ILP FORMULATION FOR DYNAMIC OFFLOADING LAYOUTS

A thesis submitted in fulfillment
of the requirements for the degree of
Master of Science

by

Ola Adamovsky

Supervised by

Prof. Danny Dolev

School of Engineering and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

September 2007

Acknowledgments

First, I would like to deeply thank my advisor Prof. Danny Dolev, for giving me the opportunity to participate in this fascinating research.

Special thanks go to Yaron Weinsberg, my co-advisor, for his efforts and support. I really enjoyed working with Yaron. I thank Yaron for being a huge source of inspiration, for carefully reviewing my drafts, providing precious advice, and helping to improve this thesis with his original ideas and insights.

Next, I would like to thank Dr. Tal Anker, whose ideas, advice, and enthusiasm for this work helped to shape it.

I would also like to thank Ittai Abraham for his huge help in understanding the mathematical model that provided this work the mathematical closure.

Special thanks to my husband's parents, Irina and Alexander Grabarnik, for constantly boosting my morale.

Last, but not least, I would like to thank my husband Maxim for his patience during my studies.

Abstract

As the race for faster and stronger CPUs is drawing to a close, it has become almost impossible to significantly increase computer system performance solely by improving the physical characteristics of the hardware without negatively impacting form factor or power consumption. Optimization approaches are therefore gaining more attention, since they provide an alternative way to increase system performance.

One way to increase system performance is to spread the computational payload between the system's several CPUs, instead of trying to improve the main CPU physical characteristics. Every commercially available PC contains a wealth of untapped computing resources: the network controller, the disk controller, graphics adapters and many other peripheral devices have their own CPUs, with power which sometimes compares with or exceeds that of their host CPU (as is the case with high-end graphics adapters). This "hidden" CPU power is usually available only to a specific group of dedicated applications, (henceforth referred to as "offload-aware applications"), which are granted exclusive access to this computational resource, and may offload some of their tasks to these peripheral devices, instead of letting the user or the system designer decide how the system resources should be distributed.

Our research utilizes a model in which offload-aware applications can use the computational resources of peripheral devices and share them with other applications. This allows,

non-offload-aware applications access to services which allow them to offload tasks to the dedicated devices, and thus release the host CPU for use by other activities. Arguably, the decision about what tasks should be offloaded to where, should be left to the programmer. However, once there are several applications in the system and several devices to use, too many criteria exist for human to make this decision unaided.

This thesis presents a solution to this decision-making process . We describe a framework that allows an optimal offloading layout to be achieved for a given set of applications. Optimization criteria and offloading constraints are provided by the programmer, along with the application set definition. This framework has been deployed within the HYDRA framework ([WDWAa]) as a tool to facilitate its offloading aspects.

Table of Contents

Acknowledgments **2**

Abstract **3**

1 Related Work **10**

1.1 Use of ILP technique to solve optimization problems 10

1.1.1 Software pipelining optimizations 10

1.1.2 Low-power scheduling 11

1.2 Use of other techniques to solve optimization problems 11

1.2.1 Simulated annealing 11

1.2.2 Genetic algorithms and hill climbing technique 12

1.3 Related frameworks 13

1.3.1 FlowOS 13

1.3.2 FarGo and FarGo-DA 14

1.4 Summary 14

2 Introduction **15**

2.1 HYDRA overview 17

2.1.1 Offcode manifesto 18

2.1.2 Constraints 19
2.1.3 Device mapping 20
2.1.4 Offcode URL 21
2.2 Multi-user environments 22

3 Mathematical Model 24
3.1 ILP overview 24
3.1.1 ILP definition 24
3.1.2 Problem example 25
3.1.3 Problem example - ILP modeling 25
3.2 ILP formulation for dynamic offloading layout 29
3.2.1 Problem domain definitions 29
3.2.2 Constraints definitions 31
3.2.3 Optimization objectives 32

4 Software Architecture for ILP Compiler 34
4.1 HYDRA SW architecture 34
4.1.1 HYDRA components 35
4.2 ILP compiler architecture 36
4.2.1 ILP compiler components 37

5 Evaluation 44
5.1 AMPL ILP solver 44
5.2 Experimental application 45
5.2.1 System ILP formulation 48
5.3 Example set 1 50
5.3.1 Example 1 50

5.3.2 Example 2 53
5.3.3 Example 3 54
5.4 Example set 2 55
5.4.1 Example 1 55
5.4.2 Example 2 56
5.4.3 Example 3 56

6 Conclusions and Future Work **58**

Bibliography **60**

List of Figures

2.1 ODF - Part I 19
2.2 ODF - Part II 19
2.3 Offcodes' Constraints 20
2.4 ODF - Part III 21
2.5 Offcode URL format 22

3.1 Giapetto's Feasible Region 27

4.1 System Architecture 35
4.2 System Architecture 37
4.3 Device Enumerator 38
4.4 Offcode Enumerator 39
4.5 Offcode Matching Database 40
4.6 ILP Subsystem 42

5.1 Application Data Flow 46
5.2 Mandatory Constraints 49
5.3 AMPL Mandatory Constraints 49
5.4 Offcode Constraints Graph 51

5.5 AMPL Gang Constraints 52
5.6 AMPL Maximize Download 52
5.7 Example 1 Solution 52
5.8 Example 1.1 AMPL solution 52
5.9 AMPL Pull Constraint 53
5.10 Example 1.2 Solution 53
5.11 AMPL, Yet Another Gang Constraint 54
5.12 Example 1.3 Solution 54
5.13 Bandwidth Goal Function 56
5.14 Example 2.2 Solution 56

Chapter 1

Related Work

Integer Linear Programming techniques has been used to solve various scheduling problems in the past. This section describes state-of-the-art scheduling research, ordered by its relevance to this work.

1.1 Use of ILP technique to solve optimization problems

1.1.1 Software pipelining optimizations

The paper [GAG94] addresses the SW pipelining problem, specifically, constructing a software pipelining schedule, given a specific set of processor resources, and maximizing a running rate, while minimizing the number of resources used. Similar to our research, the authors of the [GAG94] used the Integer Liner Programming technique to create a mathematical model to solve the scheduling optimization problem. They also came to the conclusion that the ILP technique gives a simple and yet resultful solution, which in their case gives a better performance than other scheduling algorithms for SW pipelining; in our research, we used greedy approach for comparative analysis.

1.1.2 Low-power scheduling

The [SC00] presents an ILP-based scheme for high-level synthesis for low-power applications. In addition, the authors of the [SC00] present a LP-based model for resource binding that minimizes the amount of switching at the input of the functional units. Similar to the bandwidth constraints in our work, the [SC00] uses user-defined weighting factors to define the relative importance of different objective functions. Although our solution currently allows only one objective function to be used at a time, it might be a fruitful idea to apply this approach in the future, to allow the user to chose several goal functions at a time and to determine their relative weight.

Although the ILP-solving SW we are using today does not allow it, the architecture proposed by this thesis allows the underlying engines to be easily changed, without harming the rest of the model.

1.2 Use of other techniques to solve optimization problems

1.2.1 Simulated annealing

Simulated annealing (SA) is a generic probabilistic meta-algorithm for the global optimization problem, namely locating a good approximation for the global optimum of a given function in a large search space. It is often used when the search space is discrete. In favorable cases, simulated annealing may be more effective than exhaustive enumeration of the search space.

Each step of the SA algorithm replaces the current solution with a random “nearby”

solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter T (called the temperature) that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when T is large but increasingly goes "downhill" as T goes to zero. The allowance for "uphill" moves saves the method from becoming stuck at local minima, which are the bane of greedier methods.

The method was described by S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi in 1983 [KGV83]. The method is an adaptation of the Metropolis-Hastings algorithm, a Monte Carlo method that generates sample states of a thermodynamic system, invented by N. Metropolis et al in 1953. In the simulated annealing method, each point of the search space is analogous to a state of some physical system, and the function E(s) to be minimized is analogous to the internal energy of the system in that state. The goal is to bring the system from an arbitrary initial state, to a state with the minimum possible energy.

This method was used, for example, to find a minimum makespan in a job shop, [vLAL92]. By using SA, their algorithm has been proved to find shorter makespans than other approximation approaches used in the same time, though at the cost of longer running times.

1.2.2 Genetic algorithms and hill climbing technique

A genetic algorithm is a search technique used in computing to find exact or approximate solutions to optimization and search problems. Genetic algorithms are categorized as global search heuristics and are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover.

Hill climbing is an optimization technique that belongs to the family of local search. It

is a relatively simple technique to implement, making it a popular first choice. Although more advanced algorithms may give better results, there are situations where hill climbing works well. Hill climbing can be used to solve problems that have many solutions but where some solutions are better than others. The algorithm is started with a random (and thus probably a bad) solution to the problem. The algorithm sequentially makes small changes to the solution, each time improving it a little bit. At some point, the algorithm arrives at a point where it cannot see any improvement anymore, at which point the algorithm terminates. Ideally, at that point a solution is found that is close to optimal, but it is not guaranteed that hill climbing will ever come close to the optimal solution.

