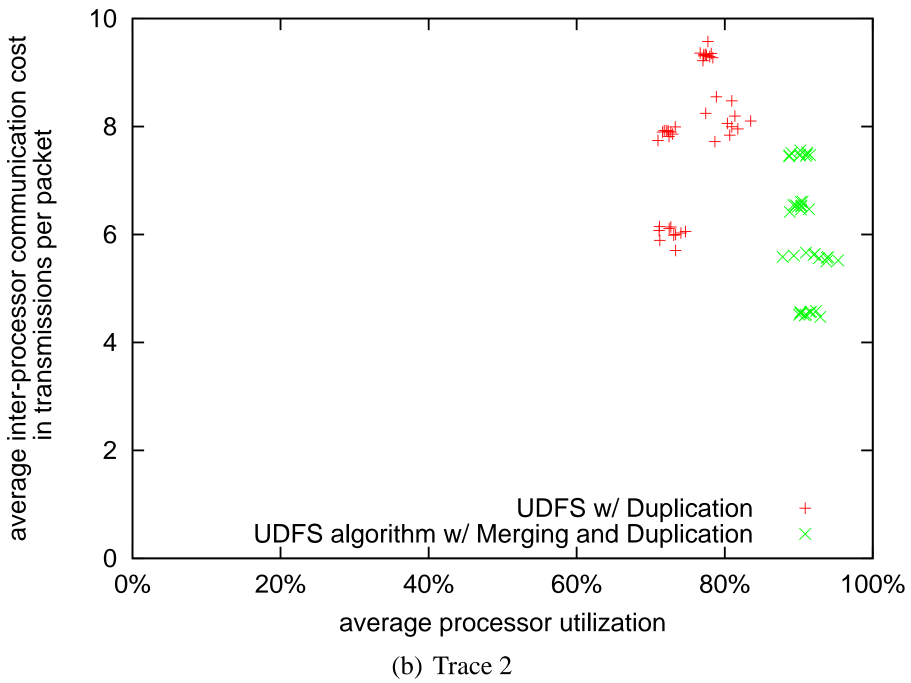


Figure 5.15. Distribution of Work w_i' per Task Instance with and without merging for Trace 2.



