2-5-2010

A Generalized Framework for Automatic Code Partitioning and Generation in Distributed Systems

Viswanath Sairaman
University of South Florida

A Generalized Framework for Automatic Code Partitioning and Generation in Distributed Systems

by

Viswanath Sairaman

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Nagarajan Ranganathan Ph.D.
Srinivas Katkoori, Ph.D.
Hao Zheng, Ph.D.
Tapas Das, Ph.D.
Manish Agrawal, Ph.D.

Date of Approval:
February 5, 2010

Keywords: Code Partitioning , Heterogeneous Systems, Distributed Execution, Code Generation

DEDICATION

To Bhavani

ACKNOWLEDGEMENTS

I would like to thank Dr. Nagarajan Ranganathan for guiding me through the process of pursuing my research and writing my dissertation. When things were moving slow, the encouragement that he provided was the only motivating factor that helped me in pursuing my goals. I would like thank him for being patient with me and helping me to regain focus and believing that I can complete this research work. This thesis dissertation would not have been possible without his in depth expertise and infinite patience in the process of helping me towards the the final goal of defending my dissertation. I would also like to thank Dr. Srinivas katkoori, for helping on innumerable accounts and always supporting me in all aspects. I would like to thank Dr. Tapas Das, Dr. Manish Agrawal and Dr. Hao Zheng for kindly consenting to be a member in my committee and guiding me in my graduation. I would like to thank Daniel Prieto and Peter Schaivo of the technical support for the computer science and engineering department. This dissertation research would have been very difficult without their constant efforts in fixing all the issues with software and hardware. I would also like to thank Ms. Theresa Collins, of the graduate program for the long hours and infinite patience in guiding me through all paperwork and the readmission process. I would like to thank Dr. Dmitry Goldgof, Graduate Program Coordinator of the computer science department for allowing me this opportunity to complete my dissertation. I would like to thank Dr.Sudeep Sarkar, for clearing all my requisitions and making sure that I would be given fair chance to present my work. Finally I would like to thank Catherine Burton, for the invaluable amount of time and expertise in guiding me with all the necessary steps for graduation. I'm really thankful for all the support that she has provided in all these years to successfully complete all formalities at USF. I would also like to thank Narender Hanchate, Karthikeyan Lingasubramanyan and Manish Shukla who have been with me in this long journey and have always supported me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

A GENERALIZED FRAMEWORK FOR AUTOMATIC CODE PARTITIONING AND GENERATION IN DISTRIBUTED SYSTEMS

Viswanath Sairaman

ABSTRACT

In distributed heterogeneous systems the partitioning of application software to be executed in a distributed fashion is a challenge by itself. The task of code partitioning for distributed processing involves partitioning the code into clusters and mapping those code clusters to the individual processing elements interconnected through a high speed network. Code generation is the process of converting the code partitions into individually executable code clusters and satisfying the code dependencies by adding communication primitives to send and receive data between dependent code clusters. In this work, we describe a generalized framework for automatic code partitioning and code generation for distributed heterogeneous systems. A model for system level design and synthesis using transaction level models has also been developed and is presented. The application programs along with the partition primitives are converted into independently executable concrete implementations. The process consists of two steps, first translating the primitives of the application program into equivalent code clusters, and then scheduling the implementations of these code clusters according to the inherent data dependencies. Further, the original source code needs to be reverse engineered in order to create a meta-data table describing the program elements and dependency trees. The data gathered, is used along with Parallel Virtual Machine (PVM) primitives for enabling the communication between the partitioned programs in the distributed environment. The framework consists of profiling tools, partitioning methodology, architectural exploration and cost analysis tools. The partitioning algorithm is based on clustering, in which the code clusters are created to minimize communication overhead represented as data transfers in task graph for the code. The proposed approach has been implemented and tested for different applications and compared with simulated annealing and tabu search based partitioning algorithms. The objective of partitioning is to minimize the communication overhead. While the proposed approach performs comparably with simulated annealing and better than tabu search based approaches in most cases in terms of communication overhead reduction, it is conclusively faster than simulated annealing and tabu search by an order of magnitude as indicated by simulation results. The proposed framework for system level design/synthesis provides an end to end rapid prototyping approach for aiding in architectural exploration and design optimization. The level of abstraction in the design phase can be fine tuned using transaction level models.

CHAPTER 1
INTRODUCTION

1.1 Distributed Heterogeneous Systems

Distributed computing is the process of executing an application in a parallel manner on multiple processing elements simultaneously. Depending on the architecture of the processing elements, topology and communication methodology employed, distributed systems can be divided into many categories.

- *Homogeneous System:* All processing elements comprising the system have similar basic architecture.
- *Heterogeneous System:* (Figure 1.2.)] Processing elements comprising the system have varying architecture and processing power.
- *Multiprocessor System:* Homogeneous system in which processing elements multiple instances of the same basic architecture and are usually connected to each other on a bus based communication network.

Parallel execution is the process of executing different parts of a single application parallely. Parallel execution does not necessarily constitute distributed execution. Parallelism in the execution of application source code can be achieved by means of different mechanisms at different levels of compilation. The highest level of parallelism is when the application compiler detects parallelism in the source code and provides means of simultaneous execution. The operating system provides the next lower level of parallel execution by employing threads. Multi-threaded operating systems are capable of executing multiple threads of a single application source. Parallelism can also be achieved at the processor level through the use of processor multi-threading. The CPU is capable of choosing instructions from multiple instruction queues thus exhibiting parallelism. Distributed execution is the process of executing the application from physically different locations.

Traditional computing systems follow a single line of execution thread through the application source code as indicated by figure 1.1. Applications containing inherent parallelism suffer slower execution speeds in the case of single thread of execution. Distributed heterogeneous systems provide an execution environment where multiple lines of execution through the application is possible. Even though compiler/operating system level multi-threaded architecture support can essentially achieve a similar result, the execution environments are clearly different. Multi-threaded architectures can create multiple lines of execution within the application, however the ability to execute multiple threads is limited by the processing capability of the processing unit. At any given instant there is only one active thread on the processing unit. In the case, when a thread in execution is halted on an event like an interrupt, a context switch occurs and the line of execution is transferred to another thread.

Figure 1.2. provides an illustration of a distributed heterogeneous system along with an application with different parts of the application mapped to the different processing elements of the distributed heterogeneous system. In the case of distributed heterogeneous systems, the application code is partitioned and mapped to different processing units in the system. Parallel execution of multiple line of application source code is thus guaranteed. Heterogeneous systems consist of processing elements with varying processing capabilities, memory architectures and also include processing elements with entirely different underlying architectures. Different system architectures for processing elements require different compilers and the final executable version of the application once compiled is not portable and can be executed only on the host processor. The differences in the compilation process is definitely a disadvantage in the use of heterogeneous systems. The advantage of using heterogeneous systems is the wide variety of architectures available to choose from.

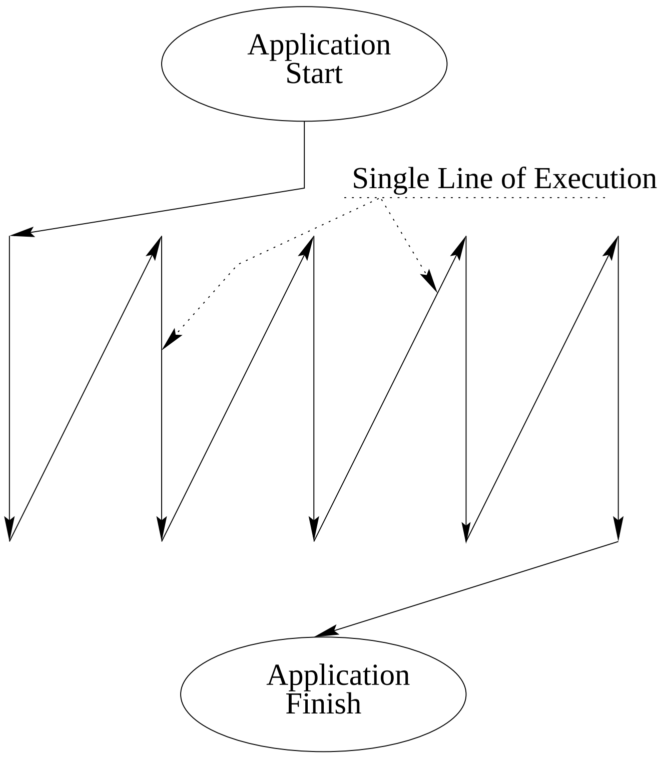


Figure 1.1. Conventional Single Line of Execution of an Application

Distributed systems provide an environment for high speed parallel computation, provided the inherent parallelism in the application is utilized to the maximum.

The implementation of distributed heterogeneous system for mapping an application involves a number of design issues which are discussed briefly. The first issue in the design process is the choice of the processing units that would be the components of the distributed system. The processing units can vary from off-shelf generic processors to custom designed ASIC's performing specific tasks. Partitioning the applications in such a way that some tasks are executed on generic processors and some tasks of the application are executed on custom designed hardware is known as the hardware/software partitioning problem. The combination of the processing units comprising the heterogeneous system required for achieving optimum performance when executing any application is specific for that application. The choice of the off-shelf processing units available in the market vary from micro-controllers with small memory to generic embedded processors with large

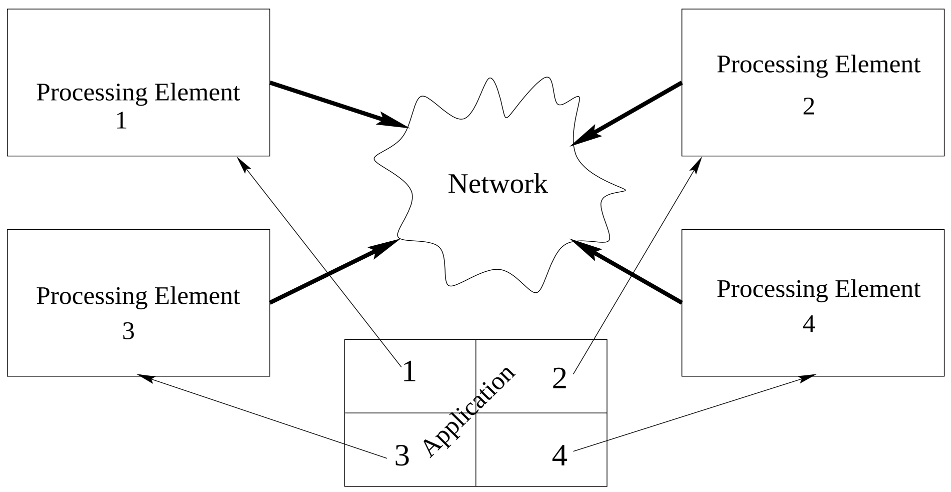


Figure 1.2. Illustration of a Distributed Heterogeneous System

memory and complex instruction sets. Generic processors are expensive but programmable and support complex instructions. A generic processor possesses the capability to execute any task of the application. Micro-controllers are RISC processors with a lower degree of programmability. The cost of the micro-controllers is much less when compared to generic processors. The second issue involved in the design of distributed systems is the communication methodology used for data transfer between the components of the system. Fast and efficient communication methodologies are essential for reducing network delays and improve overall communication. The architecture for the distributed heterogeneous system is finalized when the processing elements have been chosen and a high speed communication methodology has been identified.