Both genetic algorithms and hill climbing techniques do not operate on dynamic data sets, as genomes begin to converge early on towards solutions that may no longer be valid for later data. However, based on those techniques, [YM93] shows a dynamic hill climbing technique, which overcomes this obstacle. "Furthermore," the article says, "the algorithm moves from a coarse-grained search to a fine-grained search of the function space by changing its mutation rate and uses a diversity-based distance metric to ensure that is searches new regions of the space."

Moshe Sidi and Reuven Elbaum, [ES96], on the other hand, use genetic algorithms to design local area networks with the objective of minimizing the average network delay.

1.3 Related frameworks

1.3.1 FlowOS

FlowOS [BK03] proposes an architecture that removes the host's memory subsystem and CPU from the critical data path. The main role of the OS is to manage the data flow

between different peripheral devices and to schedule the flows between different applications. Although FlowOS does not provide an offloading framework or a programming model for creating offload-aware applications, the proposed flow abstraction can further extend this research. By defining a "flow" overlay that spans several offloaded applications, one can guarantee the required QoS for a specific application.

1.3.2 FarGo and FarGo-DA

Although not dealing with offloading, FarGo [HBSG99b, HBSG99a] and FarGo-DA [WBS02] propose a programming model that enables a developer to program relocation and disconnection semantics in a separate phase during the application development cycle. The basic assumption for their work is that the application is fully composed of a set of components that are tagged by a specific interface (called: *Complet*). The components are hosted in a virtual machine and can migrate to a remote VM using marshaling and unmarshaling mechanisms (much as in the RPC [Sri95, BN84], RMI [Sun98, rmi99], CORBA [Sie98], DCOM [BK98], or WebService [Org] models). Our framework extends the above model by defining an "offloading-layout" that is used to define the offloading aspects of the application.

1.4 Summary

In this chapter, we surveyed the existing body of work in the area of different optimization techniques, specifically, but not limited to, ILP used in this thesis. We also viewed ILP formulations for solving various kinds of scheduling problems. Development of a special mathematical model and examination of the specific system support that is required for realizing such a model are the goals of this research.

Chapter 2

Introduction

Many computer peripheral devices are software driven: they are, in fact, programmable devices with their own memory and a CPU, programmed to perform tasks set by the designer. The choice of the task set is usually dictated by the device category: for example, network controllers send and receive packets, and disk controllers read and write blocks of data.

Programmable peripheral devices usually execute some kind of control software; however, richer software can also be embedded into some devices. A famous example is the TCP Offload Engines (TOEs) [Cur04], which has been built into Network Interface Cards (NICs) firmware. This is a classic example of "host functionality" being statically offloaded into a peripheral device to achieve better system performance, lower latency, and to decrease host CPU utilization.

There have been other attempts to offload traditionally-host-performed functionality onto peripheral devices. Quite a few such attempts have been made in the context of distributed communication algorithms, particularly in cluster network environments. An example of this is fast barrier software using programmable NICs, [BPS01]; other examples

include broadcast/multicast distribution speedup, [BPDS00], and a message passing interface (MPI) with a rich set of performance enhancements achieved by offloading the MPI to the NIC [ZKW02]. There have also been similar attempts in more consumer-oriented areas, such as the acceleration of multimedia applications using intelligent peripherals [FMOB98].

All the above-mentioned examples are implementations of static offloading, in which an optimization is hard-coded into a peripheral's firmware in order to achieve the highest possible performance gain. While such approaches have proved successful in improving overall system performance, they suffer from a few drawbacks. The first and most immediate one is that such an optimization is a complex task requiring the involvement of embedded system experts, who would be aware of the intricacies and potential complications involved. A second drawback is that features hard-coded into the peripheral firmware are inflexible: each offloaded optimization is usually specifically designed for a particular application. But resources available on peripherals are very scarce; hence, only a limited number of applications may be compiled into the firmware at any given time.

Dynamic offloading tries to address flexibility and development complexity issues while preserving the benefits of the static approach. To make this possible, the peripheral devices' software is designed so as to allow future extensibility.

Several design choices have been studied for dynamic offloading. One of them is the Virtual Machine-based firmware. This was used in Myrinet clusters, research [WJPR04], to provide grounds for optimizing several MPI primitives. The other design choice involves basing the firmware on an operating system (OS) specially tailored for run-time configurability (usually using an academic OS). Both approaches improve the application's optimization potential to some extent, yet they come at a cost: virtual machines trade off performance for configurability, and OSs tend to have rather complicated loaders with non-negligible footprints [BMW03]. Furthermore, both design choices involve

aspects which lie far from industry common practices.

Existing systems which deploy dynamic offloading are not considerably more flexible than static offloading-based systems. While they do target a class of applications (as opposed to static-offloading systems which target a single specific application), the class of applications that they target is very narrow. This is in addition to the fact that dynamic-offloading systems are, in practice, very inflexible in peripheral choice and setup configuration options (as we have seen in the previous paragraph.)

In the past, peripheral devices provided minimal functionality, and left the rest to the host. This is no longer the case: trends in digital design have followed an exponential increase in the number of transistors on an integrated circuit. This ongoing trend of decreasing cost and increasing density of transistors has motivated hardware and embedded system designers to use programmable solutions in their products. The proliferation of *programmable* peripheral devices for personal computers opens up new possibilities for academic research that will influence system designs in the near future.

Programmability is a key feature that enables application-specific extensions to improve performance and offer new features for their respective applications. Increasing transistor density and decreasing cost allow for surplus computational power in devices such as disk controllers, network interface cards, video cards, and more. Such designs are cheaper and more flexible than custom ASIC solutions. The performance capabilities of programmable devices, and microprocessors in particular, will extend well into the range of applications that formerly required DSPs or custom hardware designs.

2.1 HYDRA overview

HYDRA, fully described in [WDWAb, WDWAa, Yar07], has proposed a novel offloading framework that enables utilization of various peripheral devices. The motivation for such

a framework becomes clearer as peripheral devices become powerful and programmable.

In every modern PC, there is a wealth of unused computing resources. For example, the NIC has a CPU, the disk controller is programmable and some high-end graphics adapters are already more powerful than host CPUs.

The HYDRA framework enables an application developer to design the offloading aspects of the application by specifying an "offloading layout," which is enforced by the runtime during application deployment. This framework defines a model in which applications execute cooperatively and concurrently in host processors and in device peripherals. In HYDRA model, applications can *offload* specific tasks to devices to improve the overall application's performance. The HYDRA framework also provides the necessary abstractions, programming constructs, and development tools to develop such applications.

This section provides a short overview of the HYDRA framework. For brevity, we present only the basic abstractions provided by the framework. The interested reader is advised to read [Yar07].

2.1.1 Offcode manifesto

An Offcode manifesto is the means by which an Offcode (the peace of SW being offloaded) defines its dependencies on peer Offcodes and its requirements from the target device and software environment.

The manifesto is realized in an Offcode Description File (ODF). An ODF contains three parts: the first part describes the structure of the Offcode's package, containing the binding name of the Offcode at the target device, and the Offcode's supported interfaces. The Offcode's interfaces are typically described by a standard WSDL [wo] file. Figure 2.1 presents a typical import section defined in an Offcode's ODF.

The binding name identifies the Offcode at the target device and is used in the various

```
<ocode>
 <!-- ocode package info -->
 <package>
  <bindname>Hydra.net.utils.Socket</bindname>
  <GUID>7070714</GUID>

  <interface>
    <!-- WSDL interface specification >
    <include>"/offcodes/socket.wsdl"</include>
  </interface>
 </package>
```

Figure 2.1: ODF - Part I

HYDRA APIs to identify the Offcode.

```
 <!-- ocode dependencies -->
 <sw-env>
   <import>
     <file>"/offcodes/checksum.odf"</file>
     <bindname>Hydra.net.utils.Checksum</bindname>
     <reference type="Pull" pri="0"></reference>
     <GUID>6060843</GUID>
   </import>
 </sw-env>
```

Figure 2.2: ODF - Part II

The second part of an ODF describes the Offcode's dependencies on peer Offcodes. This section enables a developer to "design" the offloading process that will occur at deployment time.

2.1.2 Constraints

HYDRA provides several constraints presented in Figure 2.3 that can be used between any two Offcodes denoted by α and β. The set of Offcodes and related constraints form an

Offloading Layout Graph, with Offcodes as nodes and constraints as edges. The runtime (recursively) processes an Offcode's ODF file to produce such a graph, which is later used by the runtime to decide on the actual placement of Offcodes.

- ***Link***: The Link constraint is denoted as $\alpha \overset{Link}{\Leftrightarrow} \beta$. This is the default constraint from α to β, which actually poses no constraints: α and β may or may not be mutually offloaded (to the same or different target device). It does, however, indicate that at least one of the Offcodes needs the other to function.
- ***Pull***: The Pull constraint is denoted as $\alpha \overset{Pull}{\Leftrightarrow} \beta$. This reference is used to ensure that both Offcodes will be offloaded to the **same** target device.
- ***Gang***: The gang constraint is denoted as $\alpha \overset{Gang}{\Leftrightarrow} \beta$. This constraint is used to ensure that both Offcodes will be offloaded to **their respective** target devices. That is, if α is offloaded, β will be too, albeit on perhaps a different device.
- ***Asymmetric Gang***: This constraint is denoted as $\alpha \overset{\sim Gang}{\rightarrow} \beta$ and provides the asymmetric version of Gang. Offloading β doesn't implies offloading α.