(a) Trace 1

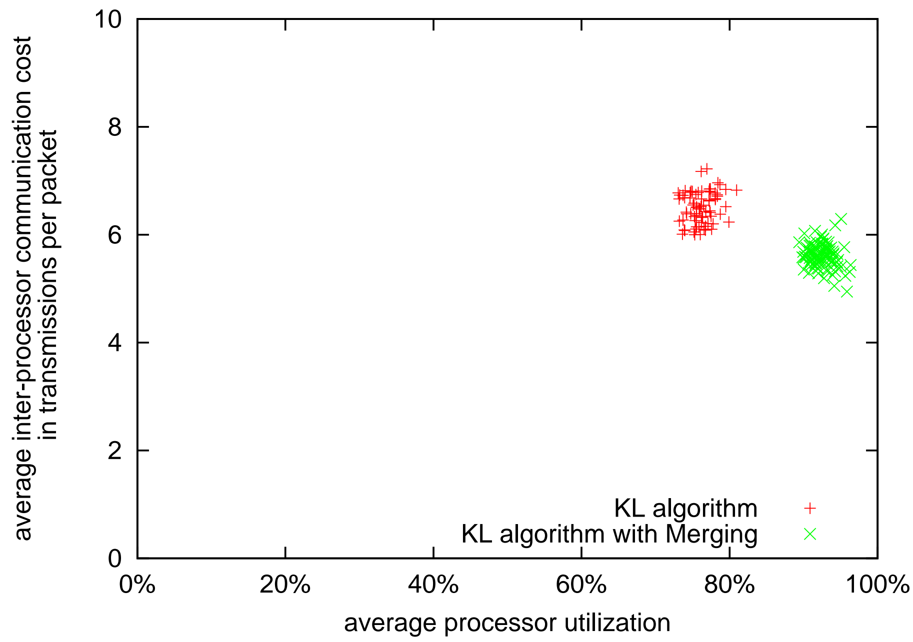


(b) Trace 2

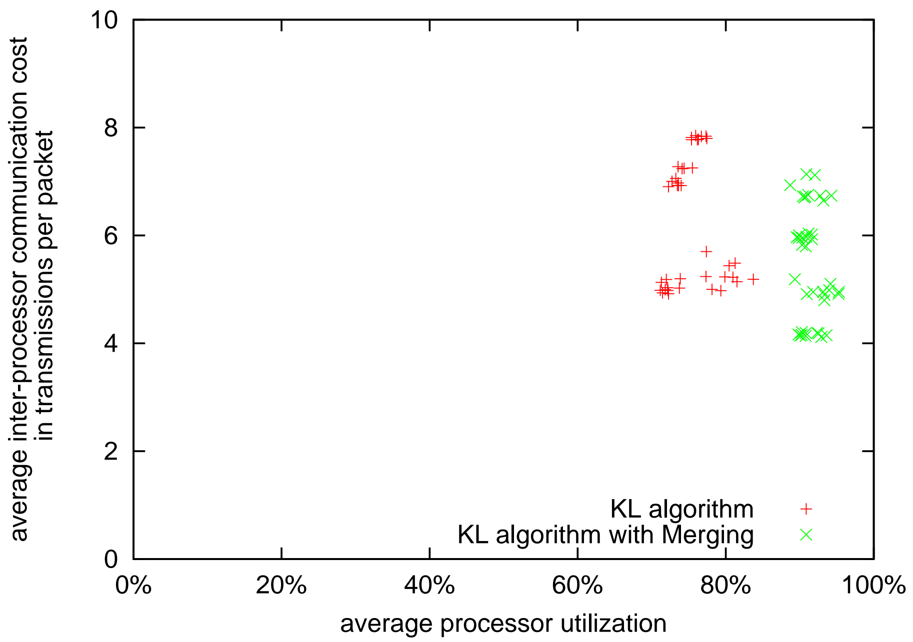
Figure 5.16. Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$.

Table 5.6. Comparison of SA algorithm with and without merging.

	Communication Cost $\overline{c}$	Throughput R
Trace 1	0.77×	1.22×
Trace 2	0.80×	1.20×

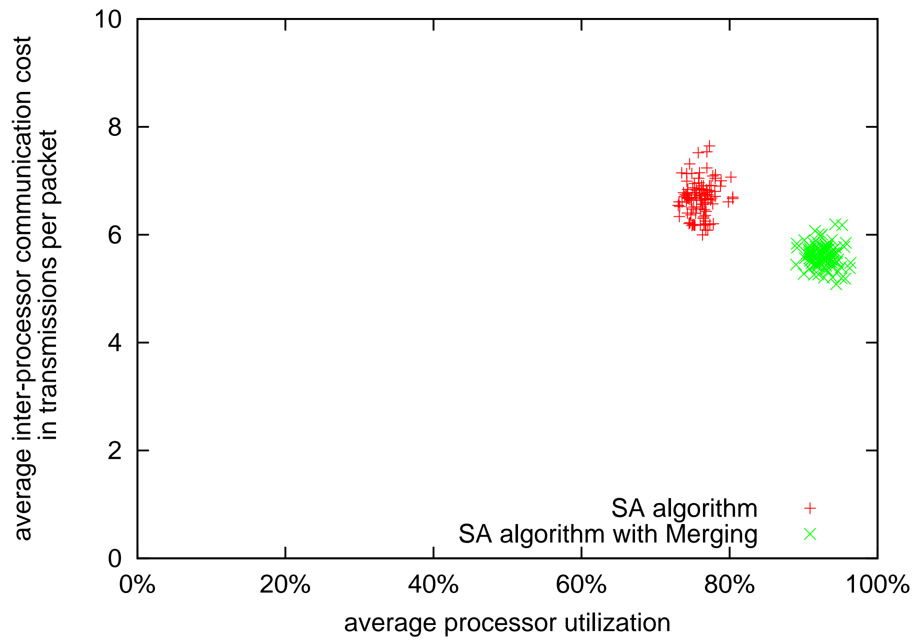


(a) Trace 1

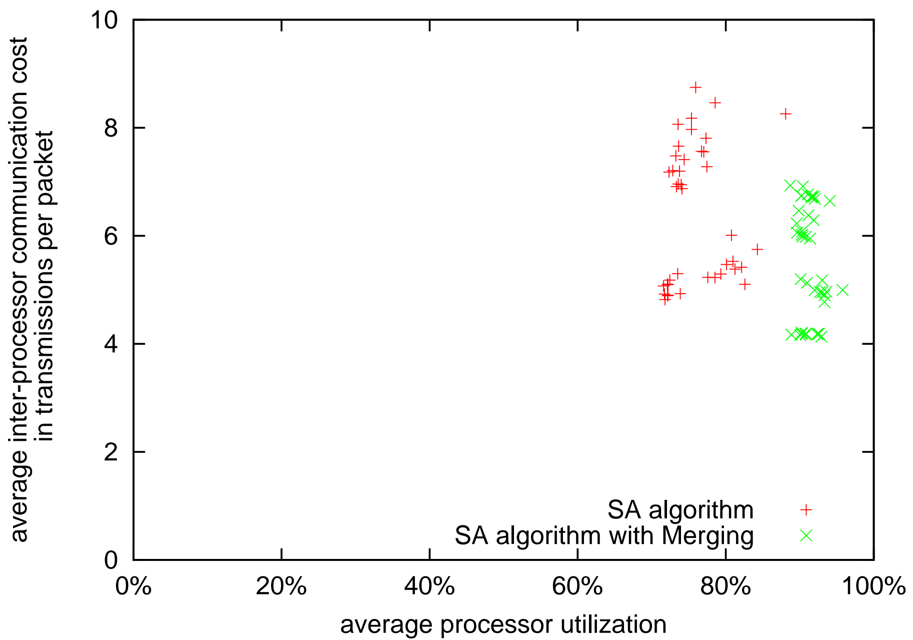


(b) Trace 2

Figure 5.17. Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$.