1.2 Mapping Application on a Distributed Heterogeneous System

Design and development of heterogeneous distributed computing systems is a challenge in itself. Efficient mapping of an application onto a distributed system is essential for low execution time and high performance. The mapping process must result in an optimized partitioning of the application code in order to promote speedy execution while maintaining a balanced work load among the processing elements of the system. In recent times a large number of scientific and consumer electronics applications are being designed to execute on distributed heterogeneous systems in order to achieve optimum performance. Consumer electronic companies define the term *Time-to-market* for any product as the total amount of time taken by the product starting from its inception as an idea on the drawing to the final product roll out. Time-to-market is vital and needs to be real short, if the company needs to maintain a sustained market presence. According to [9] a delay of about four weeks in the time to market can lead to a loss of about twenty percent and a delay of ten weeks would lead to almost a fifty percent loss. The enormous size of the design space leads to difficulty in making faster design level decisions.In the following chapters the word *application* refers to an application that is intended for a distributed heterogeneous system.

Figure 1.3. shows an overview of the process of mapping an application onto a distributed heterogeneous system. The first step in the process, is generating a task graph from the application. The task graph contains detailed information pertaining to each task in the application. Each task in the application is then profiled to obtain further information pertaining to each task, including the number of CPU cycles taken by the task for execution, power consumed in order to perform the task, size of a task in terms of the amount of data needed for performing the task and dependencies that this task shares with other tasks. The collected information is processed and the passed on to the partitioning algorithm to generate clusters of application code. An optimal mapping is identified for each code cluster and each component of the distributed system.

1.2.1 Code Partitioning

Code partitioning can be defined as the process of forming partitions (*task clusters*) by combining individual *tasks*, represented as nodes on the task graph, and mapping each partition onto a processing element in the heterogeneous distributed system. There are three inputs required for the code partitioning algorithm. A hardware library containing specific information on the components of the distributed heterogeneous system architecture, application source code and the task graph extracted from the application source code. The

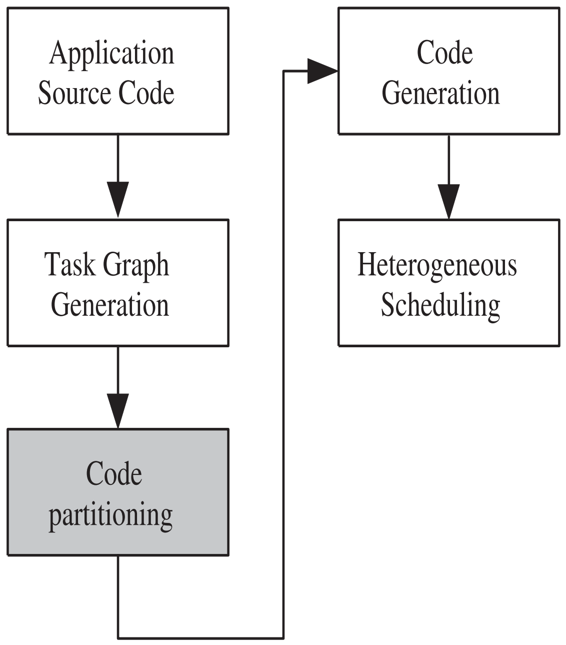


Figure 1.3. Design Flow for Mapping an Application on a Distributed System

output of the code partitioning algorithm is a set of partitions consisting of code clusters along with specific mapping information for each partition with a corresponding processing element $P.E$ on the distributed system. The code partitioning step in the mapping process is vital for the optimal execution of the application. The solution obtained from the partitioning problem can be further optimized in the heterogeneous scheduling process. Code partitioning consists of three important tasks. They are

- *Profiling:* consists of parsing the task graph and the application source code in and constructing a tree data structure in order to compute and hold communication and processing costs for tasks.

- *Partitioning:* consists using the profiling information and applying a partitioning algorithm on the task graph to generate individual code partitions.

- *Mapping:* consists of identifying the best possible processing element for each individual code partition by the process of evaluating different combinations of partitions and processing elements.

1.2.2 Code Generation

The next step in the mapping process is Code Generation. The definition of the term code generation varies with the context in which it is used. In the broadest sense of the word, it means to generate or create source code. In object oriented analysis and design (*OOAD*) domain, code generation commonly refers to synthesis of application source code for objects modeled using a modeling language like the Unified Modeling Language (UML). In the context of this work, the term has been used to indicate the process in which the partially complete code clusters obtained as output from the partitioning process are embedded with communication primitives in order to complete them and convert them to individually executable code segments. The code clusters are analyzed individually to identify and isolate data dependencies between the clusters of source code. Appropriate communication primitives are incorporated into the code clusters in order to permit efficient exchange of data and complete the execution of the application code. The steps involved in the code generation process are as follows.

- *Identify Dependencies:* Analyze individual code clusters in order to identify and isolate data dependencies between partitions.
- *Add Code:* Additional code is added to the original source of each partition in order to convert them to nearly independent executable programs.
- *Communication Synthesis:* Add communication primitives to the source code to resolve inter partition data dependencies.

The last step of the mapping process is heterogeneous scheduling. The clusters of code on interconnected processing units are scheduled for extracting the additional parallelism in the code.

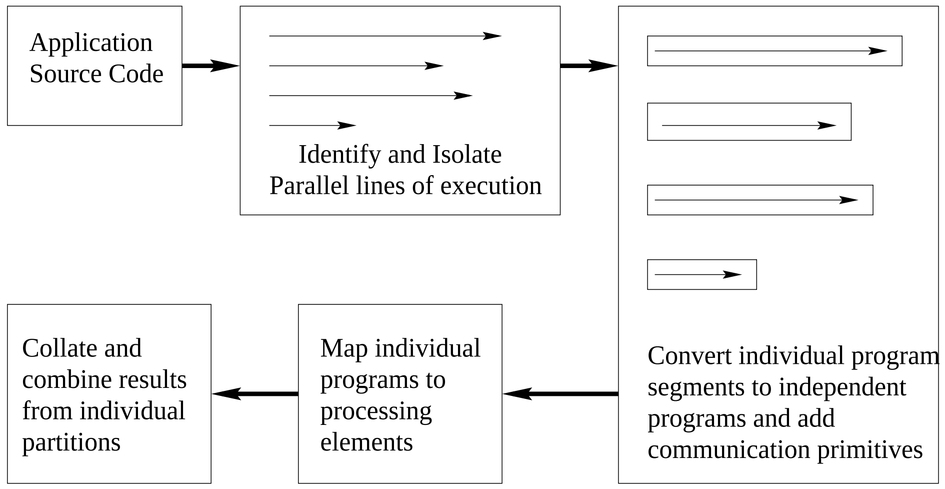

Application Source Code
Identify and Isolate Parallel lines of execution
Convert individual program segments to independent programs and add communication primitives
Map individual programs to processing elements
Collate and combine results from individual partitions

Figure 1.4. Detailed View of the Mapping Process

An elaborate depiction of the mapping process is provided in figure 1.4. The partitioner isolates individual code clusters from the original application source code and identifies a suitable processing element for mapping. The code generator completes each individual code cluster with communication primitives necessary for sending and receiving data between inter dependent tasks. The individual code clusters after the code generation step are independent of one another except for any data dependencies. The results are finally collated and returned.

1.3 Mapping Implementation Details

1.3.1 Clustering

Clustering is a technique that is ideally suited for partitioning problems. The partitioning algorithm used for code partitioning in this work is a variation of the K means clustering algorithm proposed by MacQueen in 1967. The algorithm is simple, unsupervised and aggressive . The run time for the algorithm is faster to most of the commonly used partitioning algorithms by orders. The disadvantage of the algorithm is that there is no guarantee that the solution obtained is optimal. In order to overcome this short coming the algorithm can be executed a large number of times over the same set of data and the best solution can be selected. The low run time for the algorithm allow at-least about 100 iterations of the algorithm without any significant set back on the overall run time . Clustering follows a greedy approach. Hence steering the algorithm in the direction of the optimal solution is based on the formulation of the problem and cost factor definitions. The clustering algorithm does not provide the optimum number of partitions the application can be split into. The algorithm is executed for about five different cluster numbers and the three best number of partitions are chosen to be ideal.

1.3.2 Partitioning Details

The quality of the mapping generated by the partitioning algorithm is quantified by the amount of reduction in communication overhead and by the amount of work load balance among the processing elements. There is a need for a central automated entity to identify the most suitable processing element for a particular task in the process of mapping the tasks of the application. In order to achieve maximum performance the mapping is performed by considering maximum number of attributes associated with the task. This section elaborates on finer details involved in making the decision choices that affect the performance of various stages in the mapping process. The application source code is initially divided into smaller tasks and passed through a task graph generator. The output of the process is a task graph. A task graph can be loosely defined as a graph with the set of all nodes representing the tasks of the application and the set of all edges representing control (or data) flow between tasks. A control data flow graph (CDFG) is also a form of task graph. A data flow graph is also a task graph. The task graph generator used in this work [19] parses an application source code and generates a acyclic directed graph with node representing tasks along with the time in seconds taken to execute the particular task and the edges representing data dependencies between tasks along with amount of data that is transferred between the tasks in kilobytes. The application code is profiled statically for obtaining additional information and the all the data is normalized and stored in a data reserve. The task graph generator [19] used for generating tasks from the application source code is targeted specifically for applications written using the *C* language. Hence this work is targeted towards systems using the procedural language *C* for application description.

Granularity level of a partitioning algorithm is an important factor that affects the performance of the algorithm. The *Granularity* level, a partitioning algorithm is implemented for is decided by the approximate size of each task comprising the application. Tasks consisting of loops, blocks and whole functions of code are defined to be coarse in granularity. Advantages of using coarser granularity are reduced execution times for the algorithm and lower communication overhead. The partitioning algorithm is characterized as operating of fine granularity when tasks comprising the application consist of a few instructions(or individual instructions) of source code. Fine granularity results in a huge solution space. The increase in size of solution search space increases the execution time of the partitioning algorithm and also increases the number of design choices. In this work the granularity of the tasks is assumed to be coarse.

1.4 Contributions of the Dissertation

In this work, we describe a generalized framework for automatic code partitioning and code generation for distributed heterogeneous systems. A model for system level design and synthesis using transaction level models has also been developed and is presented. The framework consists of profiling tools, partitioning methodology, architectural exploration and cost analysis tools. The partitioning algorithm is based on clustering, in which the code clusters are created to minimize communication overhead represented as data transfers in task graph for the code. The proposed approach has been implemented and tested for different applications and compared with simulated annealing and tabu search based partitioning algorithms. The objective of partitioning is to minimize the communication overhead. While the proposed approach performs comparably with simulated annealing and better than tabu search based approaches in most cases in terms of communication overhead reduction, it is conclusively faster than simulated annealing and tabu search by an order of magnitude as indicated by simulation results. The proposed framework for system level design/synthesis provides an end to end rapid prototyping approach for aiding in architectural exploration and design optimization. The level of abstraction in the design phase can be fine tuned using transaction level models. The significant contributions of the this dissertation that were described are listed as follows.