Figure 2.3: Offcodes' Constraints

Note that there is no *Asymmetric Pull* constraint as the motivation for using *Pull* is a tight interaction between two Offcodes. Enabling asymmetry may result in the placement of two Offcodes in two different execution domains, which we would want to prevent. Figure 2.2 presents the mechanism by which a constraint is set on an Offcode reference. In this example, a *Pull* constraint is set for the peer Offcode denoted by: "Hydra.net.utils.Checksum."

2.1.3 Device mapping

The last part of the ODF is concerned with device mapping. To enable dynamic mapping between Offcodes and peripheral devices, on different host configurations, a developer is

required to supply a list of potential target *device classes* that can be used for offloading.

```
<!-- device classes -->
<targets>
  <device-class id=0x0001>
    <type>NIC</type>
    <mac>ethernet</mac>
    <bus>pci</bus>   <!-- (optional) -->
    <rate>1000</rate> <!-- (Mbps) -->
    <vendor>3COM</vendor> <!-- (optional) -->
  </device-class>

  <device-class id=0x0002>
    <type>NIC</type>
    <mac>myrinet</mac>
    <rate>10000</rate> <!-- (Mbps) -->
  </device-class>
</targets>
</offcode>
```

Figure 2.4: ODF - Part III

Figure 2.4 shows a sample Offcode for which the developer has indicated the classes of potential devices on which it can operate. It is the runtime's responsibility to locate an instance of such an Offcode that will comply with: a) being suitable for running at one of the local devices and b) being in one of the listed classes. Alternatively, the developer can specify the exact target for each Offcode using an Offcode's URL.

2.1.4 Offcode URL

An Offcode is uniquely identified by an Offcode URL. The URL consists of four parts: the host, the device's physical address, the hardware identifier, and the Offcode's binding name that is unique per device. The physical address and the hardware identifier uniquely identify the target device. Figure 2.5 presents an Offcode's URL format and a sample URL

for some PCI device. A PCI device is physically addressable by a bus number (8 bits), a device number (5 bits), and a function number (3 bits). The hardware identifier is further identified by a 32-bit signature that includes the vendor identifier and the device identifier.

```
[host]:/[physical-address]/[hardware identifier]/[binding-name]

Example:
---------
The Hydra runtime \offcode\ on the Netgear GA-620T (TigonII chipset)
device is identified by the string:
   localhost:/pci/00/11/1385/620A/Hydra.Runtime
```

Figure 2.5: Offcode URL format

2.2 Multi-user environments

The strength of our proposed programming model lies in the system's ability to reuse Offcode components. Although reusability may simplify and speed up the development cycle, in multi-user environments, reusing the same Offcode in several applications may substantially complicate the offloading layout design. Intuitively, the problem of defining an optimal offloading layout graph for a group of offload-aware applications may be computationally hard.

Our research uses the Integer Linear Programming methodology (ILP) to optimize such complex layouts. ILP formulation enables expression of every offloading layout graph as a set of linear equations. Any ILP solver can then be used to solve the equations, given a target optimization function. In the following chapters we will show examples of one specific ILP solver that ran on a few specific scenarios, although the software architecture proposed in chapter 4 supports any ILP solver.

In section 3.2 we provide a mathematical presentation of an offloading layout graph, together with a few examples of criteria that could be used as target optimization functions. We also show a detailed example of how to formulate a sample offloading graph and a short description of an ILP solver used to solve those examples. Although the examples might seem rather obvious to the reader, we present them only to prove the concept.

We solve the scenarios with different devices and Offcodes. These scenarios become non obvious very fast, and it is very hard for a person to find an optimal solution using simple methods. We also argue - and it will become clear from the additional examples - that a greedy solution does not apply, not only because it is hard to satisfy all the constraints with a greedy solution, but also because a greedy solution will not be optimal if some more complex constraints are present.

The offloading layout which we use is usually statically defined or set during deployment in order to minimize the overhead costs which result from offloading operations.

The research which follows deals with defining the offloading layout, given a set of constraints, a desired goal function, and a list of devices present in the system. The exact offloading mechanism is not a part of this paper and will not be discussed here; the interested reader can refer to [Yar07].

Chapter 3

Mathematical Model

3.1 ILP overview

3.1.1 ILP definition

Linear Programming (LP) is a tool used to solve a subset of optimization problems; it focuses on minimizing or maximizing objective function $f(x_1, .., x_n)$ subject to a set of constraints. The objective function must be a linear function of x_j, while constraints are linear equilities/inequalities, involving the same decision variables as the objective function.

An LP problem is called *integer* linear programming (ILP) if all variables are required to be integers. 0-1 integer programming is a special class of ILP in which variables are required to be 0 or 1 rather than arbitrary integers.

3.1.2 Problem example

This example is taken from the Operations Research: Applications and Algorithms book, [Win].

Giapetto's Woodcarving Inc. manufactures two types of wooden toys: soldiers and trains. The manufacture of wooden soldiers and trains requires raw materials and two types of skilled labor: carpentry and finishing. The weekly amount of resources is limited as is the demand. The goal is to find the numbers of soldiers and trains that will maximize the weekly profit. All numeric quantities and assumptions about this problem are summarized below:

1. There are two types of wooden toys: soldiers and trains.
2. A soldier sells for $27, uses $10 worth of raw materials, and increases variable labor and overhead costs by $14.
3. A train sells for $21, uses $9 worth of raw materials, and increases variable labor and overhead costs by $10.
4. A soldier requires 2 hours of finishing labor and 1 hour of carpentry labor.
5. A train requires 1 hour of finishing labor and 1 hour of carpentry labor.
6. At most, 100 finishing hours and 80 carpentry hours are available weekly.
7. The weekly demand for trains is unlimited, while, at most, 40 soldiers will be sold.

3.1.3 Problem example - ILP modeling

To model a linear problem, the decision variables are established first, since they will determine the value of the objective function and, hence, the optimal solution. In Giapetto's

shop, the objective function is the profit, which is a function of the number of soldiers and trains produced each week. Therefore, the two decision variables in this problem are the following:

- x_1: Number of soldiers produced each week
- x_2: Number of trains produced each week

Once the decision variables are known, the objective function of this problem is simply the revenue minus the costs for each toy, as a function of x_1 and x_2.
$z = (27-10-14)x_1 + (21-9-10)x_2 = 3x_1 + 2x_2$
* Note that the profit depends linearly on x_1 and x_2 – this is a linear problem.

Now we need to analyze the assumptions made for this problem to formulate the constraints (or else the model is very likely to be wrong). The first three assumptions determined the decision variables and the objective function. The fourth and sixth assumptions say that finishing the soldiers requires time for carpentry and finishing.

One soldier requires 2 hours of finishing labor, and Giapetto has at most 100 hours of finishing labor per week, so he can't produce more than 50 soldiers per week. Similarly, the carpentry hours constraint makes it impossible to produce more than 80 soldiers weekly. Note here that the first constraint is stricter than the second. The first constraint is effectively a subset of the second; thus, the second constraint is redundant.

Since both soldiers and trains require finishing time, both need to be taken into account. The general constraint in terms of the decision variables is: $2x_1 + x_2 \leq 100$.

Now that the constraint for the finishing hours is ready, the carpentry hours constraint is found in the same way to be: $x_1 + x_2 \leq 80$.

According to the problem description, there can be at most 40 soldiers produced each week: $x_1 \leq 40$.

The demand for trains is unlimited, so no constraint can be written for it. The model is finished and consists of the equations:

(1) $max\ z = 3x_1 + 2x_2$ *(objective function)*

(2) $2x_1 + x_2 \leq 100$ *(finishing constraint)*

(3) $x_1 + x_2 \leq 80$ *(carpentry constraint)*

(4) $x_1 \leq 40$ *(demand for solders)*

(5) $x1 \geq 0, x_2 \geq 0$ *(sign constraints)*

Note the last constraint. It ensures that the values of the decision variables will always be positive. The problem does not state this explicitly, but it's still important (and obvious).