(a) Trace 1



(b) Trace 2

Figure 5.18. Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$.

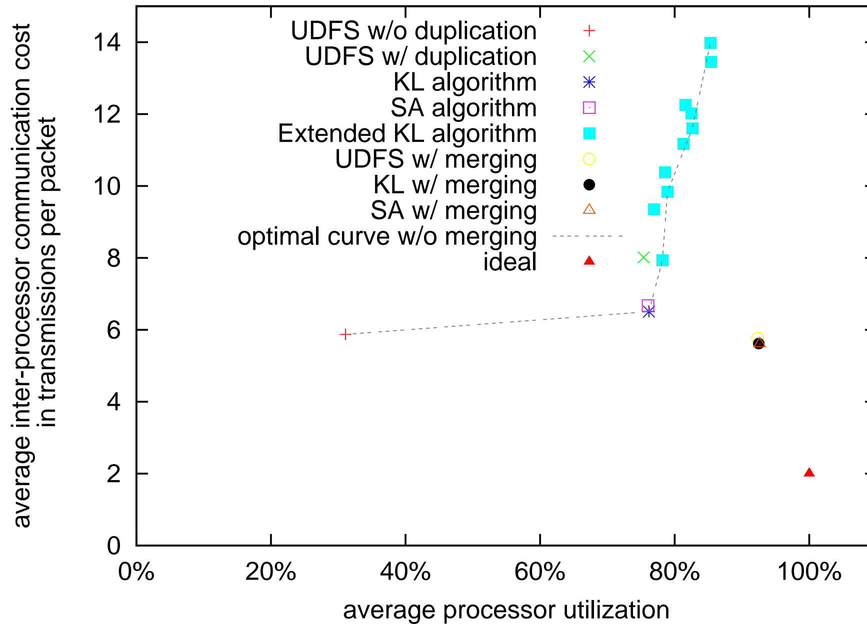


Figure 5.19. Interconnect Bandwidth $\overline{c}$ in Comparison to Processor Utilization $\overline{u}$ for All Mapping Algorithms

After we got all the simulation results of all the algorithms, it is more informative to put all of them in one figure. In figure 5.19, results from all algorithms are plotted. From figure, we can see that with merging process, better mapping results can be obtained.

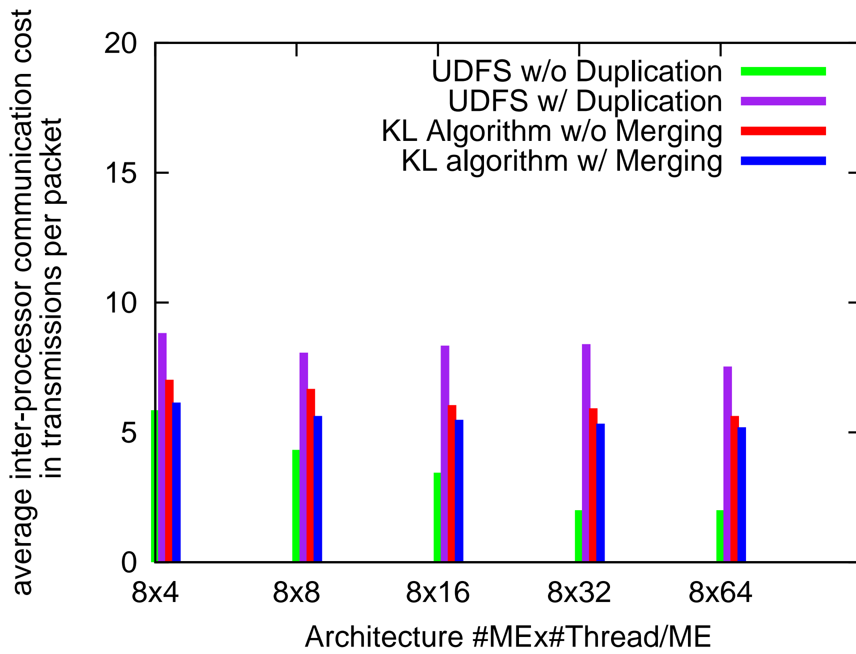
5.9 Architecture Exploration

After developing the mapping algorithms for our packet processing systems, we will apply the algorithms to different packet processing system architectures in this section.