- A new framework for automatic code partitioning and code generation.
- A new algorithm for code partitioning based on k-means clustering for partitioning application programs
- Clustering algorithm based on execution time, power consumption and communication overhead as metrics.
- Algorithm and software for code generation to convert code clusters into independent code executables.
- A new TLM based model for architectural level design space exploration.
- A complete software tool that can perform automatic code partitioning and code generation.
- Results on select benchmarks.

1.5 Outline

This chapter introduced and discussed in brief the design of heterogeneous distributed systems and the mapping of application onto such a system using powerful tools of code partitioning and code generation. The rest of the dissertation is organized as follows. Chapter two briefly covers relevant research and industry efforts in the areas of code partitioning and code generation. Code partitioning algorithm based on hierarchical clustering is discussed in detail in chapter three. The software architecture of the implementation of the code partitioning algorithms along with the profiling tools used in the work are discussed in chapter four elaborately. Chapter five explains the code generation approach used in this work. A detailed view of the proposed framework for TLM based SoC exploration is presented in chapter six. Chapter seven contains conclusions and scope for future work.

CHAPTER 2
RELATED WORK

This chapter provides a comparative analysis of existing contributions in related work area. This work is a combination of code partitioning and code generation. The first part of the chapter provides a classification of the partitioning problem for different application environments. A brief overview of the different approaches for the partitioning applications with different methods of description is presented next. A classification of code generation problem based on the mode of communication between the partitioned clusters of code is then described briefly. Finally the chapter concludes with an overview of the various approaches to the code generation problem and some of the most significant contributions in that field. Code partitioning for a heterogeneous environment is a challenge by itself. The problem has been investigated at various levels of abstraction. Code partitioning has been studied under different architectures with varied constraints.

2.1 Partitioning Problems in Different Application Domains

Code partitioning is essentially a special case of the partitioning problem. Based on the specific application domain the code partitioning problem is mapped on, the partitioning problem can be classified into categories. The application domain categories include the following.

- *HW/SW Partitioning:* Partitioning applications modeled distributed embedded systems using the hardware/software co-design process.
- *Graph/Automata:* Partitioning applications modeled using a Task graph or Finite state automaton essentially for functional verification.

- *Heterogeneous Systems:* Partitioning problem for applications designed for execution on heterogeneous systems, implemented using high level programming languages.

Based on the finer implementation details of the application and the target platform, the partitioning problem for heterogeneous systems can be further classified into the following categories.

- Multitasking applications designed for mapping onto multiprocessor systems.
- Object oriented approaches requiring a high level of compiler and operating system support.
- Application implemented using procedural languages like C.

The specific reason for a separate classification for object based applications is based on the reasoning that object based applications need to be partitioned in congruence with object definition and behavior. Object based applications are generally implemented to promote modularity of the source code and the ease of scalability with the aid of strong compiler and operating systems support. Procedural languages introduce additional challenges to the partitioner. Code density in the case of procedural languages is not uniform hence optimal partitioning of tasks and simultaneously maintaining an uniformly balanced work load is difficult. In the case of multi tasking application for multiprocessor systems, the partitioning problem is essentially a two part problem. The first part is at the level of the operating system and its ability to provide simultaneous execution (*multi-threading*). The second part of the problem is the actual partitioning for the multi processor system. Figure 2.1. represents some of the major contributions in the area of code partitioning.

2.2 Code Partitioning Based Approaches

Relevant work in code partitioning is presented in three categories. Applications implemented using high level programming languages and mapped on to heterogeneous and multiprocessor based systems are discussed first. In heterogeneous systems we present three sub categories, multi tasking based applications, object based applications and procedural language based applications.

2.2.1 Heterogeneous Systems

One of the earliest approaches for application partitioning on heterogeneous architectures was based on ADA [22] programming language. A call-rendezvous graph (CRG) is constructed from the application using a parser. A CRG can be compared closely to a task graph. The nodes in the CRG represent program units/tasks/subprograms. The edges in the CRG represent call and task interaction relationships. A weighted CRG (WCRG) is obtained by including edge weights representing inter node communication. The technique described by the author involves parsing ADA applications through a parser and subsequently employing a partitioner to tightly map the clusters of program units onto the heterogeneous systems with the primary goal of reducing communication overhead. The partitioner isolates tasks based on Abstract Data Types or program units with behavior similar to objects. The technique works well in a small to medium scale. The approach is however not scalable to large or complex applications. The partitioner is limited in capacity to handle fewer number of tasks and has a high run time. Another disadvantage of the approach is the granularity of partitioning. Since the size of each node in the CRG is a program unit and the partitioner is targeted for object based behavior the partitioning is performed at a very high level and cannot be modified to finer granularity.

Nacul et al [4], have developed an automated code partitioner with a code generator for dynamic multitasking applications. The phantom system is developed primarily for embedded application with very little operating system support. The motivation for the work is the idea of serializing compilers. A *serializing compiler* is a translator that converts a multitasking application (in this case written in the C language) into processor independent source code. The embedded processors are usually shipped along with tool chains that can be used for compiling applications for that specific processor. The output of the phantom system can be compiled by the tools chains accompanying the embedded processor. The code partitioner is based on a generic clustering algorithm. The partitioner divides the original application source code into code blocks and a scheduler is synthesized to execute the blocks of code. Each block of code is termed by the authors as an Atomic Execution Block(AEB). AEB's are non preemptive and represent a set of nodes on a CFG. The phantom system uses compile time information to divide the application into tasks and then form the partitions of code blocks. The approach has been tested for dynamic multi tasking applications. The major downside of the approach is the generic clustering mechanism used to form the code blocks. The clustering lacks efficiency as iterative improvements to the solution are achieved by randomly chasing the next move. The clustering mechanism employed does not take into account the performance variation of the code blocks on varied architectures.

The authors of the J-orchestra approach [26] propose an automatic partitioning for real time Java based systems. The J-orchestra approach is targeted for applications designed for execution in ubiquitous computing environments. *Ubiquitous computing* environments are heterogeneous computing systems where the processing elements comprising the environment are varied in architecture and processing power and also include embedded processors interconnected to each other in a wired and/or wireless network. The framework provides tools for rapid prototyping for mapping applications on to ubicomp domains. The automated process explores the design space for optimal partitioning and mapping of applications. The key contribution of this work is automating the design exploration process. In the case of *ubicomp* environments exploring the design space for isolating optimal high performance designs is a challenge. The implementation of the design has been tested to be dependable, sophisticated and scalable. A case study was performed with the *Kimura* system comprising of about 4400 lines of uncommented source. The automatic partitioner was used in the development of the *kimura* ubicomp system. The partitioner used in the J-orchestra approach used the original application source code as input and then rewrites the source to reduce inter partition communication in a distributed environment. The implementation also includes a graphical user interface for the front end. The communication overhead is reduced by rewriting the source code to convert remote method invocations to direct object referencing in the byte code. The result of the approach is automatic partitioning is achieved easily but the final solution is not essentially optimal. The model proposed is targeted primarily for Java based object oriented applications. The scope of the partitioning solution is thus restricted.

Partitioning of instructions for wide issue super-scalar processors is dealt with in [31]. The focus of their work is on mapping code to the functional units of a clustered architecture with the objective of optimizing the tradeoff between work load balancing and reducing communication overhead. The approach is targeted for multitasking applications mapped on to multi processor environment.

The closest comparison to our work is in the area of procedural language based partitioner. In [18] the authors present an approach for partitioning and mapping programs written in ANSI C onto a Custom Computing Machine (CCM). CCM's are usually multi FPGA computing platforms. The application needs to partitioned for efficient execution on functional units and the original application source code is converted to a data flow representation and mapped on to FPGA's. The next step in the process is scheduling the operations for high speed execution. A simulated annealing based partitioning algorithm is implemented and the scheduler uses a partial schedule as input to complete the entire schedule. The compiler consists of two parts, an architecture independent front end which handles all the lexical and syntactical analysis in order to generate directed acyclic graphs (DAG's) based on the application source and a architecture dependent back end in order to generate the architecture specific code. The authors have used an intermediary form - GDL graph description language to represent the architecture specific code. The schedule is tested by converting source to VHDL and the validity of the approach is ascertained. The main set back of the approach is using a simulated annealing based partitioning algorithm. Simulated annealing guarantees a solution that is optimal but at the cost of very high running time. As the complexity of the source code increases simulated annealing based partitioner take many days to converge. Hence the approach is not scalable.

In 2006 after this work was completed there was a case study [35] completed investigating the code partitioning for high performance reconfigurable architectures. The author considers the task of partitions between multiprocessors and FPGA's. In this case study different partitioning schemes are investigated for a custom designed architecture.

2.2.2 Graph/Automata Approaches

The code partitioning problem has also been considered equivalent to graph / automata based partitioning algorithms. The nodes in the graph/automaton represent components of the application. Hence the problem can be formulated as a graph/automaton partitioning problem and solved. The contribution [23] is based on extended finite state machines(EFSM) models. The application source code is input in a high level language, in this case ESTEREL. The EFSM is generated from the application source code. The approach works only if the application can be modeled as an EFSM. In most cases translating the given application and modeling it as an EFSM is extremely difficult and in some cases may even be impossible. Hence this is a major drawback for this contribution. The partitioning is performed for the domain of hardware/software partitioning problem. Hence this contribution is discussed in detailed in the next section.

A genetic algorithm based approach for scheduling and mapping conditional task graphs for synthesis of low power embedded systems was investigated in [7]. The major restriction of this approach is that it is restricted to CDFG's and the major focus is on the synthesis of a schedule for the application on an embedded system.

2.2.3 Hardware/Software Partitioning Systems

Hardware/Software partitioning is an important step in the hardware/software codesign process. The problem consists of generating a mapping for each of the task in the application. The tasks can be executed on a general purpose processor (software) or can be synthesized into independent instances of processing elements performing only specific computations (hardware). This is equivalent to the code partitioning problem

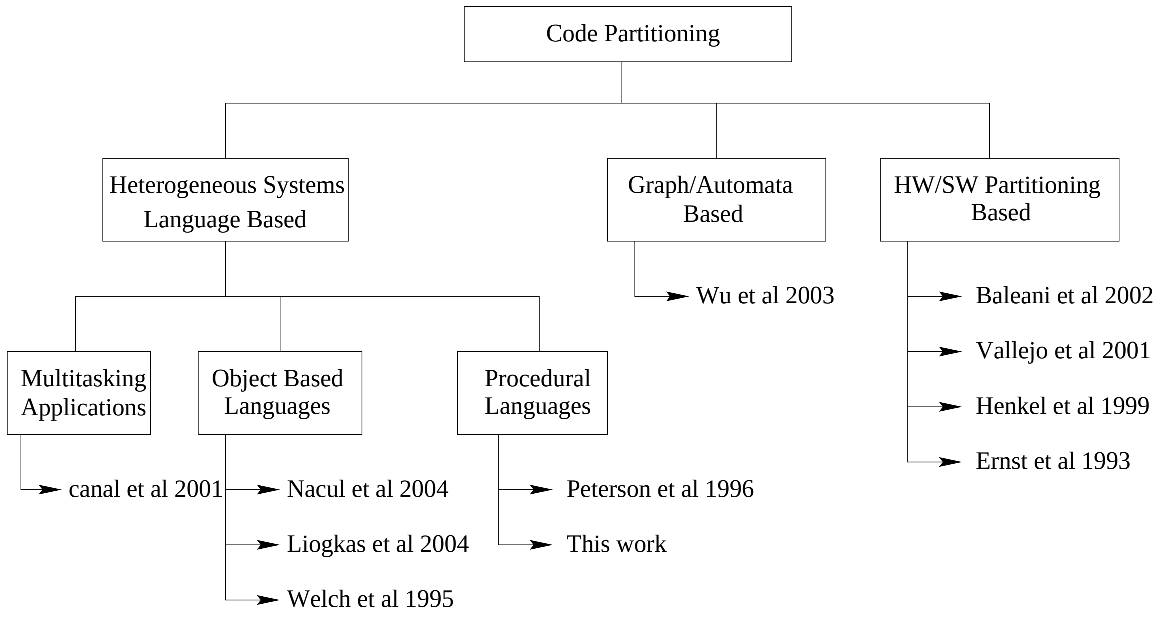


Figure 2.1. A Taxonomy of Prior Works in Code Partitioning

where the processing elements are components of a heterogeneous systems with varying architectures, processing speed and cost. In this section we discuss some contributions in the hardware/software partitioning area that are closely comparable to our work.