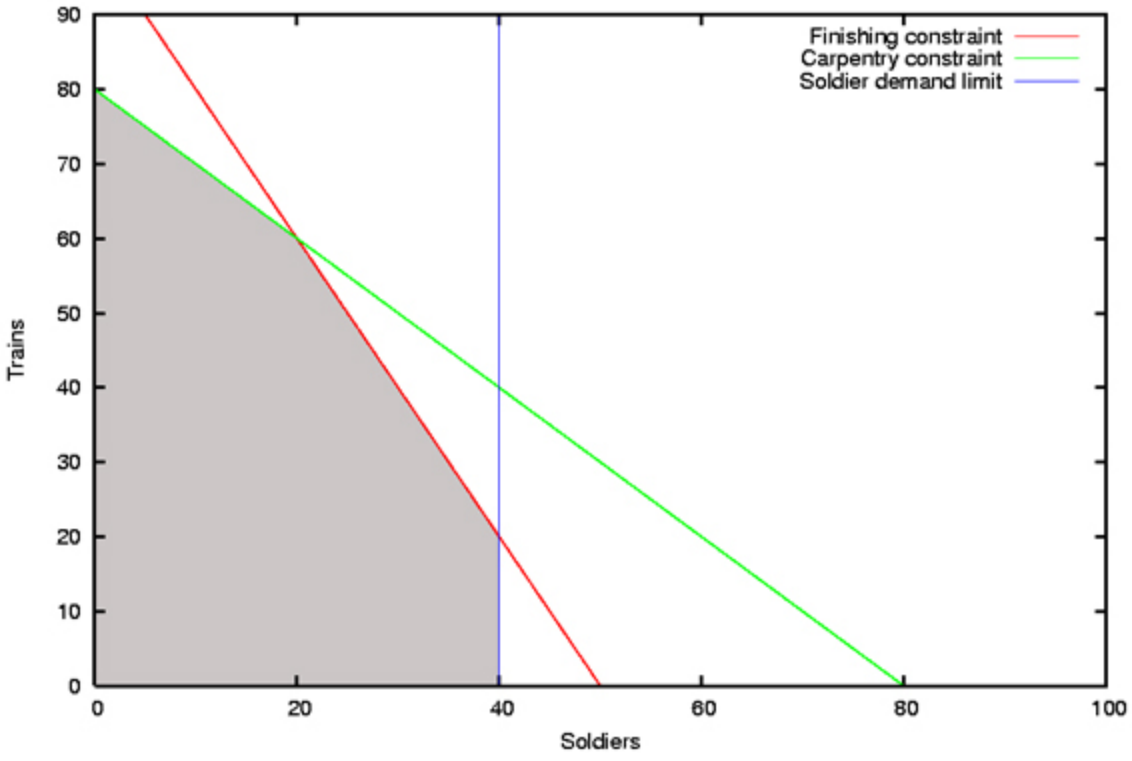

Figure 3.1: Giapetto's Feasible Region

Let's check the problem's solution space. With two decision variables, it has two dimensions. The solution space that satisfies all the constraints is called the feasible region. Figure 3.1.3 shows the feasible region for Giapetto's shop. Any (x_1, x_2) pair that falls into that region is a potential solution to the problem.

Using substitution, we can see that the (trains=60, solders=20) vertex makes the highest value of objective functions (180) among other vertexes. The optimal solution is always found on the boundary because of the continuity of the target function. So, we conclude that is the optimal solution.

Two-dimensional cases are relatively simple ones. In general, once the problem is modeled as an ILP problem, i.e., all constraints and objective functions are formalized, any ILP solver can be used to achieve the solution.

3.2 ILP formulation for dynamic offloading layout

This section provides an Integer Linear Programming methodology (ILP) for optimizing such complex layouts. The purpose of such a formulation is to enable expression of every offloading layout graph as a set of linear equations. Given a target optimization function, any ILP solver can then be used to solve the equations.

We provide the mathematical presentation of an offloading layout graph and later present, as an example, two possible criteria that could be used as target optimization functions. We have included several detailed examples of formulating a sample offloading graph. As the simple graphs are trivially easy to solve, the strength of such a formulation is only apparent in complicated scenarios. Such scenarios make the offloading layout design process significantly more difficult. In such cases, a greedy solution does not provide an optimal solution; hence, the need for such a formulation is apparent. This section provides the necessary ILP formulation to optimize the offloading layout graph.

A result for our problem will be a matrix that will provide a solution to the optimization problem described below, while satisfying all the necessary constraints.

3.2.1 Problem domain definitions

We begin by defining the basic elements of the layout graph. The layout graph $G = (V, E)$ includes the set of Offcodes as vertices, and the channel constraints among them are the edges. At deployment time, the runtime associates with each node n (Offcode) a compatibility target vector $\vec{C}_n$ representing the potential target devices that can host the Offcode. Note that the host CPUs are included in the list of devices. Let $N = |V|$ be the total number of Offcodes, and let $K = |\vec{C}|$ be the number of HYDRA-compatible devices.

1. Let K be the number of HYDRA-aware devices in the system.

2. Let $k = 0$ identify the host "device" (i.e, the host CPU).

3. Let N be the number of application Offcodes used by the given applications.

4. Let $G = (V, E)$ be an offloading layout graph - the actual result we would like to achieve.

5. Let $\vec{C}$ be a constant binary bit vector. $C_n^k = 1$ if Offcode n *can* be offloaded to device k.

   Naturally, we have a mandatory constraint here:

6. $\forall n \in N, k \in K, C_n^k \in \{0,1\}$ - Each Offcode n can be either offloaded or not offloaded to a device k.

Now, let $\vec{X}$ be the ILP output vector.

1. $X_n^k = 1$ if Offcode n *should* be offloaded to device k.

2. $\forall n, X_n^0 = 1$ iff Offcode n has *not* been offloaded - for each Offcode n, the value of an ILP output vector for the host device (device 0) will be 1 if the Offcode will not be offloaded.

   Naturally, similar to the constraint vector, we limit the value for the output vector:

3. $\forall n \in N, k \in K, X_n^k \in \{0,1\}$.

   In addition, we do not allow the same Offcode to be offloaded to more than one device:

4. $\sum_{n=1}^{N} \sum_{k=1}^{K} X_n^k \cdot C_n^k = 1$

3.2.2 Constraints definitions

To simplify the presentation, we assume that the first entry in each vector $\vec{C}$ corresponds to the host CPU. Let $E_m^n = (m,n)$ be an edge E from Offcode m to n:

1. *Link Constraint* does not really provide a mathematical constraint.

2. $\forall E_m^n \in Pull, \forall k : X_n^k = X_m^k$. The *Pull Constraint* between Offcodes m and n indicates that if both are offloaded, they should be offloaded to the same device. For example, if Offcode n is responsible for filtering packets received from the network and Offcode m is responsible for processing those packets, they should probably be offloaded to the same network device. Note, though, that this constraint doesn't present a limitation for one of them to not be offloaded at all. Hence, the layout graph is where we offload the filtering to the device, but we process the packets on the host; this graph will be perfectly legal.

3. $\forall E_m^n \in Asymmetric\ Gang : \sum_{k=1}^{K} X_n^k \leq \sum_{k=1}^{K} X_m^k$. The *Asymmetric Gang Constraint* from Offcode n to Offcode m makes sure that if n is offloaded so should m. Note that this *Does Not* mean the opposite direction. m may be downloaded alone. This constraint is useful when n uses m tightly, but not vice versa. For example, m is again an Offcode that filters packets, this time outgoing, and n is responsible for sending a specific type of packets. This is perfectly fine for m to be downloaded alone, but once n (the packet producer that needs filtering) is being downloaded, it wouldn't make sense to send the packet back to the host to be filtered.

4. $\forall E_m^n \in Symmetric\ Gang : \sum_{k=1}^{K} X_n^k = \sum_{k=1}^{K} X_m^k$. This constraint implies that both m and n always go together; similar to Siamese twins, wherever one of them goes, the other one simply must follow. For example, imagine Offcodes that use some mutual resource (actually much as Siamese twins do). It is possible to think about

two instances of an Offcode that performs streaming. They can reside on a NIC and on a "Smart Disk" device. Since we don't want packets to traverse the bus twice (which will inevitably happen when only one of the instances is offloaded), we must have a *Gang Constraint* between the instances.

These equations are sufficient to represent the joint offloading layout graph as a set of linear equations.

3.2.3 Optimization objectives

We have identified several optimization functions, two of which are presented below. The list is by no mean complete; additional objectives functions can be easily added to address various application needs. We will suggest some of those in the future work section.

1. Maximized Offloading – The trivial objective is to offload as many Offcodes as possible. The motivation for such a goal is to minimize the CPU usage and memory contention at the host: $$\max\left(\sum_{n=1}^{N}\sum_{k=1}^{K}X_n^k\right).$$

2. Utilize bus bandwidth between devices as much as you can. We assign each Offcode a "price," which is an average bus bandwidth this Offcode will use (different scales may be used here, for example use scale -10 .. 10, -10 defines a high penalty in offloading this Offcode in terms of bus utilization, 10 - this Offcode uses 100% of the bus bandwidth in average).

   In addition, we define a capability matrix of prices for each Offcode that describes the maximum bus bandwidth that might be needed by each Offcode . We will validate our solution with this matrix, after the problem has been solved.

So, let P be the price vector, where P_n defines the average for Offcode n, the objective function will be: max $(P * C)$, where C is a constraint matrix, built from the constraint vectors described above.

We will present a detailed example that uses everything that we have described in the evaluation chapter.

Chapter 4

Software Architecture for ILP Compiler

In this chapter, we present the detailed design of the application, whose inputs are the devices, Offcodes, and constraints we talked about in the previous chapters, and whose output is the offloading layout graph that tells the runtime system which Offcodes actually should be offloaded.

4.1 HYDRA SW architecture

The system implements the model and provides facilities for programming, testing, deploying, and managing OA-applications and Offcodes. Both the host OS and the target device firmware must support the interfaces defined by the programming API and implement the runtime functionality. A critical decision is to modularize the framework into independent parts, so that modifying one will not affect the rest.

Runtime library requirements for a particular target device may be provided by the device manufacturer, system integrator, or by researchers and the open source community. The second half of the runtime system exists on the host as operating system extensions.

Our host implementation for Linux is modular, in that it maintains strict separation between device-specific code and generic code. It is implemented as a set of kernel modules that are loadable on demand and do not require kernel source code modifications.