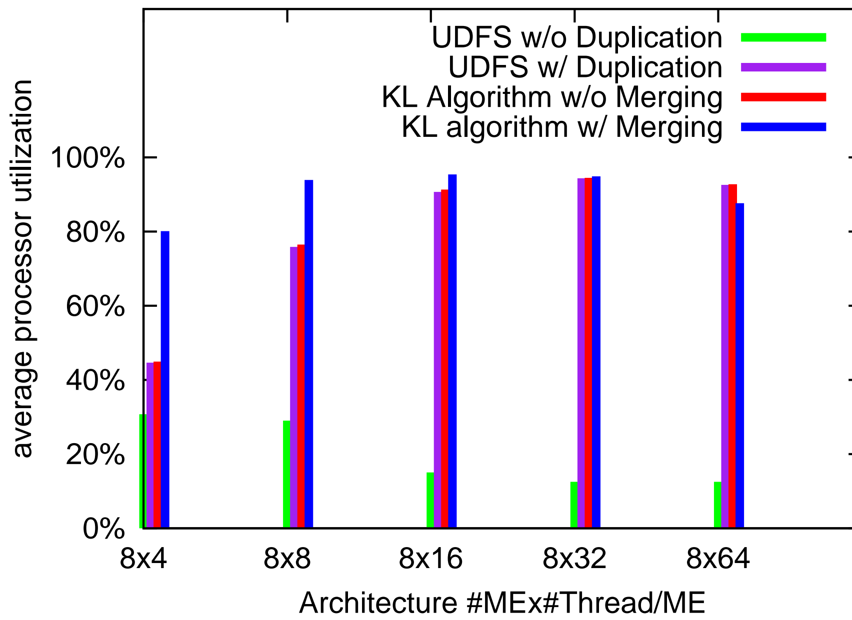
Our default architecture has 8 processing cores and each one can accommodate 8 threads. In this architecture, the total number of threads that can run in parallel is 64. This architecture serves as a good starting point for our algorithms research. But it will be very interesting to see how performance can change as we change the number of cores or number of threads of each core. In this section, we apply the developed algorithms to different architectures. We then plot the results and do the comparison.

We organize the architectures into three catalogs. In the first catalog, we fix the number of cores and change the number of threads in each core. In the second catalog, we fix the number of threads in each core and change the number of cores. In the third catalog, we fix the total number of threads and change the number of cores and the number threads in each core at the same time.

In this first catalog, the number of cores is 8. The number of threads in each core changes from 4 to 64. In the second catalog, the number of threads in each core is 8 and the number of cores changes from 4 to 64. In the third catalog, the total number of threads is 512. The architectures include 4x128, 8x64, 16x32, 32x16, 64x8, 128x4 with form AxB where A is the number of cores and B is the number of threads in each core. The results are shown in the figure 5.20, figure 5.21,figure 5.22. From the figures, we can see that in catalog 1, as the number of threads in each micro-engine increases, the average inter-processor communication cost decreases. This is because more nodes are processed in the same core so less inter-processor communication is required. On the other hand, the average processor utilization increases as the number total number threads available increases except for UDFS without duplication where total number of node is limited to 25. In catalog 2, the average utilization follows the same trend. As for inter-processor communication cost, because the number of threads in each core is fixed and the number of cores increase the inter-processor communication cost increase. This is because more tasks are distributed to different cores and more communication between tasks are required. In catalog 3, we can clearly see that because we fix the total number of threads, so when the number of threads in each core decreases, the inter-processor communication cost increases. As for utilization, except for UDFS without duplication algorithm, others maintain the consistent average processor utilization. This is because the total number of threads is fixed.