In [23], the authors use finite automaton and constraint based approach for a target architecture consisting a general purpose core and a reconfigurable functional unit. The reconfigurable part is implemented using programmable FPGA units. The partitioning process is automated and takes a high level description of the application as input (in this particular case the language ESTEREL is used). Open source GNU based compiler *GCC* tool chains are used to convert the high level description of the application into ANSI C functional equivalent tagged clusters of code. The code clusters are mapped on to the FPGA units. The partitioning algorithm clusters the individual components of the generated ANSI C code based on individual properties of the component. The authors term this process as clubbing. The main contribution includes an automated partitioner and a code generator for synthesizing ANSI C code for the FPGA units. The number of cycles needed for each C language instruction is computed using GCC tool chains as performed in our work. The work does not consider other factors important to the partitioning process like the power consumption of individual clubs and memory usage. The main limitation of this work is that it can be applied only for extended finite state machines.

Another notable contribution in considering power in the process of partitioning was investigated in [16] and [15]. A comparative study of simulated annealing and Tabu search based algorithms applied to system level partitioning is provided in [28]. The main contribution of this work is system level hardware/software partitioning algorithm which takes input in the form of a high level machine independent description of the application in VHDL. Communication between hardware (co-processor) and the software (micro processor) is achieved using message passing calls implemented in VHDL. Partitioning algorithms are implemented using simulated annealing and tabu search based approaches and the authors claim that after testing on bench marks they find tabu search based approach to be more effective than simulated annealing based approach and that tabu search based approaches converge faster than simulated annealing based approaches. The partitioning algorithm is fine grained to the the level of loops and basic blocks. Including fine implementation details at the system level, restricts the number of available design choices. This work is referenced in the section for related with in code partitioning even though it is system level hardware/software design because of the similarities between in the problems encountered in both these areas in the partitioning step.

In[10], the authors use binary code of the application to extract information necessary for partitioning. This technique is efficient as binaries provide accurate runtime information, the only drawback being the overhead in applying the technique to different target architectures. The approach of clustering for hardware/ software partitioning was attempted in [24] but it does not take into account the effects of power dissipation.

There are other notable contributions partitioning applications implemented in procedural languages designed for hardware/software partitioning environments. Some of the contributions are as follows. In NIMBLE [37] the author have constructed an entirely retargetable framework for mapping applications onto reconfigurable embedded systems. The target architecture is described using an architecture description language(ADL). The

ADL and the high level description of the application in ANSI C are the inputs to the framework. The partitioning algorithm used in this work is based on hierarchical clustering approach. Fine grained granularity at the level of loops and basic blocks is used by the partitioning algorithm, hence the solution obtained fully utilizes maximal available instruction level parallelism. The output of the hardware/software partitioner comprises of synthesized code for FPGA's and modified application C source code. The partitioner analyzes the source code to identify loops and modifies the source to include loop unrolling for optimal execution on the FPGA's. The down side for this approach is the fine granularity of the tasks comprising the application. The modeling is primarily intended for maximizing parallelism with the loop unrolling technique. The fine level of granularity of the task at that level prohibit scalability of the approach to huge applications. The effective execution time even for moderate sized applications would be highly expensive rendering this approach to be in effective.

2.3 Communication Mechanisms in Distributed Applications

Communication methods used in distributed applications depend on processing elements comprising the target architecture. the application source language, topology of the distributed system and the methodology used for partitioning. General techniques used for communication in most distributed applications include CORBA, MPI, PVM and Java RMI. Phantom [4] uses a shared memory technique employing semaphores for synchronization. Shared memory systems do not require remote call communication procedures but synchronization of the shared data in order to avoid deadlock and for data coherency. Yang [34] uses message passing for communication in a shared memory environment. In [26], Java RMI is used for communication and hence useful only for Java based applications. We briefly discuss the features of some the communication methods used in distributed systems.

Remote method invocation (RMI) allows Java developers to invoke object methods, and have them execute on remote Java Virtual Machines (JVMs) [11]. Under RMI, entire objects can be passed and returned as parameters, unlike many other remote procedure call based mechanisms which require parameters to be either primitive data types, or structures composed of primitive data types. That means that any Java object can be passed as a parameter - even new objects whose class has never been encountered before by the remote virtual machine. RMI is Java specific, and writing a bridge between older systems becomes the responsibility of the programmer. Additionally, it is designed for object based applications only.

Common Object Request Broker Architecture (CORBA) [11] is a competing distributed systems technology that offers greater portability than remote method invocation. Unlike RMI, CORBA is not tied to a particular language, and as such, can integrate with legacy systems of the past written in older languages, as well as future languages that include support for CORBA. CORBA is not tied to a single platform (a property shared by RMI), and shows great potential for use in the future. In contrast, for Java developers, CORBA offers less flexibility, because it does not allow executable code to be sent to remote systems. CORBA services are described by an interface, written in the Interface Definition Language (IDL). IDL mappings to most popular languages are available, and mappings can be written for languages written in the future that require CORBA support. CORBA allows objects to make requests of remote objects (invoking methods), and allows data to be passed between two remote systems. Remote method invocation, on the other hand, allows Java objects to be passed and returned as parameters. This allows new classes to be passed across virtual machines for execution (mobile code). CORBA only allows primitive data types, and structures to be passed - not actual code.

Message Passing Interface is a standard for writing message passing programs allowing efficient communication by avoiding memory-to-memory copying and allowing overlap of computation and communication. MPI [11] is more suitable for large multiprocessor homogeneous systems. In order to utilize the maximum capability of the MPI, the target architecture needs to be restricted to homogeneous systems with identical architectures.

PVM is built around the concept of a virtual machine and is useful when the application is executed on a networked collection of hosts, particularly if the hosts are heterogeneous. PVM [11] contains resource management and process control functions that are important for creating portable applications that run on clusters of workstations and MPP. The larger the cluster of hosts, the more important PVM's fault tolerant features become. The ability to write long running PVM applications that can continue even when hosts or tasks fail, or loads change dynamically due to outside influence, is quite important to heterogeneous distributed computing.

We choose PVM as the message passing technique in our method because it has these salient features. It is best suited for largely heterogeneous distributed systems. Other features like easy portability and robustness make it a better choice. MPI is good but not easily portable and works better in a homogeneous environment. Even though PVM based communication techniques require a minimal operating system support, it has been utilized in this work in order to facilitate a rapid prototype design at the system level. This enables easier design space exploration without hard design level decisions of specific implementation details.

2.4 Code Generation for Partitioned Applications

Automatic Code Generation has been used in a lot of scenarios in some cases as a mere translator for conversion from one language to different other target languages. Figure 2.2. shows some of the most significant contributions in the area of code generation. Pertaining to partitioning problems code generation can be studied for the two distinct methods : a) code generation from State chart or automata [6],[14],[17],which (mostly seen in HW/SW partition problems) and b) code generation for software application partition problems. The closest to our approach are Pangea, Coign and J-orchestra.

In [34], the problem of parallelization of applications for distributed memory architectures is addressed. A directed acyclic graph (DAG) was used to model parallel computation. The tasks in the DAG were clustered with the help of a scheduling algorithm called DSC

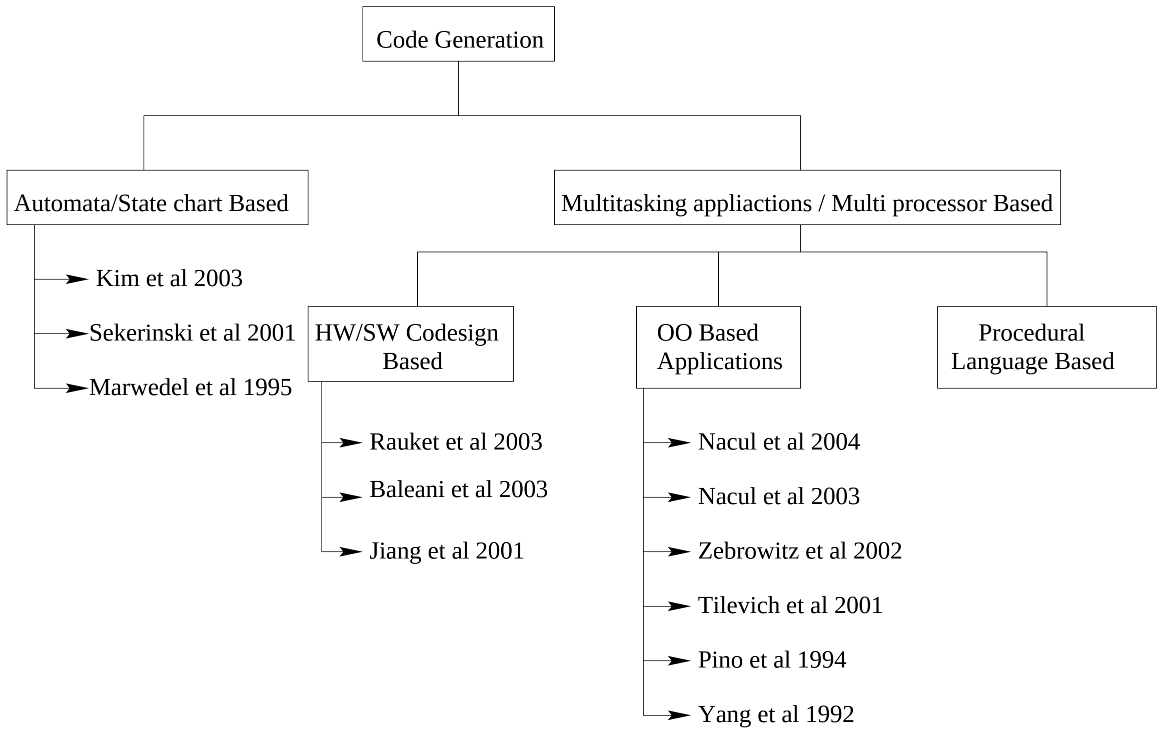


Figure 2.2. A Taxonomy of Prior Works in Code Generation

(Dominant Sequence Clustering). The code generation method proposed by them involves executing a schedule of an arbitrary task graph based on an asynchronous communication model.