4.1.1 HYDRA components

The HYDRA runtime is composed of several components as shown in Figure 4.2. It is accessed through an offloading access layer that consists of a user-level library linked to each OA-Application and a kernel-level set of generic services.

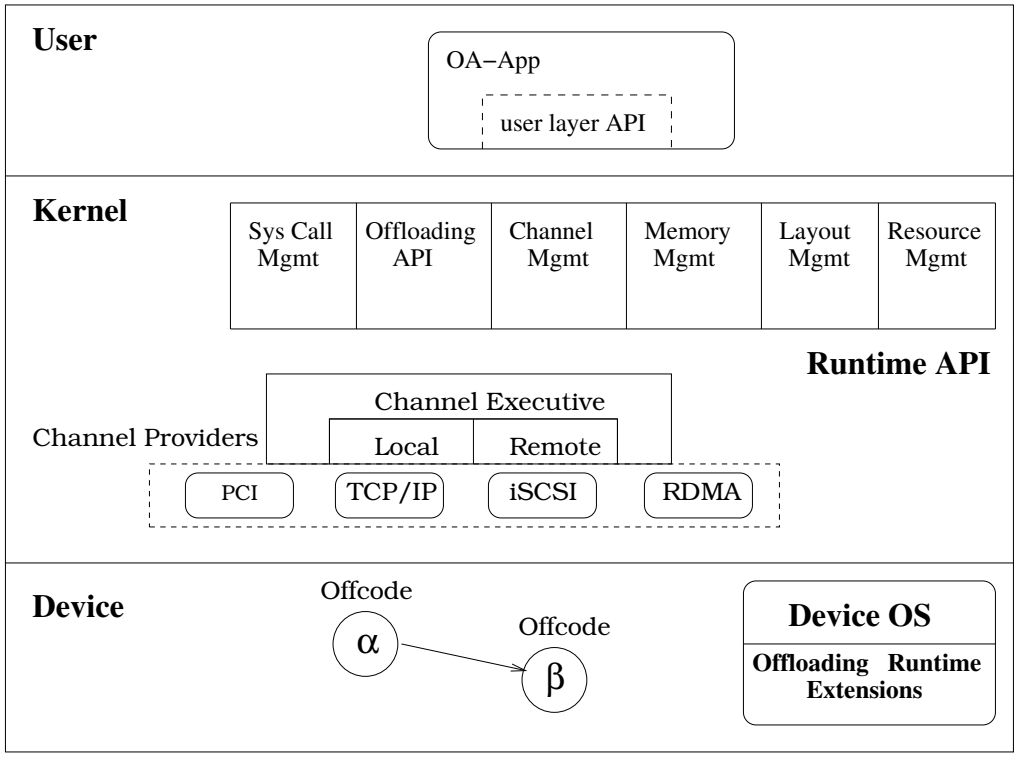

Figure 4.1: System Architecture

The kernel layer consists of several functional blocks. The *System Call Management* and *Offloading API* blocks implement the various APIs defined in the programming model. The *Channel Management* unit manages the channels by interacting with

the *Channel Executive*. This module handles channel creation by using a particular *Channel Provider*. These providers are target-specific and provided as an extended driver for each programmable device. A channel provider creates various specialized channel types to the device and provides a cost metric regarding the "price" for communicating with the device through a specific channel, in terms of latency and throughput. The *Channel Executive* uses this capability information to decide on the best provider for a specific Offcode. The *Resource Management* unit keeps track of all active Offcodes and related resources. Resources are managed hierarchically to allow for robust clean-up of child resources in the case of a failing parent object. The *Memory Management* module exports memory services such as user memory pinning that is used by zero-copy channels. The *Layout Management* unit performs layout-related functionalities such as analyzing the offloading layout graph. This unit receives the offloading layout graph as input and produces the mapping between Offcodes and target devices. The module can be easily extended to support future offloading constraints.

4.2 ILP compiler architecture

The red components in Figure 4.2 mark the dynamic offloading framework's specific components that are discussed in this paper: the ILP compiler and the application API.

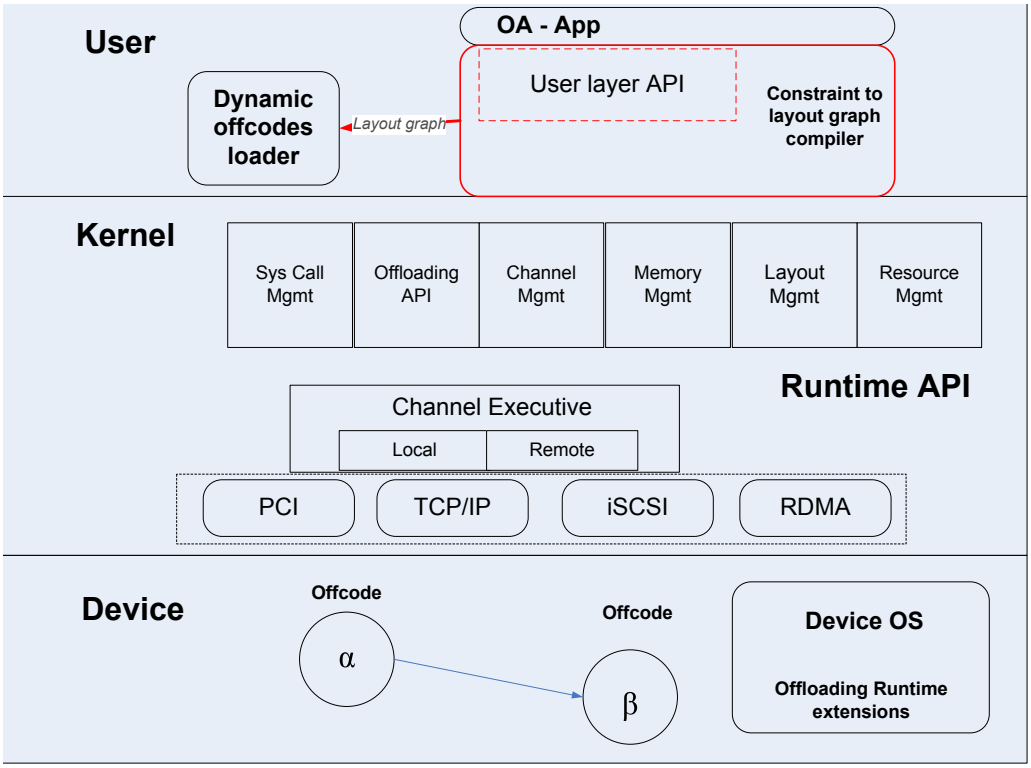

Figure 4.2: System Architecture

4.2.1 ILP compiler components

Device Enumerator

This component is responsible for gathering findings of all leaf enumerators and providing a combined output to the *Offcode matching database*. The device enumerator is implemented as a container of leaf enumerator objects. This component has a proprietary interface, both for input and output, and its implementation is OS independent.

The leaf enumerator provides device identifiers; it searches through some source and reports all found devices, for example, it parses “lspci” output in Linux. Naturally, this

object is operating system aware, as every system represents its devices differently. All enumerators provide their outputs in XML files of a defined format, which the *Device Enumerator* understands.

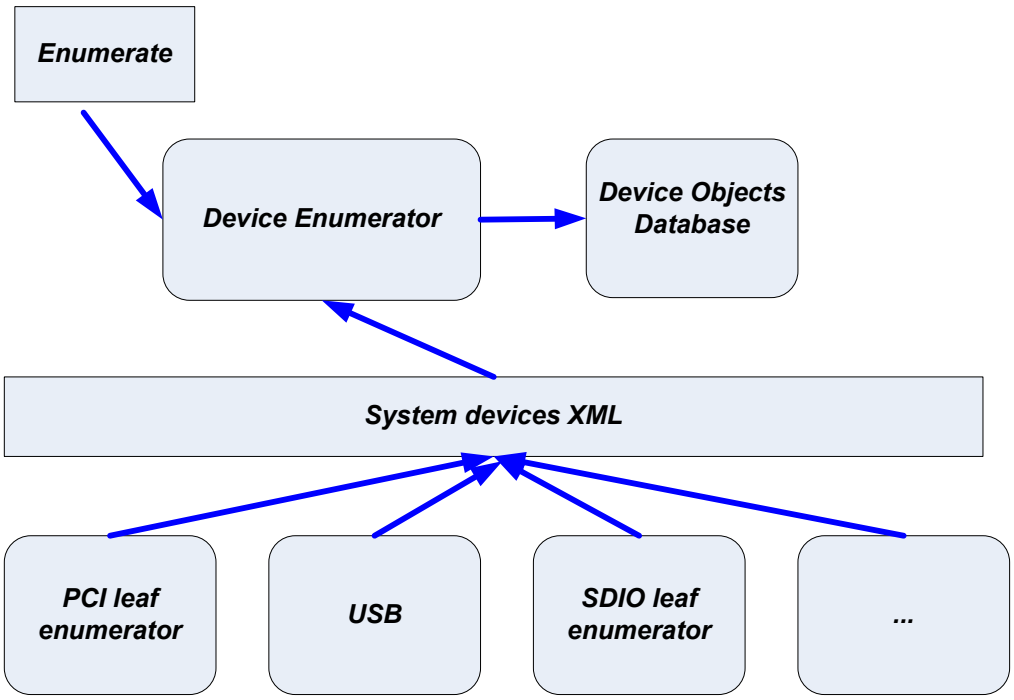

Figure 4.3: Device Enumerator

Offcode Enumerator

This component is responsible for enumerating Offcodes. It iterates through Offcodes' ODFs, instantiates entries that describe Offcodes and provides them to *Offcode matching database*.