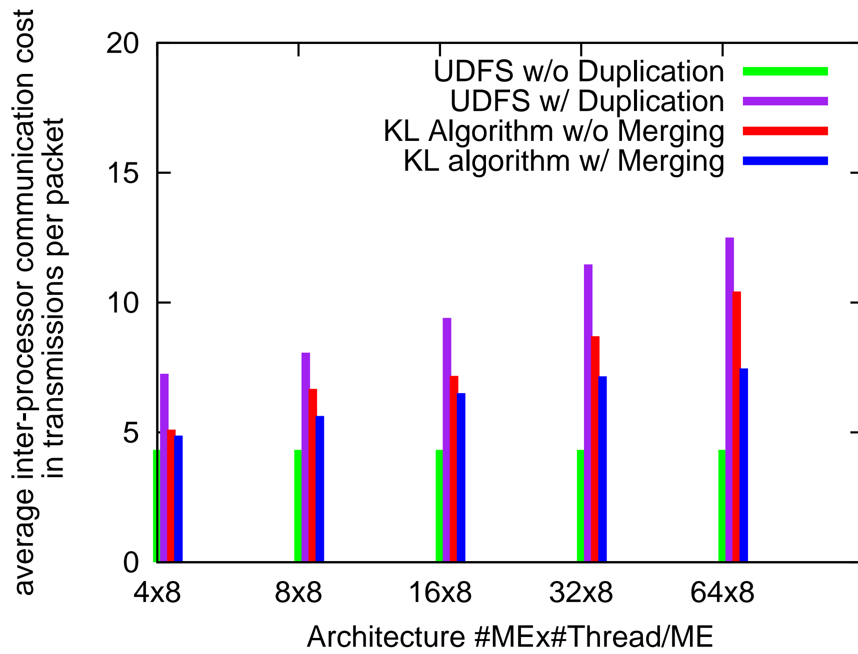


(a) average inter-processor communication cost

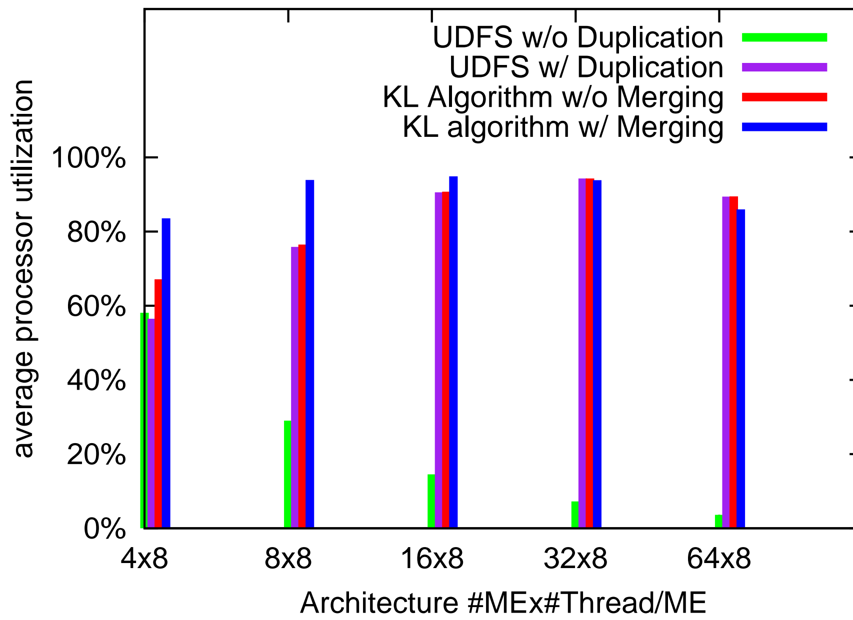


(b) average processor utilization

Figure 5.20. Architecture Exploration: Catalog 1

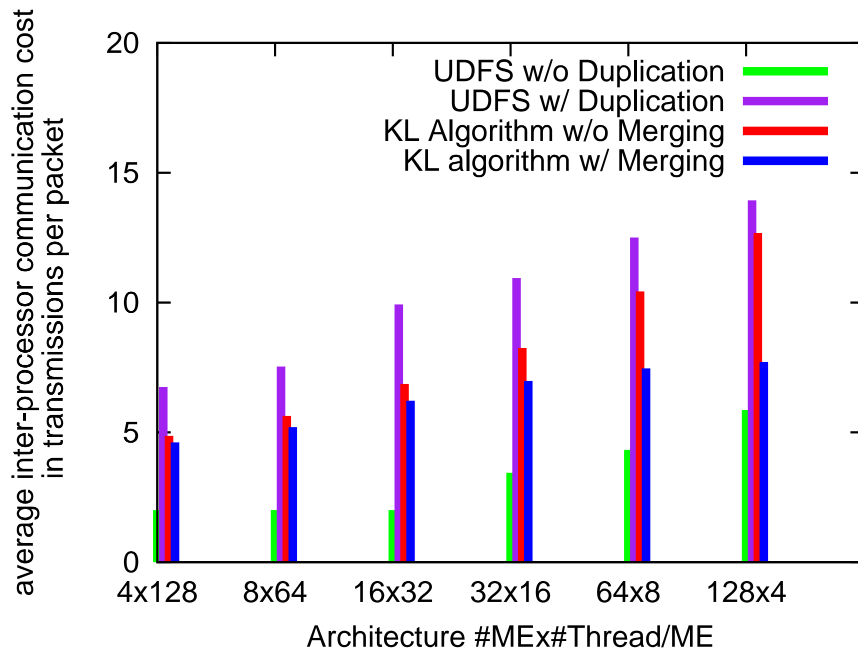


(a) average inter-processor communication cost

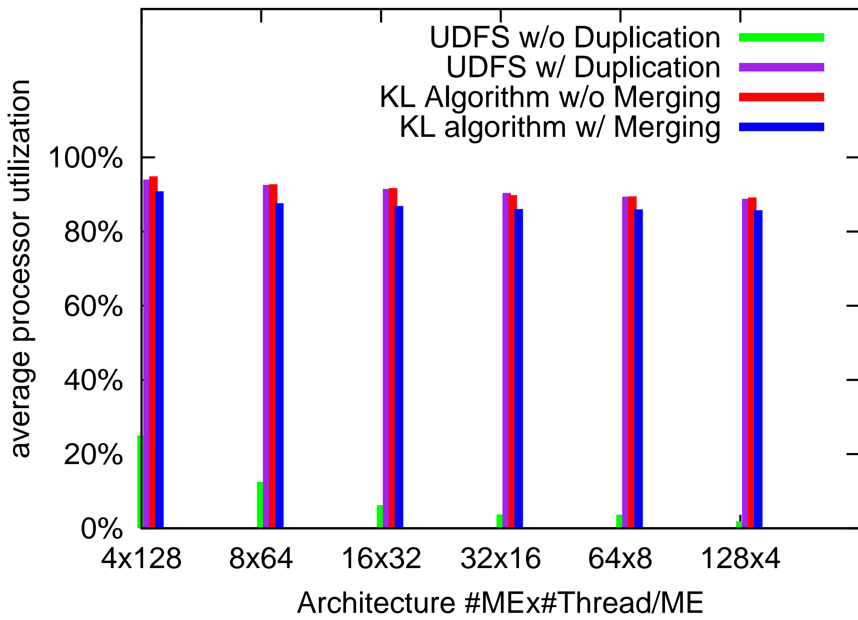


(b) average processor utilization

Figure 5.21. Architecture Exploration: Catalog 2



(a) average inter-processor communication cost



(b) average processor utilization

Figure 5.22. Architecture Exploration: Catalog 3

CHAPTER 6

IMPLEMENTATION CONSIDERATIONS ON INTEL IXP SYSTEM

Our model is designed for multi-core,packet processing systems especially network processors. In this chapter, we discuss the architecture of the latest Intel network processor. Then based on the architecture, we introduce the mapping between elements in our model and the real hardware. Finally, we discuss the applicability and limitation of task mapping model on real network processor systems.

6.1 System Architecture

Intel's IXP2xxx series network processors are chip multi-core processors. The data path architecture is shown in Figure 6.1. In the figure, we omit the control processor - an Intel Xscale core and two media interfaces. In the figure, we can see that IXP2400 network processor has eight integrated programmable microengines. Each one of them has 4K instruction stores. Microengines are connected sequentially by next neighbor registers. The connections are annotated in the figure by red lines. Next neighbor registers are fast paths for microengines to talk to their neighbors. The IXP2400 network processor also has one memory interface for DDR DRAM, two interfaces for QDR SRAM and one on-chip 16-K byte scratchpad memory. These three types of memory are shared among all of the microengines.