The tasks in the DAG access data in a shared memory programming style. Message Passing was used for communication. The authors in [17], propose a code generation framework for hybrid automata which deals with continuous and discrete data dependency. The automatic code generation process proposed is decomposed into two phases: one translating each primitive into a piece of code and the other scheduling the pieces of code consistent to data dependency. In [21], a CAD tool, SynDEx is used for partitioning and code generation. The authors develop fast automatic prototyping process dedicated to parallel architectures. In [23], the authors introduce an automated hw/sw partition and code generation flow for control applications. It uses the EFSM model and derives hw/sw implementation automatically from high-level synchronous specifications. Using the automated synthesis flow, embedded applications can be mapped from high-level language specifications like ESTEREL, down to hw/sw implementation on the reconfigurable architecture platform. However, this approach is limited to the applications that can be modeled and programmed by extended finite state machines. Recently, there has been a lot of work in automatic code generation for embedded systems pertaining to software synthesis of hardware/software Co-design problems. Most of these methods use conversion of automata into code. This method is more suitable for hw/sw Co-design partitioning problems. Though this method can also be used for software applications, there is an overhead of translating of the automata into software code.

There has not been much focus to the problem of code generation for distributed software. Mostly, when applications need to be distributed, programmers manually partition the application and assign those sub programs to machines. With the help of middle ware such as RRPC, CORBA and DCOM, these sub programs are successfully executed on the respective machines. Often the techniques used to choose a distribution are ad hoc and create one-time solutions biased to a specific combination of users, machines, and networks. To address this issue, the authors in Coign[12] proposed a automatic distributed partitioning system for binary software applications based on the COM model. Given an application (in binary form) built from distributable COM components, Coign constructs a graph model of the application's inter-component communication through scenario-based profiling. Later, Coign applies a graph partitioning algorithm to map the application across a network and minimize execution delay due to network communication. Using Coign, even an end user (without access to source code) can transform a non-distributed application into an optimized, distributed application. This method cannot be used as a generalized approach as very few of the real world applications are written as collections of the COM component.

J-Orchestra operates at the Java byte-code level and rewrites the application code to replace local data exchange (function calls, data sharing through pointers) with remote communication (remote function calls through Java RMI , indirect pointers to mobile objects). The resulting application is guaranteed to have the same behavior as the original one (with a few, well-identified exceptions). J-Orchestra receives input from the user specifying the network locations of various hardware and software resources and the code using them directly. A separate profiling phase and static analysis are used to automatically compute a partitioning that minimizes network traffic.

The main objective of our program and J-orchestra is the same - Automatic application partitioning for a distributed embedded environment. The J-orchestra approach is restricted to object based environments and specifically to Java based applications. The approach described in this work, is more generalized and based on applications that can be described using the procedural language C. In the case of J-orchestra approach the whole application source is converted to byte code and the byte code is rewritten and then the partitions are identified. partitions are computed. In J-orchestra all the local data communications are replaced with remote communication calls thus resulting in additional overhead.

Pangaea[3] is an automatic partitioning system. Pangaea is based on the Java Party infrastructure for application partitioning. Java Party is designed for manual partitioning and operates at the source code level, Pangaea is also limited in this respect. The source code is analyzed and partitioned and the primitives generated are specific to the back-end adapters. In this work the adapters tested were based on CORBA, RMI and Java Party. The work is restricted to a specific set applications that can be modeled using object based techniques. The technique is directed to take full advantage of the inherent parallelism in the application. The partitioner is not responsible for the back-end distribution of the independent code clusters. Hence the technique requires manual intervention and also substantial programming for developing the back end adapters for specific execution environments.

2.5 Context of Dissertation Research

Significant research contributions in the areas of code partitioning and code generation were discussed in this chapter. The primary short comings of all the research works existing at the time of implementation of this work are listed below.

- No automatic code partitioning tools were available
- Existing works were targeted for specific architectures
- In most applications, the code partitioning step was done with human intervention
- Very minimal support was available for procedural languages. Most of the tools were targeted towards object based applications

This dissertation describes a code partitioning technique for applications which can be described using the procedural language C, and are targeted for distributed heterogeneous environments A new code generation methodology for executing partitioned code clusters in a distributed environments is also described. In this chapter, prior work in the area of code partitioning and code generation was studied with emphasis to those directly related to this work. An extensive comparison of the existing contributions in both the areas was provided. The conclusive differences between the existing works and this dissertation was also provided.

CHAPTER 3
CLUSTERING METHODOLOGY

The code partitioning problem is equivalent to the traditional graph partitioning problem. The edges of the directed graph translate to control flow in the code. The nodes represent centers of computation like tasks or instructions. This chapter describes in detail the methodology applied for partitioning the task graph and there by partitioning the application source code. The chapter is organized as follows. The partition algorithm is explained in detail first. . The motivation behind using a clustering based approach is then discussed in detail. The architecture of the environment used and the assumptions are covered then The chapter concludes with a set of all the results of the experiments conducted to estimate the correctness and the quality of the partitioning algorithm on a set of benchmarks.

3.1 Partitioning Process

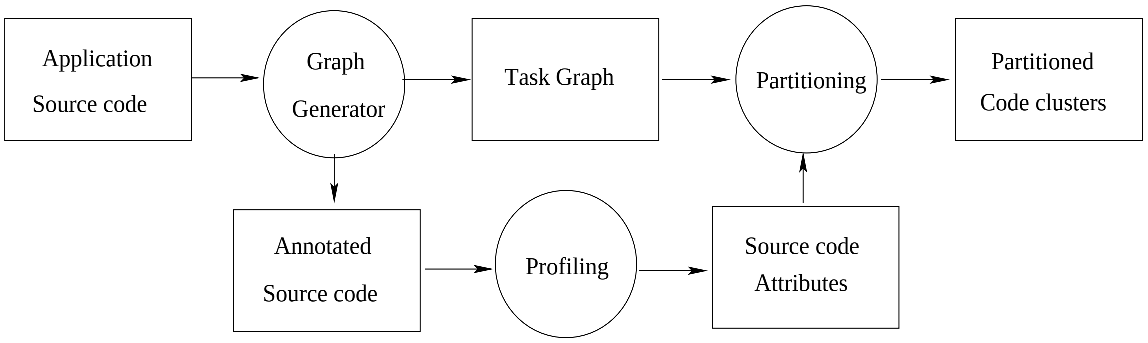


Figure 3.1. Steps Involved in Code Partitioning

The process of code partitioning has two major sub processes. The application description which is in the form of a single C language Source code file is partitioned into code clusters. Each cluster is then mapped onto a PE of the distributed system. Figure 3.1. shows a detailed view of the steps involved in the partitioning process.

3.1.1 Overview of the Partitioning Metrics

The first step in the partitioning process is the generation of a task graph from the application description. The application description in the form of C-language source code is passed through the task graph generation tool[19]. The nodes of the task graph generated correspond to blocks of code from the actual source of the application. The edges of the task graph represent dependencies in the execution of the application with the weights corresponding to the amount of data transfer in Kilo Bytes (KB). Figure 3.2. shows a snap shot of the task graph generated from the application source code by the task graph generation tool. Each node in the figure is represented with two numbers. The first number is the task number corresponding to that node and the second number is the execution time required for the task on the host processor in which the task graph was generated.

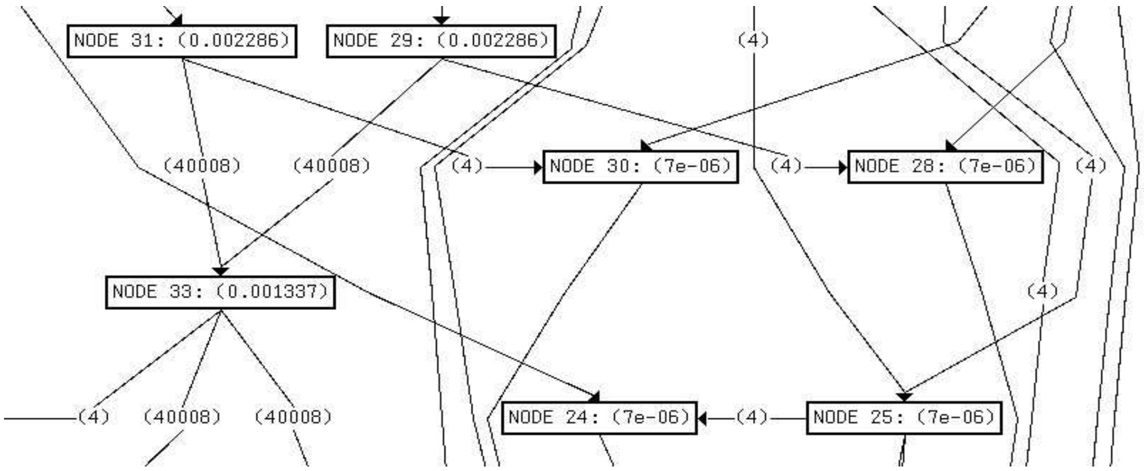
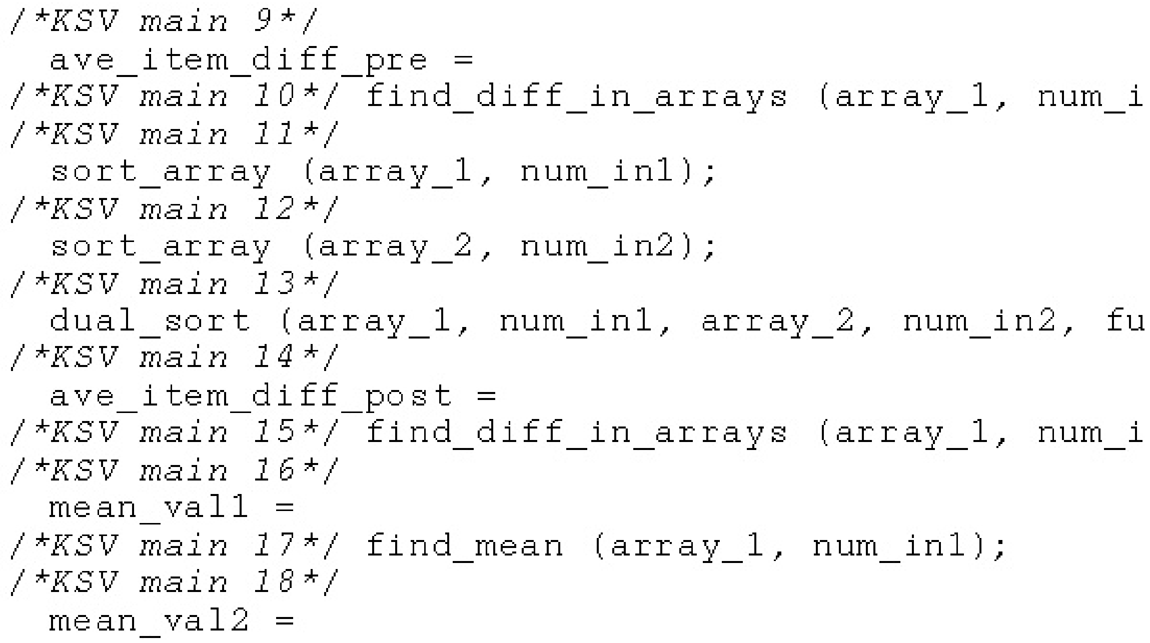