As an input, the Offcode enumerator is provided with a list of ODF files describing all known Offcodes. It iterates through the list of files, using an XML parser and ODF scheme on each ODF file to parse it.

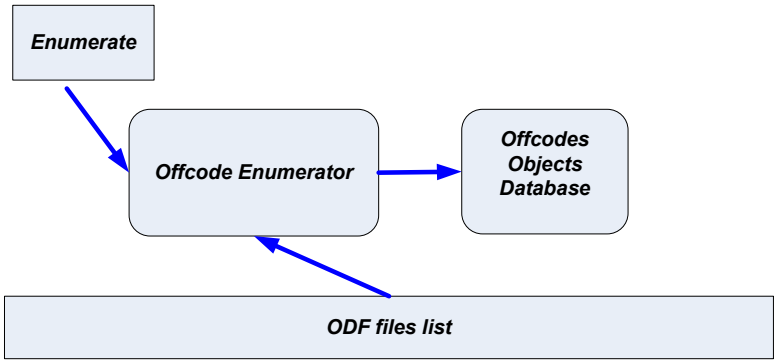

Figure 4.4: Offcode Enumerator

Offcode matching database

This component is responsible for activating the Offcode enumerator and device enumerator, described above, to build a database of Offcodes and devices. Then it filters out Offcodes that don't match any of the devices. It interacts with the graphical front end, providing it with the resulting list of Offcodes eligible for offloading and devices those Offcodes can be offloaded to.

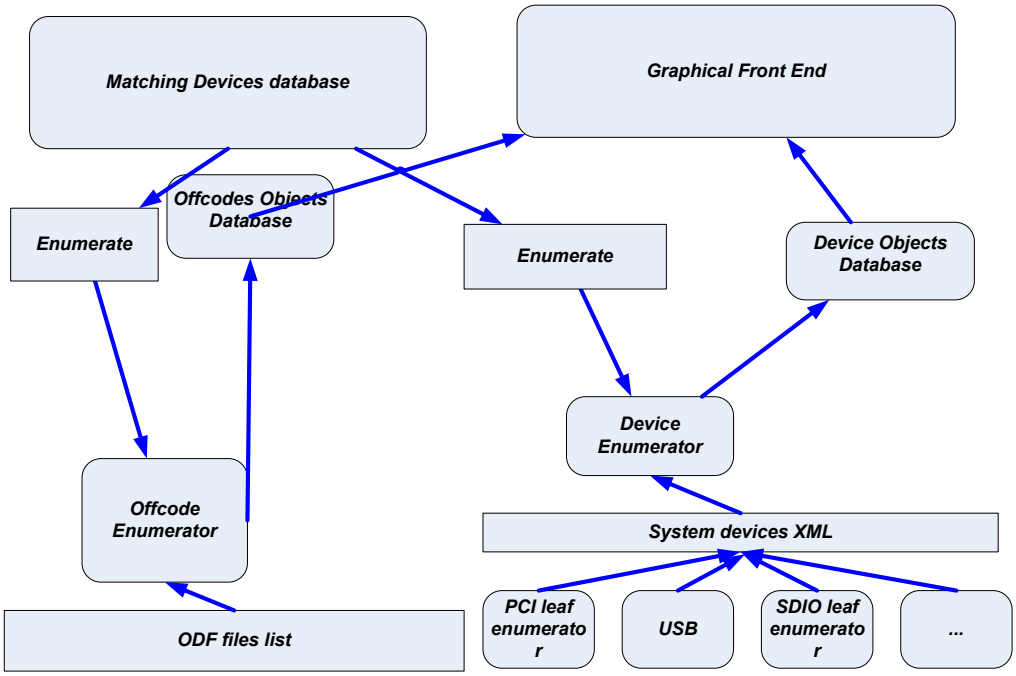

Figure 4.5: Offcode Matching Database

Constraints Container

The constraints container contains objects that represent all constraints available in the system. They have logical representation that is used for ILP formulation. This logical representation shapes the [in]equations, once the constraint has been instantiated for particular Offcodes. Each constraint object has graphical representation as well, which is provided to the graphical front end by the container.

To make our system more scalable, in this case by making it easy to add additional constraints, we use a separate class to describe every kind of constraint. Every such class, for example, the *PullConstraint* class, *gangConstraint* class etc, inherit from a generic abstract (in Java terms) *Constraint class* that provides an identical interface for all kinds of constraints. This abstract class practically enforces all its successors to implement a constructor, a function to provide graphical representation for a graphical front end and a function that provides logical representation for the ILP formulation.

Goal Function Container

The goal functions container is similar to the constraints container, except that the goal functions container holds goal functions representations. There is a similarity between objects representing goal functions and constraints as well. Goal functions also have an equation-shaping logic and graphical representation.

Similarly to the constraints collection, goal functions can also be easily extended by new ones, but it's in hands of the system designer rather than the user.

ILP Subsystem

The ILP subsystem is in fact a set of components. It consists of an external ILP solver engine and several decorators that "stitch" the engine to the rest of the system.

One of the decorators translates abstract equations from constraints and goal functions into the ILP solver input format. The others translate the ILP solver's output to the format readable to the user, a format that the graphical front end understands and, naturally, a format that can be used as an input to the dynamic Offcode offloading engine.

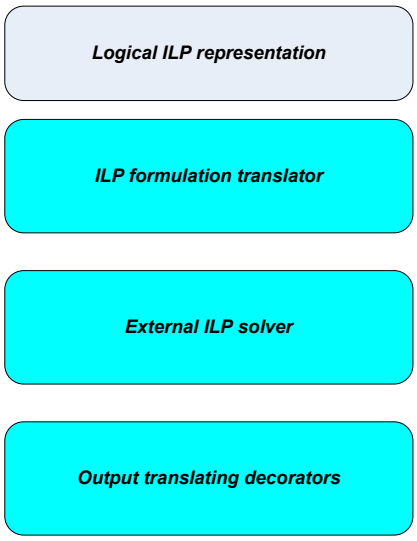

Figure 4.6: ILP Subsystem

It is clear that practically any ILP solver can be rather easily incorporated into this architecture. In fact, the system designer has to only provide those decorators together with the solver (and those are usually relatively easy to implement).

Chapter 5

Evaluation

In this section, we describe two real-life scenarios and translate them into ILP instances according to the model presented in Section 3.2. We present solutions found by solving ILP and compare them to greedy solutions. We also describe the free off-the-shelf ILP solver we used throughout this evaluation.

We chose a specific group of application as experimental system, complex enough to provide non-trivial examples yet simple enough to embrace. Devices, Offcodes, their types and numbers are invariants of the system along with several mandatory constraints. The only unknown left are constraints and goal functions, normally given by the user. We create several scenarios on top of this system by fixing different sets of "user-defined" constraints and goal functions.

5.1 AMPL ILP solver

To solve the ILP problems that we define in the following section, we use AMPL [AMP] SW. "AMPL is a comprehensive and powerful algebraic modeling language for linear and

nonlinear optimization problems, in discrete or continuous variables, developed at Bell Laboratories," can be found at www.ampl.com.

AMPL SW provides both a command line environment for the user and a command line interface, so that it can be invoked from another application. We explain the way this SW should be used in the architecture section of this paper.

5.2 Experimental application

There are, in fact, several applications and several peripheral devices involved. Here are the participating applications:

1. The Tivo PC application that is composed of several Offcodes and is supposed to run on peripherals. It reads content from the OSD [Ruw] device and sends it to the graphics card.

2. A firewall application that filters network traffic and stores a log on another OSD device.

3. A back-up utility that writes on to an OSD device.

Devices present in the system

There are four offloading capable devices present in the system:

- Network Interface Card (NIC)
- Two Object Storage disk (OSD) devices and a
- graphics card (GPU).

We stream a movie and run backups in parallel, while the firewall is running on the NIC, logging part of its activities on one of our OSD devices. Figure 5.1 shows the complete data flow between devices for the chosen application load.

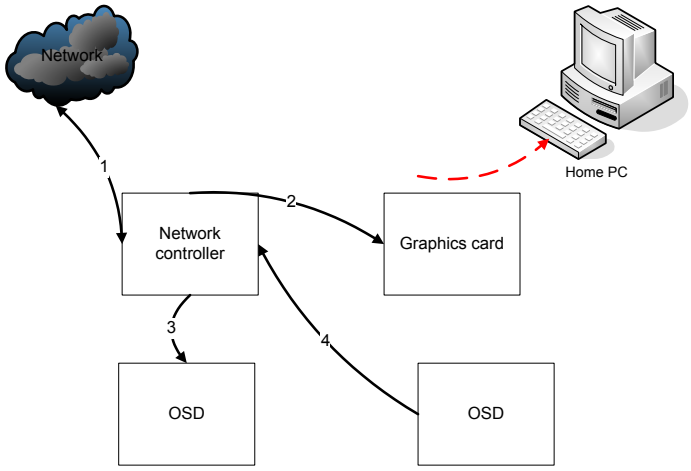

Figure 5.1: Application Data Flow

Available offcodes

The applications described introduce the following Offcodes:

- Offcode A - OSD writer. We have one instance of this Offcode, which is allowed to run on OSD devices OSD1 and OSD2. This Offcode is used to write any received input to the storage device.