6.2 Model Implementation

From the previous chapters, we know that there are two different types of elements in our model: tasks and edges. When the model is to be implemented on a real network

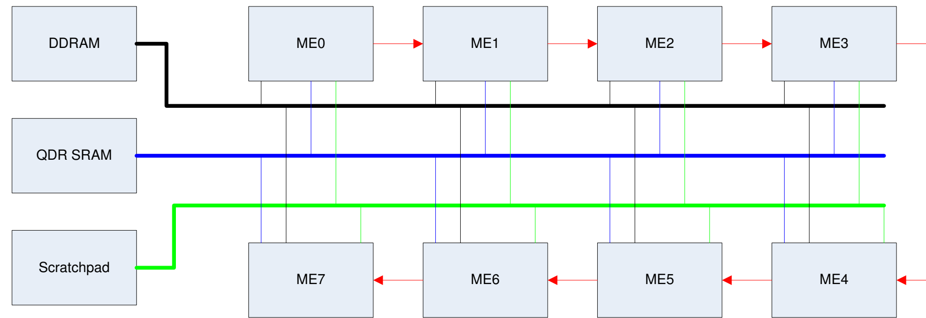


Figure 6.1. IXP 2400 network processor data path architecture

processor system, there are certain constraints on the mapping relation between each type of element and the hardware resource. In this section, we discuss how hardware resources relate to elements in our model. First we give an overview on microengines and hardware threading, and introduce how tasks are mapped to hardware threads. Then, we focus on memory subsystem, and how edges are mapped to each type of memory.

6.2.1 Processing Units

A task is a collection of instructions that perform certain processing step in a network application. Therefore, execution of tasks must be performed on processing units. On IXP 2xxx series network processors, the execution units are hardware threads in each microengine. Each hardware thread has its own set of registers, thus microengine can switch between hardware threads without additional overhead. Usually, the events that trigger thread switches are waiting on external events such as memory reference and signal handling. By switching threads, network processors can hide the idle cycles spent on waiting, therefore increasing the speed of overall application execution.