Figure 3.2. Snap Shot of a Part of the Task Graph

The other output produced by the task graph generation tool is annotated C code. Figure 3.3. shows a snap shot of the annotated source code output from the task graph generation tool.Comments are introduced in the original source code in order to indicate task boundaries and task numbers. The annotations include the functional block to which the task belongs to and the task number. The keyword *main* indicates that the task belongs to the main functional block of the application. The annotated C code is used as an input to profiling tools to obtain the various attribute values for each of the tasks. This work investigates the partitioning problem in the context of a distributed embedded system environment. The exact fit of a code cluster to a PE yields the most optimal solution. In order to map a task to the most suitable PE, the task is analyzed thoroughly and attribute values computed. Larger number of attributes considered for tasks help model the behavior of tasks in an accurate fashion.



```
/*KSV main 9*/
  ave_item_diff_pre =
/*KSV main 10*/ find_diff_in_arrays (array_1, num_i
/*KSV main 11*/
  sort_array (array_1, num_in1);
/*KSV main 12*/
  sort_array (array_2, num_in2);
/*KSV main 13*/
  dual_sort (array_1, num_in1, array_2, num_in2, fu
/*KSV main 14*/
  ave_item_diff_post =
/*KSV main 15*/ find_diff_in_arrays (array_1, num_i
/*KSV main 16*/
  mean_val1 =
/*KSV main 17*/ find_mean (array_1, num_in1);
/*KSV main 18*/
  mean_val2 =
```

Figure 3.3. Sample Annotated Code

3.2 Motivation for Clustering

Tasks have varying values for the different attributes on each of the different PE's. In this work the entire *population* is the set of all tasks comprising the application, *clas-*

sification is partitioning the application into clusters and the *metrics* are the attribute values of the tasks on each of the PE's. The primary motivation for applying a clustering based approach to partition the application source code, is to utilize the inherent nature of tasks to cluster within the application, there by achieving maximum optimization of all the factors contributing to the total cost of the solution. Another important characteristic of clustering based approaches is the exceptionally low execution time, when compared to other partitioning methodologies[24]. Clustering tasks by considering a large number of attributes associated with the task optimizes the clustering process and improves the quality of the solution. Since the attributes of the tasks are varied in nature, in order to cluster tasks, the attributes are taken collectively and modeled to generate a new metric called the *closeness* metric.

3.2.1 Attributes for Closeness

The *Closeness* metric between any two clusters can be compared to the Euclidean distance between the clusters in a K dimensional space, with K denoting the number of attributes used in clustering process. The first step is identifying the attributes of the tasks that are vital for modeling their behavior on any PE. The attributes of the task considered for computing the closeness metric in this work case are as follows:

- Data communication coming into a task
- Execution time for the task on the PE
- CPU cycles necessary for execution on the PE
- Power consumption of the task on the PE

The attribute values for each task are different in nature, are represented using different units and vary on widely different scales. The amount of data communication is represented in Kilo Bytes (KB). The execution time for the task is measured in Milli-seconds. The CPU usage cycles is a number and is usually in the range of hundreds of thousands. Power is measuring in nano joules. The closeness metric is a combination of all the above attributes. A straight forward combination of the different attributes in this case is not an optimal solution, since there is huge difference in the attribute values. In order the combine the attributes in the most optimal manner, the attributes need to be normalized. In this work a global normalization [8] policy is adopted. The normalization process can be tweaked in order to obtain the closeness metric which is biased towards one or more of the attributes of the tasks.

3.2.2 Computing Closeness

The values of the attributes for the task are obtained from the profiling information. Some attribute values are obtained by using mathematical extrapolation. For all the tasks in the application description, the maximum and minimum values for each of the attributes is computed. The Range(R_i) of i th attribute is given by the formula. $R_i = Max_i$ - Min_i. The normalized metric is given by the following formula.

$$\begin{aligned} X[i,j] &= indicator_k * ((attribute_k(i) \\ &- attribute_k(j))/R_k) \forall i \neq j, \forall k \in 2,3,4 \end{aligned} \tag{3.1}$$

where $indicator_k$ has a value of 1 if the value for attribute k exists or has a value of 0. In order to tweak the normalization process to be biased on one/some particular attributes, the $indicator_k$ for that particular k can be modified to have a different value and indicator can now include a range of values instead of just zero or one. The closeness value is computed for every pair of clusters at every iteration of the partitioning process. The driver routine uses the closeness value to steer the partitioning algorithm to optimal solution.

3.3 Partitioning Algorithm

The partitioning algorithm is implemented with a driver routine to execute the clustering and compute the cost the solution. The driver routine for partitioning executes the function *improvisecluster()* in a tight loop until the termination condition is reached. The termination condition for the driver routine is designed for achieving a multiple goals. Clustering is based on a greedy approach, hence there is a probability that the solution obtained may be a local optima. Hence the driver routine executes the clustering in a tight loop. Another disadvantage of this approach is that the algorithm might not terminate even after reaching the global optima, thus resulting in an infinite loop. In order to overcome these drawbacks, the termination condition is actually a two part solution. The algorithm checks for if one of the following conditions is met. 1) iterations exceed a fixed maximum number(*user-defined*) 2) cost criteria has been attained. In this case there is still a small probability that the solution obtained might be a local optima. Hence in order to absolutely confirm that the solution is indeed the global optimum, the entire driver routine is executed a hundred times with varying number of user defined iterations for each run. The *improvisecluster()* function contains the clustering algorithm and would be explained in detail subsequently.

The reduced running time of the clustering approach helps the algorithm to converge to a solution in timings that are faster than simulated annealing and tabu search based approaches by orders of magnitude. Clustering algorithm can be executed for a large number of iterations to guarantee a solution and still finish executing much earlier than the time taken for a single iteration of simulated annealing based approach. The partitioning algorithm based on clustering returns cluster centers. A cluster center is the central node for a group of nodes of nodes. Tasks are assigned to a cluster center C_i if the task forms is reachable from the center of C_i and the closeness value for C_i with the other cluster centers is the least when compared with the closeness values in the case when the task belongs to another cluster center. Figure 3.6. gives an illustration of the clustering process. The figure shows that there two cluster centers at present namely a and b. Each cluster center has attached to it a set of two nodes. The node x is currently not associated with any cluster. There are two options for assigning the node x. The entire application can be now partitioned with x being assigned a as the cluster center or b as the cluster center. Every

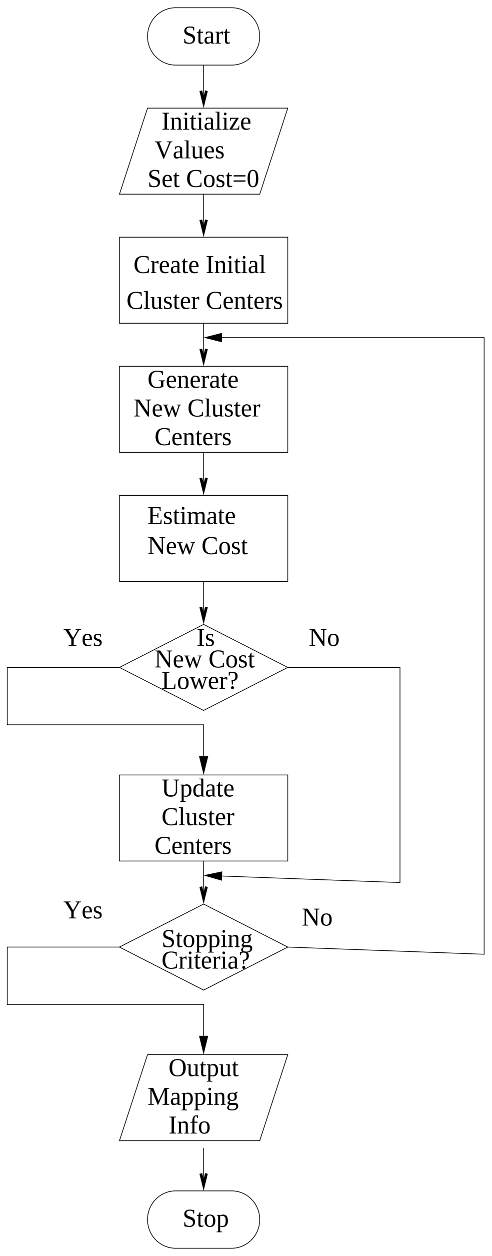


Figure 3.4. Flow Chart for the Clustering Algorithm

```
Result : cluster centers
currentcost = 0;
createinitialsetofclustercenters ;
saveclustercenters ;
while (nocostimprovement || maxiterations) do
    IMPROVISE_CLUSTERS() ;
    estimate_cost() ;
    if ($updatedcost < currentcost$) then
        save_old_configuration() ;
        flagupdated = true ;
    end
end
if ($flagupdated \neq true$) then
    revert_to_old_configuration() ;
end
```

Figure 3.5. Algorithm for Partitioning Based on Clustering

non cluster center node in the application task graph is assigned to every cluster center in the graph and the cost is computed. The configuration leading to a total cumulative optimum cost is the ideal mapping. Clustering based algorithms follow greedy approach. The algorithm proceeds in the direction yielding the lowest cost. Unfavorable or high cost solutions are reverted to to an earlier configuration with lower cost and the algorithm proceeds to search in a new direction.

3.3.1 Improvising Clusters

The improvise cluster function takes as input a set of randomly generated cluster centers and works towards the optimal solution.

A node (say h) which is not already present in the set of initial cluster centers, is chosen from the list of tasks to replace an existing cluster center i if the amount of gain obtained by replacing i with h is the maximum over the set of all non cluster center tasks. This process is repeated until the number of new cluster centers selected equals the amount of clusters requested by the user. The new set of cluster centers thus obtained is subjected to improvisation process until the stopping criteria is reached. Clustering is a heuristic

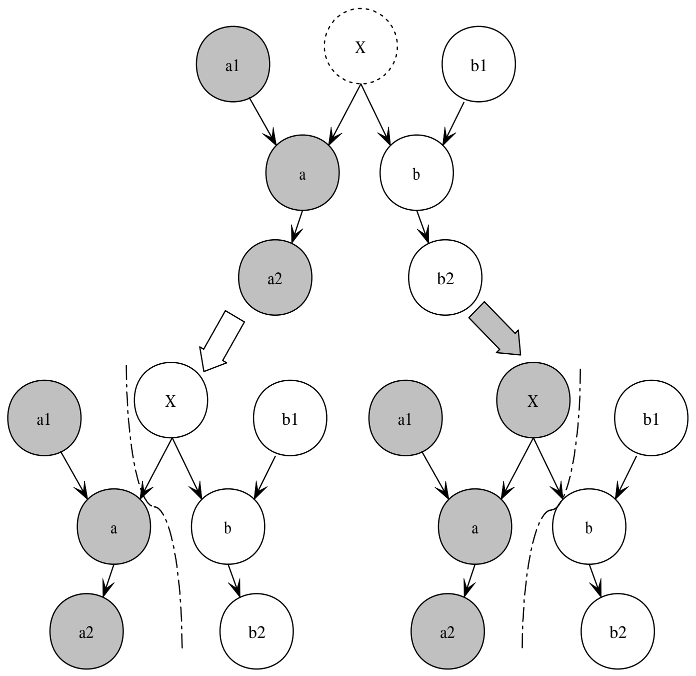


Figure 3.6. Illustration of Possible Partitioning Options

```
Data   : tasklist ;
         normalized task attributes ;
         hardwarelibrary ;
         initial set of clustercenters ;
Result : Improvised set of clustercenters
createinitiallistofclustercenter ;
for ($i \in clustercenter$) do
    choose ($h \ni clustercenter$) ;
    for ($j \in clustercenter \& j \neq i$) do
        $gain_j$ = closeness($j$, $h$ ) ;
    end
    choose $h$ with least $\sum gain_j$ ;
end
```

Figure 3.7. Algorithm for Improvising Clusters

based approach and hence does not guarantee a solution. Minimal execution times for the clustering algorithm is ideally suited for executing the algorithm numerous times.