- Offcode B - OSD reader. We also have one instance of this Offcode, which is allowed to run on OSD devices OSD1 and OSD2. This Offcode is used to read from the storage device.

- Offcode C - network firewall. We have only one instance of this Offcode, which is allowed to run on the device NIC and implements a simple filtering function for incoming and outgoing network packets. As a part of its filtering operation, this Offcode logs statistics about the dropped packets on the storage device, using Offcode B.

- Offcode D - network movie server. We have only one instance of this Offcode, which is allowed to run on the device NIC; its only purpose is to forward a movie from some source (e.g., storage) to a client, also attached to a network. It uses Offcode B to read the movie stream from the network and uses Offcode A to stream a movie to the network client.

- Offcode E - actual movie streamer. We also have only one instance of this Offcode. It should run on the graphics processing device (GPU) and finally show us the movie. Naturally, it uses the OSD reader (Offcode B) to read the movie from the network.

5.2.1 System ILP formulation

Now, in terms of the ILP formulation described in section 3.2, our application should be represented in the following way:

K **(number of devices) = 4,**

N **(number of Offcodes) = 5**.

Let's now define the decision variables and initial constraints matrices.

Constraint matrix =

	A	B	C	D	E
OSD1	1	1	0	0	0
OSD2	1	1	0	0	0
NIC	0	0	1	1	0
GPU	0	0	0	0	1

X =

$$\begin{pmatrix} X_1^1 & X_2^1 & X_3^1 & X_4^1 & X_5^1 \\ X_1^2 & X_2^2 & X_3^2 & X_4^2 & X_5^2 \\ X_1^3 & X_2^3 & X_3^3 & X_4^3 & X_5^3 \\ X_1^4 & X_2^4 & X_3^4 & X_4^4 & X_5^4 \end{pmatrix}$$

C =

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

So, the first group of constraints (C*X) is the mandatory one. It imposes the model's feasibility and is required for sanity:

$\forall n \in \{1..5\}, \vec{C_n} * \vec{X_n} <= 1$:

And if we formulate it in the AMPL tool, section 5.1, we will get the following:

- n = 1: $X_1^1 + X_1^2 <= 1$,
- n = 2: $X_2^1 + X_3^2 <= 1$
- n = 3: $X_3^3 <= 1$,
- n = 4: $X_4^3 <= 1$,
- n = 5: $X_5^4 <= 1$

Figure 5.2: Mandatory Constraints

- subject to $n1 : X1_1 + X1_2 + X1_3 + X1_4 <= 1;$
- subject to $n2 : X2_1 + X2_2 + X2_3 + X2_4 <= 1;$
- subject to $n3 : X3_1 + X3_2 + X3_3 + X3_4 <= 1;$
- subject to $n4 : X4_1 + X4_2 + X4_3 + X4_4 <= 1;$
- subject to $n5 : X5_1 + X5_2 + X5_3 + X5_4 <= 1;$

Figure 5.3: AMPL Mandatory Constraints

Each Offcode instance can be offloaded to, at most, one device. Note that we give the AMPL the whole constraint without simplifying it.

In the next two subsections, we shall prove two claims of this research:

1. The model presented by us is sound (provides correct results).
2. The model presented by us is optimal, or, at least, provides results better than the greedy approach.

We will show examples for both of our claims. Although, the examples are rather intuitive, this is done only so the reader can easily follow our examples.

5.3 Example set 1

In this set of examples, we provide several groups of user-defined constraints. Each such group, combined with definitions from the previous section, completes the ILP problem definition. Then we present the solutions generated by ILP solver and show that they satisfy the constraints.

5.3.1 Example 1

First, we'll try to build an easy-to-track example with not-too-complex constraints and a trivial objective function.

We would like to impose minimum overhead on the host CPU, i.e., offload as much functionality as possible. The easiest way to accomplish that is by tailoring gang constraints through the whole bunch of Offcodes. We will use the *Symmetric Gang*, so any chosen Offcode results in choosing the rest.

Using decision variables from the previous section, we define the following constraints, complying with the policy above:

- C $\overset{Gang}{\Leftrightarrow}$ A: $-X_1^1 - X_1^2 + X_3^3 = 0$
- D $\overset{Gang}{\Leftrightarrow}$ A: $-X_1^1 - X_1^2 + X_4^3 = 0$
- D $\overset{Gang}{\Leftrightarrow}$ B: $-X_2^1 - X_2^2 + X_4^3 = 0$
- E $\overset{Gang}{\Leftrightarrow}$ B: $-X_2^1 - X_2^2 + X_5^4 = 0$

Figure 5.4 presents the dependency graph of those constraints.

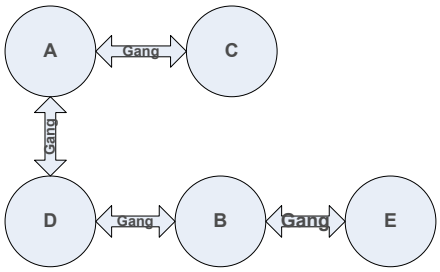

Figure 5.4: Offcode Constraints Graph

Figure 5.5 presents the AMPL tool input representation of these constraints.

As we already mentioned, the user-defined objective function is to maximize the number of downloaded Offcodes : $\max(\sum_{n=1}^{N}\sum_{k=1}^{K} X_n^k)$

By running AMPL software, described in section 5.1, we acquire the following result:

It's easy to see that both sets of constraints are satisfied:

1) sanity ones supplied with application → each Offcode ended up in the right device.

2) gang group we just defined → all Offcodes were chosen.

- subject to $CgangA : -X_1^1 - X_1^2 + X_3^3 = 0;$
- subject to $DgangA : -X_1^1 - X_1^2 + X_4^3 = 0;$
- subject to $DgangB : -X_2^1 - X_2^2 + X_4^3 = 0;$
- subject to $EgangB : -X_2^1 - X_2^2 + X_5^4 = 0;$

Figure 5.5: AMPL Gang Constraints

maximize download: $X_1^1 + X_1^1 + X_1^2 + X_1^3 + X_1^4 + X_2^1 + X_2^2 + X_2^3 + X_2^4 + X_3^1 + X_3^2 + X_3^3 + X_3^4 + X_4^1 + X_4^2 + X_4^3 + X_4^4 + X_5^1 + X_5^2 + X_5^3 + X_5^4;$

Figure 5.6: AMPL Maximize Download

- Offcode A to device 1
- Offcode B to device 1
- Offcode C to device 3
- Offcode D to device 3 and
- Offcode E to device 4

Figure 5.7: Example 1 Solution

- $X_1^1 = 1; X_1^2 = 0; X_1^3 = 0; X_1^4 = 0;$
- $X_2^1 = 1; X_2^2 = 0; X_2^3 = 0; X_2^4 = 0;$
- $X_3^1 = 0; X_3^2 = 0; X_3^3 = 1; X_3^4 = 0;$
- $X_4^1 = 0; X_4^2 = 0; X_4^3 = 1; X_4^4 = 0;$
- $X_5^1 = 0; X_5^2 = 0; X_5^3 = 0; X_5^4 = 1;$

Figure 5.8: Example 1.1 AMPL solution

5.3.2 Example 2

Now we will make this a bit more interesting by requiring that both OSD accessing Offcodes (i.e., the reader and the writer) be offloaded to the same device. We will accomplish this by adding a pull constraint between Offcodes A and B.

A $\overset{Pull}{\Leftrightarrow}$ B: $X_1^1 <= X_2^1;\ \ X_1^2 <= X_2^2$

- subject to $ApullB1 : X_1^1 - X_2^1 <= 0;$
- $subjecttoApullB2 : X_1^2 - X_2^2 <= 0;$

Figure 5.9: AMPL Pull Constraint

In this case, AMPL generates a similar solution:

- Offcode A to device 2
- Offcode B to device 2
- Offcode C to device 3
- Offcode D to device 3 and
- Offcode E to device 4

Figure 5.10: Example 1.2 Solution

The difference between this solution and the previous one is that Offcodes A and B are offloaded into OSD2 instead of OSD1. But this is perfectly fine with the sanity constraints and also with the A $\overset{Pull}{\Leftrightarrow}$ B one.

5.3.3 Example 3

In this case, we force Offcodes A and B to be downloaded to different devices rather than to the same one, by switching the pull constraint to a gang constraint.

A $\overset{Gang}{\Leftrightarrow}$ B: $X_1^1 + X_1^2 - X_2^1 - X_2^2 >= 0$

- subject to $AgangB : X_1^1 + X_1^2 - X_2^1 - X_2^2 >= 0;$

Figure 5.11: AMPL, Yet Another Gang Constraint

In this case, the following solution has been generated:

- Offcode A to device 1
- Offcode B to device 2
- Offcode C to device 3
- Offcode D to device 3 and
- Offcode E to device 4

Figure 5.12: Example 1.3 Solution

We see that, again, the solution matches our expectations. Offcodes A and B have been downloaded to different, yet compatible devices.