However, the number of hardware threads in one microengine is limited to eight on IXP 2400 due to the fact that registers are expensive in chip design. Since it is not possible to run more than one task in the same hardware thread,in a potential implementation on a network processor, we assume that each hardware thread can host at most one task.Furthermore,

network processors have a limited amount of local instruction store size, which posts a restriction on the total amount of instructions that could be mapped on one microengine.

6.2.2 Inter-processor Communication

To achieve communication between tasks, it is important to consider inter-processor communication in our model. In the IXP 2400 network processor system, there are two types of communication mechanism. One is based on shared memory and the other is based on next neighbor registers.

- Memory based inter-processor communication. IXP 2400 network processor system has two types of memory: SRAM and DRAM. A large amount of slower DRAM is used to provide low-cost storage capacity, while a small size of fast SRAM is used to reduce execution time by hosting frequently used data. This SRAM is available in the form of scratchpad SRAM and QDR SRM. In IXP 2400, scratchpad SRAM is for parameters and inter-processor communications. QDR SRAM is for packet queue storage and lookup table while DDR DRAM is for packet payload storage. All these memories are shared among all microengines via shared bus, especially can be accessed directly by each hardware thread. With the shared memory, threads on microengines can communicate with each other by storing messages in the memory and inform the receivers to collect the data by accessing the same storage location.

- Next neighbor register based inter-processor communication. Besides shared memories, IXP 2400 network processors also include next neighbor registers to facilitate the communication between two adjacent microengines. The topology of the connection is shown in figure 6.1. We can see that microengines are connected in a sequential manner. Only adjacent microengines can communicate directly with their next neighbors.

Although many techniques on general purpose computing systems (e.g, hardware threading and shared memories) are adopted in network processors, cache is not a widely accepted

practice. The efficiency of cache is determined by locality in data. Unfortunately, content of packet, which is the biggest portion of data on network processor systems, can not be predicted. Packet processing systems have to examine every new packet in order to do further processing. Thus, the use of cache is limited to some specific applications such as IP table lookup etc. For this reason, cache is often omitted in network processor design. IXP 2400 does not have any cache. Therefore, we ignore cache in our discussion.

6.3 Applicability and Limitation of Task Mapping Model

Our work is a conceptual model for general multi-core packet processing systems. When it is adapted to a real network processor system like the IXP 2400, the actual performance could be different from the result predicted by the model. In this section, we discuss the applicability and limitation of our model on real systems.

The system parameters used in the evaluation in previous chapter is directly related to the IXP2400 architecture. In our model, we have 8 cores and each core can accommodate 8 threads. This is the same as the IXP 2400 network processor system which has 8 micro-engines and each one has 8 hardware threads. Thus, our model can reasonably reflect the real system processor utilization.

As for inter-processor communication model, we have two possible situations. In our model, we assume that each processor can communicate with each other. So if inter-processor communication is implemented using shared memory, our model reflect model the real situation except that our model does not consider the timing issue related to memory access. That is, in our model, we consider the communication cost as the amount of data transferred between processors without considering resource contention and delay when multiple processors want to access the same memory. Therefore, our simulation results may not be exactly accurate,but can still be used to compare different algorithms.

If the inter-processor communication is implemented using next neighbor registers, then our model is not suitable for this implementation. The reason is that in next neighbor

register based communication, only adjacent processors can communication directly via next neighbor registers. Our assumption that any two processors can communicate with each other is not satisfied. Also our mapping algorithm tries to find the mapping with the minimum total inter-processor communication cost and does not consider communication between any two processors. Thus, it is possible that two processors may generate a large amount of communication that exceeds the available bandwidth of one such point to point communication link. For such a scenario a new mapping algorithm would need to be developed.

CHAPTER 7

CONCLUSIONS

In this work, we have explored different task mapping algorithms for multi-core, packet processing systems. we also implemented these algorithms and compared the results of the algorithms.

We first reviewed the previously designed algorithms which include UDFS algorithm and duplication process. We then applied the KL algorithm to our problem and were able to reduce the inter-processor communication by 20% while maintaining the similar utilization. We then modified the original KL algorithm by considering utilization during the mapping process. In this extended KL algorithm, we incorporated a tradeoff factor α to tradeoff between inter-processor communication and processor utilization. The best α is different for different system configurations in terms of communication bandwidth and computing power. Simulated annealing(SA) algorithm was then implemented. The parameters for SA algorithm were decided by following literatures or by doing experiments. Results from SA algorithm shows that it can produce decent results that are comparable to KL algorithm. In order to further improve the utilization, merging operation was applied to the task graph before mapping algorithms were applied. The mapping results showed that merging is a good way to improve the utilization and at the same time keep the communication cost lower. Finally, we applied the mapping algorithms to different packet processing system architectures. The results show how inter-processor communication cost and processor utilization change as system architecture changes.