3.3.2 Steering Logic

The partitioning algorithm is steered by the cost computed after each iteration. The factors that go into measuring the cost of a partition are varied in nature and specific to the user and the objective of the partitioning algorithm. In this work, the two factors taken into consideration cost measurement are 1) total amount data communication between partitions and 2) amount of workload imbalance. The amount of workload imbalance is used both as a constraint and a cost factor. A cap value for the imbalance is set by the user and all solutions exceeding the cap are rejected automatically. The solutions qualifying the imbalance constraint use both the amount of communication and the imbalance for cost computation. Combining the factors that contribute towards cost is fundamental to the proper evaluation and usage of the cost factors. In this work the cost factors are combined using the weighted geometric mean model since this describes the relation between the cost factors in a much accurate fashion than the traditional sum of products model.

$$Pcost = \sqrt{Communication^{(W1)} * Imbalance^{(W2)}} \tag{3.2}$$

In the above equation, W1 and W2 are weights assigned by the user. The work-load balance criteria is used as a constraint to prevent trivial solutions of dense clusters.

3.4 Architecture and Assumptions for Experimental Setup

The experimental set up used to verify the validity of the approach and estimate quality of the solution is explained in this section . The assumptions about the architecture under consideration are presented first. The actual set up for estimating various attributes values is then discussed.

The Architecture under consideration is a distributed heterogeneous system with n Processing Elements $(PE_1, PE_2, ...PE_n)$. All the components are connected together by means of a high speed network, with data transfer speeds known in advance. Delay encountered in bus arbitration is assumed to be negligible for the sake of simplicity. The framework has been designed with scope for scalability and inclusion of new parameters. Hence bus arbitration protocols and delays encountered in arbitration can be modeled into the existing framework with minimal modification to the original design. The partitioning algorithm would still remain unchanged. The communication latency here is the product of the capacity of the communication link with the frequency of usage of the link. Since most of the simulation was carried out on a sparcv9 sun-blade system operating at 650MHz in built with a sparcv9 floating point processor, the sparc processor would be referred to as the host processor. Some parts of the profiling was done in a PC , with Intel Pentium P4 1.3 Ghz processor running Microsoft Windows XP and a MacOsX system with a powerpc G4 1.1 Ghz and running OsX 10.4.11 - Tiger operating system, in order to obtain a generalized view estimate of the task attributes. The processing elements are assumed to have the capability to execute any of the tasks, although the time to take to execute a particular task might vary from one processor to the other. In the set of simulation experiments conducted the memory capacity was fixed. The framework also supports scalability to support changes in the total memory usage. Simple scalar tool set can be used for profiling memory usage and the results can be incorporated into the existing partitioning algorithm.

3.4.1 Computation of Attribute Values

The execution time on the host processor is obtained using the task graph extraction [19] tool. The assembly language listing of the source code for the sparc processor is obtained using *gcc* - GNU-C compiler. The number of CPU cycles that each task would occupy is obtained from assembly listing and the instruction set architecture information. The assembly listing on other processing elements is obtained using a *cross compiler*. A cross compiler tool chain can be constructed for a wide variety of target architectures supported by the gcc compiler. Once a tool chain has been compiled and created for a particular architecture, the cross compiler can be used to compile and generate assembly listing of any application for that specific target architecture . In this work a cross compiler tool chain was set up for the Intel Strong ARM processor series, Motorola M6811hc micro-controller series and the powerpc series. The host platform used was Solaris 2.8. The gcc version used was 2.95.3.

The tool computes the frequencies of each type of instruction for every task on all the processing element. The product of frequencies and the complexity of the instruction is the total amount of CPU usage cycles for a particular instruction type. The sum of all such products taken over all the types instruction would then result in the total number of CPU cycles that would be used by the task on that particular processing element. Although assigning a static complexity value is an approximation, it is a faster mechanism when compared to, actually computing the over head of each instruction individually and obtaining runtime counts. Additionally since the CPU cycles for all the processing elements are approximated and CPU cycles are used only as one of the metrics in modeling the behavior of the tasks, there are no disadvantages in approximation. Instruction level power characterization model for Strong ARM series of processors was done by [2] In [32] the authors have characterized the Intel Strong ARM series of processors and they estimate the amount of energy consumption in joules when each instruction is executed on the arm processor. For the Motorola M68HC11 series of micro controllers the energy values for the instruction set were obtained from [5] From the computation programs, having already obtained the assembly listing of the tasks, we can now compute the energy consumed by the task using the energy values provided by [32].

3.5 Experimental Results

The clustering algorithm was tested on five different benchmark programs of varying levels of communication. Table 3.1. denotes the number of nodes(tasks) and the edges present in each of the five benchmark programs chosen for simulation. In addition to testing

Table 3.1. Size of the Programs Used in Simulation

Program	#Nodes	#Edges
FFT	22	26
Statistical Tool1	44	97
Statistical Tool2	90	163
Laplacian Edge Detection	105	189
Arithmetic Coding	136	216

the clustering mechanism on five different benchmarks, for each bench mark the number of parameters taken into consideration by the clustering algorithm was varied in order to study the effect of varying number of parameters on the effectiveness and performance of the clustering algorithm. The constraints that were varied in order to test the performance of the clustering algorithm include the following: the execution time taken for each task to complete its execution, the amount data in KiloBytes(KB) needed to be transferred from the task to other tasks, the number of CPU cycles needed by the task to complete execution and finally the amount of energy needed for the task to complete execution. The default version of the clustering algorithm uses only two parameters, the execution time and the communication overhead from the task to the other tasks. When three parameters are used by the clustering algorithm, the number of CPU cycles used for the execution of the task is also included in addition to the existing two parameters. Finally the energy consumed by the task is also taken into account when four parameters are used by the clustering algorithm. The parameters were varied for each of the five benchmarks. The benchmarks chosen are a mix a of small/medium to large code size benchmarks. Hence the clustering algorithm was tested for three to seven clusters in the case of small to medium benchmarks and five to nine clusters in the case of large benchmarks. The size of the benchmarks were indicated by the amount of the edges present in the task graph of the benchmark.

Table 3.2. Simulation Results Obtained by Running FFT

Percentage improvement in communicationlatency for FFT					
Algorithm	Clustering Based Approach				
#Clusters	3	4	5	6	7
#Pi=2	62.8	41.7	21.2	17.0	14.2
3	31.8	30.7	23.1	16.9	16.9
4	36.6	34.9	25.4	17.1	15.6
	Simulated Annealing Based Approach				
#Pi=2	77.3	55.4	34.4	22.7	20.8
3	30.6	40.4	60.8	18.9	15.8
4	58.6	45.6	32.6	24.5	19.7
	Tabu Search Based Approach				
#Pi=2	43.2	5.0	7.3	5.7	7.4
3	31.5	12.5	13.1	4.6	8.4
4	34.5	7.0	13.9	12.2	7.8

Table 3.2. contains the results of executing the clustering algorithm for one of the smaller benchmarks the fast fourier transformation-FFT. Tables 3.3. and 3.4. contain the results of executing the clustering algorithm for the medium sized benchmarks statistical tools 1 and 2 . Both the statistical tools perform various statistical functions beginning with smaller ones including computing the correlation, variance, regression and also including some high level statistical functions containing complex computations. Both the statistical tools are computationally intensive applications and also require a large amount of communication bandwidth in order to transfer data and results from one module to another.

Table 3.5. contains the result of executing the clustering algorithm on image processing algorithm namely the Laplacian edge detection algorithm. Edge detection is also a computationally intensive benchmark. The communication bandwidth needed in terms of the KiloBytes of data transferred between the nodes is however lesser than that in the case of the statistical tools I and II. Table 3.6. is the final table containing the results of the execution of the clustering algorithm and is the largest benchmark that the algorithm was tested for. Arithmetic encoding of data files is both computationally intensive and also the amount of data transferred between the nodes is huge, hence the communication between the nodes is an important criteria in this case.

Table 3.3. Simulation Results Obtained by Running Statistical Tool 1

Percentage improvement communicationlatency for Statistical Tool1					
Algorithm	Clustering Based Approach				
#Clusters	3	4	5	6	7
#Pi=2	45.2	55.0	51.5	41.8	39.0
3	50.0	41.4	44.4	42.6	46.2
4	56.5	55.7	47.1	40.0	43.6
	Simulated Annealing Based Approach				
#Pi=2	60.1	67.9	59.8	49.1	44.3
3	57.6	45.8	49.2	50.7	53.1
4	63.6	65.9	53.3	48.3	51.9
	Tabu Search Based Approach				
#Pi=2	45.5	55.8	40.1	41.9	31.3
3	38.0	36.1	32.7	16.8	18.7
4	37.2	36.7	32.4	18.9	18.3

The number of constraints used for each execution run of the clustering algorithm is represented in Table 3.7.. The results are indicative that in the case of the chosen benchmarks as the number of constraints used by the clustering algorithm increases the amount of savings in terms of the overall optimization in communication over head reduces.

The results from the above table also indicate that the parameter number of clusters also has a negative effect on the clustering algorithm. Increasing the number of clusters in the application increases communication overhead and hence the reason for the decrease in improvement of communication bandwidth. The cost evaluation function developed for the clustering algorithm was unidirectional and the primary goal was to optimize the total communication over head between the clusters. However the cost function can be modified according user preferences to alter the objective of the algorithm and the same partitioning algorithm could be tweaked to accommodate simultaneous optimization of execution time and power consumed. The user has to maintain caution that introducing complementary terms in the cost function might result in the case where the algorithm is unable to find an optimal solution and the the solution obtained may be a local maxima/minima.

Table 3.4. Simulation Results Obtained by Running Statistical Tool2

Percentage improvement in communicationlatency for Statistical Tool2					
Algorithm	Clustering Based Approach				
#Clusters	5	6	7	8	9
#Pi=2	65.8	68.8	65.5	48.6	44.6
3	61.7	48.8	43.7	35.8	35.8
4	31.8	28.5	32.8	30.2	28.5
	Simulated Annealing Based Approach				
#Pi=2	70.6	69.8	68.7	73.1	71.4
3	66.1	62.8	63.8	60.8	59.8
4	71.4	65.8	64.1	57.9	55.4
	Tabu Search Based Approach				
#Pi=2	60.5	53.6	39.6	40.1	35.3
3	50.2	42.3	33.4	13.9	13.2
4	21.9	19.2	24.9	11.2	9.8

3.5.1 Comparison of Cumulative Results

The code partitioning was implemented with simulated annealing based algorithm to estimate the performance. A Tabu search based partitioning algorithm was also implemented for the case of comparing clustering with the other methods. In the computational results in the tables above we have seen that simulated annealing based approaches have performed better than clustering based approaches and they have also performed better consistently by a margin of about 10-15 % The Initial temperature for the simulated annealer was determined by finding the average change in cost for a set of random moves from the starting configuration and selecting the temperature which leads to an accept probability of 0.95 [29]. The tabu search algorithm implemented was also adapted from [28]. We can see from the results that clustering based approaches outperform tabu search based partitioning algorithm in most of the cases. Considering the over head involved in executing each of the separate approaches the first factor compared is the execution time needed for the completion of each approach. In Table 3.8. a comparison between the execution times for a clustering based approach and a simulated annealing based approach is presented.