So, now, when we have shown the reader that the solution basically works, we can move to the second claim we promised to show - optimality of the solution.

5.4 Example set 2

In this section, we will use the same method we used in the previous section to define several more examples. We will use even more complicated/diverse scenarios. But this time we focus on showing the optimality of the solutions. We choose one scenario to show that a greedy approach finds a suboptimal solution.

5.4.1 Example 1

We will use the same system described in the Section 5.2, while adding the bandwidth constraint, i.e., we need to formulate such a constraint, so the host-to-device traffic will be minimal. Basically, we must ensure that data exchange between two Offcodes won't happen via host, i.e., collaborating Offcodes must be offloaded together (if at all).

In the obvious case, constraints similar to those, which we defined in example 1 of the previous section will do, because all the Offcodes get to be offloaded. However, in this case it is rather a side effect of maximizing offloaded Offcodes .

So, we will use the following group of constraints:

- A $\overset{Gang}{\Leftrightarrow}$ C
- A $\overset{Gang}{\Leftrightarrow}$ D
- D $\overset{Gang}{\Leftrightarrow}$ B
- E $\overset{Gang}{\Leftrightarrow}$ B

and a very simple penalty vector P = (1, 2, 3, 4, 5)

The objective function here will be

maximize bandwidth: $(X_1^1 + X_1^2 + X_1^3 + X_1^4 + 2 * X_2^1 + 2 * X_2^2 + 2 * X_2^3 + 2 * X_2^4 + 3 * X_3^1 + 3 * X_3^2 + 3 * X_3^3 + 3 * X_3^4 + 4 * X_4^1 + 4 * X_4^2 + 4 * X_4^3 + 4 * X_4^4 + 5 * X_5^1 + 5 * X_5^2 + 5 * X_5^3 + 5 * X_5^4)$

Figure 5.13: Bandwidth Goal Function

The solution, obviously, will be the same as in the previous section, because all Offcodes can be offloaded to the devices.

5.4.2 Example 2

Now let's leave the constraints as in the previous example and change bus utilization for some of the Offcodes. This means altering penalty vector P. We choose it to be:

Vector P = (-1, 2, 3, 4, 5)

Now, our solution will be:

- Offcode A to device 0 - DO NOT OFFLOAD Offcode A
- Offcode B to device 1
- Offcode C to device 3
- Offcode D to device 3 and
- Offcode E to device 4

Figure 5.14: Example 2.2 Solution

5.4.3 Example 3

Now, let's change the constraint to be: D $\overset{Gang}{\Leftrightarrow}$ A - if D is offloaded, so should A.

Vector P = (-1, 2, 3, 4, 5)

Now the solution will be to offload all the Offcodes, which is correct, because we gain more by offloading D than we lose by offloading A. On the other hand, a greedy solution would obviously have dropped both A and D.

So, in this section we observed the bandwidth objective function and the optimality of the model we describe in this paper.

Chapter 6

Conclusions and Future Work

In this thesis, we have presented a mathematical model for building an optimized layout graph for Offcodes offloading, as well as a software architecture for an application that implements this model.

We have shown that the model presented by this paper is sound, it provides a correct solution for every example we have presented, and this model provides optimal solutions for those problems, while other methods, greedy for example, provide suboptimal solutions. And although the solutions to the examples above seem rather visible to a user, once a few more constraints are added, it becomes humanly impossible to solve this problem.

We have also presented a scalable OS and HW specification independent SW architecture for implementation of this model. Its scalability allows us to expand this model without much implementation effort, and the sky is the limit here. There are many constraints that can be added. Some might seem non-linear at first, but it is up to the researcher to bring them to a simpler, linear form. As for goal functions, we presented only two of them, but there are many more things that can be optimized, e.g., runtime memory consumption, runtime RAM access latency, number of IO accesses (in case those can be offloaded to

some peer-to-peer protocol,) and much more.

This research is an ongoing effort. The solution we present in this research deals only with an off-line pre-compilation. Taken into account that solving an ILP problem is relatively fast, and there are a lot of studies dealing with optimizing the ILP solution overhead, while preserving the soundness of the solution, the future of this work is to make it updateable in the runtime environment. It should be running in the system all the time, tracking new applications that are being loaded into the system, hot-plug devices being plugged in and out, new Offcodes being updated in the Offcode database, constantly monitoring the system resources, and updating the offloading layout according to the system parameters and to the user needs.

Taken into account the presented SW architecture, it should be relatively simple to adjust it to be an online application. All the presented modules should still be there, several online modules should be added, like hot-plug devices observer, and its databases should be constantly updated. When several solutions are applicable (satisfy the constraints), iteratively change the Offcode prices to be closer to the capability matrix to find the optimal solution or even empirically switch between the solutions to achieve the best results.

Bibliography

[AMP] Windows Standard AMPL. It's a solver for linear optimization problems. Available at site: http://www.ampl.com/.

[BK98] N. Brown and C. Kindel. *Distributed Component Object Model Protocol — DCOM/1.0. Internet Draft*, January 1998. Available at http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt.

[BK03] Matthew Burnside and Angelos D. Keromytis. High-speed i/o: the operating system as a signalling mechanism. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 220–227, New York, NY, USA, 2003. ACM Press.

[BMW03] S. Beyer, K. Mayes, and B. Warboys. Dynamic configuration of embedded operating systems, 2003.

[BN84] A. Birell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BPDS00] Darius Buntinas, Dhabaleswar K. Panda, Jose Duato, and P. Sadayappan. Broadcast/multicast over myrinet using NIC-assisted multidestination messages. In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 115–129, 2000.

[BPS01] D. Buntinas, D. K. Panda, and P. Sadayappan. Performance benefits of NIC-based barrier on Myrinet/GM. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS '01*, April 2001.

[Cur04] Andy Currid. TCP offload to the rescue. *Queue*, 2(3):58–65, 2004.

[ES96] Reuven Elbaum and Moshe Sidi. Topological design of local-area networks using genetic algorithms. *IEEE/ACM Transactions on Networking*, 4(5):766–778, 1996.

[FMOB98] M. Fiuczynski, R. Martin, T. Owa, and B. Bershad. On using intelligent network interface cards to support multimedia applications, 1998.

[GAG94] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 85–94, New York, NY, USA, 1994. ACM.

[HBSG99a] O. Holder, I. Ben-Shaul, and H. Gazit. System support for dynamic layout of distributed applications. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 163–173, Austin, TX, May 1999.

[HBSG99b] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. Dynamic layout of distributed applications in fargo. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 163–173, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[Org] World Wide Web Organization. Web services activity. Available at: http://www.w3.org/2002/ws/.

[rmi99] Sun Microsystems Inc. *Java Remote Method Invocation Security Extension (draft)*, 1999. Available at: http://java.sun.com/products/jdk/rmi/.

[Ruw] Thomas M. Ruwart. OSD: A tutorial on object storage devices.

[SC00] Wen-Tsong Shiue and C. Chakrabarti. Ilp-based scheme for low power scheduling and resource binding. Presses Polytech. Univ. Romandes, 2000.

[Sie98] Jon Siegel. OMG overview: CORBA and the OMA in enterprise computing. *Commun. ACM*, 41(10):37–43, 1998.

[Sri95] R. Srinivasan. RPC: Remote Procedure Call protocol specification version 2, 1995.

[Sun98] Sun Microsystems, Inc. *Java Remote Method Invocation (RMI) Specification*, October 1998. Available at http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

[vLAL92] Peter J. M. van Laarhoven, Emile H. L. Aarts, and Jan Karel Lenstra. Job shop scheduling by simulated annealing. *Oper. Res.*, 40(1):113–125, 1992.

[WBS02] Yaron Weinsberg and Israel Ben-Shaul. A programming model and system support for disconnected-aware applications on resource-constrained devices. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 374–384, New York, NY, USA, 2002. ACM Press.

[WDWAa] Yaron Weinsberg, Danny Dolev, Pete Wyckoff, and Tal Anker. Accelerating distributed computing applications using a network offloading framework. In *IPDPS'07*.

[WDWAb] Yaron Weinsberg, Danny Dolev, Pete Wyckoff, and Tal Anker. Hydra: A novel framework for making high-performance computing offload capable. In *LCN'06*.

[Win] Wayne L. Winston. Operations research: Applications and algorithms. (Thomson, 2004). 4th Edition.

[WJPR04] A. Wagner, Hyun-Wook Jin, D.K. Panda, and R. Riesen. Nic-based offload of dynamic user-defined modules for myrinet clusters. *Cluster*, 0:205–214, 2004.

[wo] w3 organization. Web Service Definition Language (WSDL). http://www.w3.org/TR/wsdl.

[Yar07] Yaron Weinsberg. *An Operating System Specification for Dynamic Code Offloading to Programmable Devices*. PhD thesis, 2007. Supervisor-Prof. Danny Dolev.

[YM93] D. Yuret and M. Maza. Dynamic hillclimbing: Overcoming the limitations of optimization techniques, 1993.

[ZKW02] Qianfeng Zhang, Chamath Keppitiyagama, and Alan Wagner. Supporting mpi collective communication on network processors. *cluster*, 00:75, 2002.