BIBLIOGRAPHY

[1] Asymmetric multiprocessing. `http://en.wikipedia.org/wiki/Asymmetric_multiprocessing`.

[2] Message passing interface. `http://en.wikipedia.org/wiki/Message_Passing_Interface`.

[3] Multi-core computing. `http://en.wikipedia.org/wiki/Multi-core`.

[4] Openmp. `http://openmp.org`.

[5] A. Tengg, A. Klausner, B. Rinner. Task allocation in distributed embedded systems by genetic programming. In *8th International Conference on parallel and distributed computing, application and technologies* (2007).

[6] Ananth Grama, George Karypis, Vipin Kumar Anshul Gupta. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[7] Berman Fran, Anthony Hey, Geoffrey Fox. *Grid Computing: Making The Global Infrastructure a Reality*. Wiley, March 2003.

[8] Bokhari, S.H. A shortest tree algorithm for optimal assignments across space and time in a distributed processor systems. *IEEE transaction on software engineering* (November 1981).

[9] Butenhof, David R. *Programming with POSIX Threads*. Addison-Wesley.

[10] C.C. Shen, W.H. Tsai. A graph matching approach to optimal task assignment in distributed computing system using a minimax criterion. *IEEE transaction on computer* (March 1985).

[11] D.A Bader, R. Pennington. Cluster computing: Applications. *The International Journal of High Performance Computing 15*, 2 (May 2001), 181–185.

[12] David E. Culler, Jaswinder Pal Singh, Anoop Gupta. *Parallel Computer Architecture - A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[13] Eatherton, W. The push of network processing to the top of the pyramid. In *Keynote Presentation at ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)* (Princeton, NJ, Oct. 2005).

[14] E.Kohler. The click modular router. *ACM transaction on Computer System 18*, 3 (August 2000).

[15] G.S. Almasi, A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings publishers, Redwood city, CA, 1989.

[16] Hillis, W. Daniel, Steele Guy L. Data parallel algorithms. *Communications of the ACM* (December 1986).

[17] Husak, Dave. The c-5 digital communication processor. *HotChips* (2000).

[18] Intel Corporation. *Intel IXP2400 Network Processor*.

[19] J, Quinn Michael. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Inc., 2004.

[20] Jacek Radajewski, Douglas Eadline. Beowulf howto. Linux documentation project, November 1998.

[21] John L. Hennessy, David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers.

[22] John L. Hennessy, David A. Patterson. *Symmetric multiprocessing*. Morgan Kaufmann, September 2006.

[23] Kohler, Mark. *NP complete*. Embedded Systems Programming, November 2000, pp. 45–60.

[24] Lekkas, Panos C. *Network Processors : Architectures, Protocols and Platforms*. McGraw-Hill Professional, 2003.

[25] Lo, 7. V. M. Heuristic algorithms for task assignment in distributed systems. *IEEE transaction on computer*, 11 (November 1988).

[26] Mallik, A., and Memik, G. Automated task distribution in multicore network processors using statistical analysis. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)* (Orlando, FL, Dec. 2007).

[27] N. Fisher, J.H. Anderson, S. Baruah. Task partitioning upon memory-constrained multiprocessors. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (August 2005).

[28] Ramaswamy, R., and Wolf, T. 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), pp. 42–50.

[29] Ramaswamy, Ramaswamy, Weng, Ning, and Wolf, Tilman. Application analysis and resource mapping for heterogeneous network processor architectures. In *Network Processor Design: Issues and Practices, Volume 3*, Mark A. Franklin, Patrick Crowley, Haldun Hadimioglu, and Peter Z. Onufryk, Eds. Morgan Kaufmann Publishers, Feb. 2005, ch. 13, pp. 277–306.

[30] Robert Ennals, Richard Sharp, Alan Mycroft. *Task Partitioning for Multi-core Network Processors*. Springer Berlin / Heidelberg, 2005.

[31] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. Optimization by simulated annealing. *Science 220*, 4598 (1983).

[32] Stone, H.S. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE transaction on software engineering*, 1 (January 1977).

[33] Tessier, Russell. *Fast Place and Route Approaches for FPGAs*. PhD thesis, Massachusetts Institute of Technology, 1992.

[34] V.B.Gylys, J.A.Edwards. Optimal partitioning of workload for distributed systems. *Digest of Papers, COMPCON* (Sept. 1976).

[35] W.Plishker. Automated task allocation for network processors. In *Proc. Network System Design Conf.* (2004).

[36] W.W.Chu. Optimal file allocation in a multiple computing system. *IEEE transaction on computer*, 10 (Oct. 1969).