Table 3.5. Simulation Results Obtained by Running Laplacian Edge detection

Percentage improvement communicationlatency for Laplacian Edge detection					
Algorithm	Clustering Based Approach				
#Clusters	5	6	7	8	9
#Pi=2	67.1	69.9	64.9	50.3	47.1
3	62.5	57.7	53.1	40.5	37.9
4	35.2	32.1	33.2	31.3	27.8
	Simulated Annealing Based Approach				
#Pi=2	72.1	73.4	72.2	71.5	66.9
3	68.9	65.1	64.2	55.7	54.3
4	66.2	60.3	58.7	50.4	47.2
	Tabu Search Based Approach				
#Pi=2	64.1	59.6	59.6	43.5	40.7
3	51.3	50.7	49.8	23.7	20.8
4	24.4	23.6	26.1	10.3	8.9

Observing from Table 3.8., we see that the comparison of execution times between the algorithms is not practical. simulated annealing based approaches explode in time complexity with increasing number of nodes in the application. The rate of increase of execution in comparison with the rate of increase of number of nodes in the application is very high in the case of simulated annealing based approaches. Tabu search based partitioning schemes are also increasing at a high rate when compared to clustering based algorithm. In some cases simulated annealing based approaches do not converge to an optimal solution within an acceptable time frame.

Table 3.6. Simulation Results Obtained by Running Arithmetic Encoding of Files

Percentage improvement communicationlatency for Arithmetic Encoding of Files					
Algorithm	Clustering Based Approach				
#Clusters	5	6	7	8	9
#Pi=2	68.9	70.3	65.6	51.9	50.5
3	63.6	63.4	59.7	44.2	40.3
4	38.7	36.3	34.6	30.8	28.1
	Simulated Annealing Based Approach				
#Pi=2	74.5	76.7	75.8	67.1	66.9
3	70.6	69.3	68.5	59.4	54.6
4	58.5	55.1	53.8	43.6	40.7
	Tabu Search Based Approach				
#Pi=2	66.3	64.7	61.1	44.9	42.4
3	53.2	52.2	52.7	27.7	24.5
4	27.1	25.7	26.8	11.1	10.2

Table 3.7. Constraints Terminology

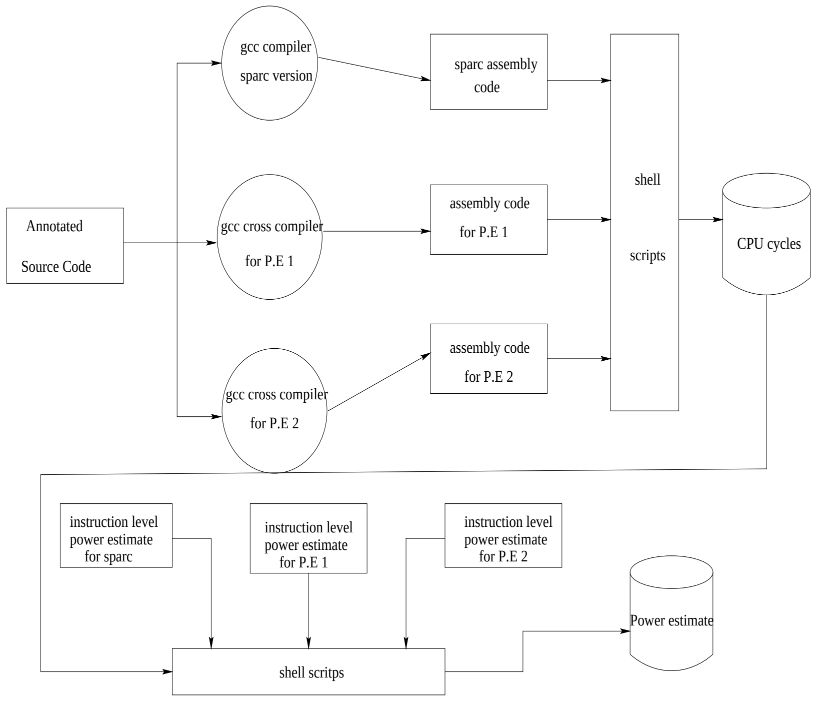
#Constraints(P_i)	Constraints Used
i=2	ExecutionTime+DataSentout
i=3	P_2 + CPUCyclesUsed
i=4	P_3 + PowerDissipated

Table 3.8. Comparison of Execution Times for Clustering, Simulated Annealing and Tabu Search Based Approaches

Nodes	Execution time for one iteration		
	Clustering	SA based(hours)	Tabu Search
22	20 sec	4:49:16.85	14
44	35 sec	6:38:37.44	82 sec
90	1:05 min	15:23:7.34	1:25 min
105	2:05 min	17:15:43.8	3:25 min
136	3:22 min	20 hr	5:45 min

CHAPTER 4
SOFTWARE ARCHITECTURE

The software implementation of the partitioning algorithm can be divided into two parts. One part comprises of the profiling tools used for gathering information from the application source code and the other part is the *C++* language based object oriented design and implementation of the algorithm. This chapter elaborates on the profiling set up and then discusses the detailed object oriented design for C++ software architecture. The first step of the software implementation process is the profiling set up. The specific characteristics of the application source code are analyzed and the values for each of the attributes of every node in the task graph is computed in the profiling process. The frame work constructed for the profiling process consists of compilers, custom designed CAD tools and third party instruction level simulators. The different tools are tethered together in a tight process flow described in the design of the frame work. The entire frame work is automated and requires no human intervention. The first input to the profiling process consists of the application source code containing annotations to demarcate task boundaries. The second input to the profiling process is a hardware library file containing specific details of the target architecture on which the application needs to be profiled. In the case of heterogeneous distributed systems, there are multiple architectures involved. Hence the hardware library file usually contains architecture specific details for the all the components in the heterogeneous system in addition to the connectivity information between the components, like the topology of the distributed architecture and the speed of transfer on these communication links.



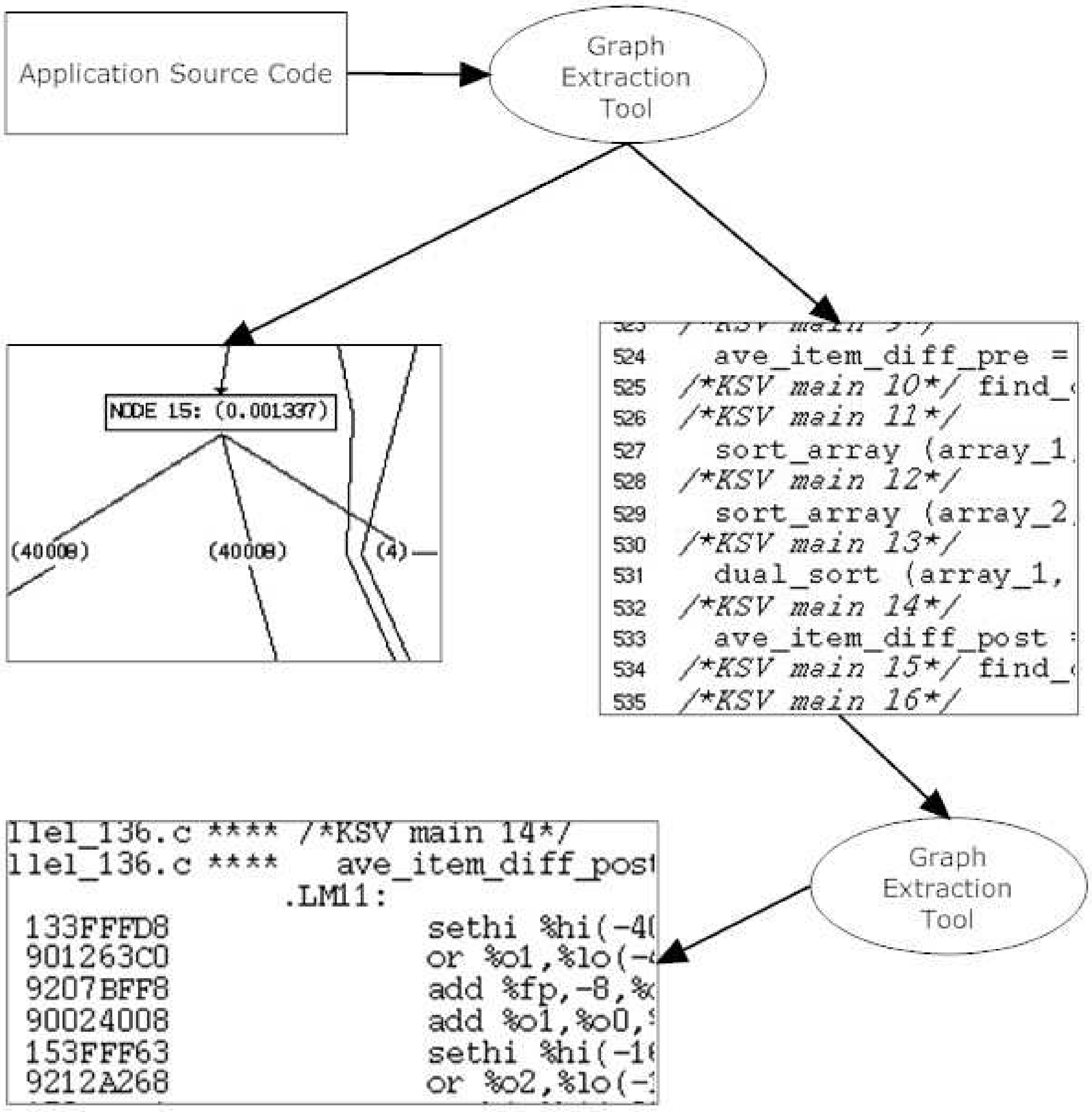
Annotated Source Code
gcc compiler sparc version
gcc cross compiler for P.E 1
gcc cross compiler for P.E 2
sparc assembly code
assembly code for P.E 1
assembly code for P.E 2
shell scripts
CPU cycles
instruction level power estimate for sparc
instruction level power estimate for P.E 1
instruction level power estimate for P.E 2
shell scritps
Power estimate

Figure 4.1. Application Source Code Profiling Tools

4.1 Profiling Setup

The profiling framework is described in the figure 4.1.. The tools used in the profiling process can be divided broadly into two categories. Cross compilers and shell scripts processing the cross compiler output.

Cross compilation is the process of compiling an application source code on a host architecture with the object of generating a binary capable of executing on a target architecture. Cross compilation process is especially useful in the case of embedded systems where the target architectures usually do not have any support for operating systems or compilers. A cross compiler tool chain is built for any specific target architecture on the host processor. The process is elaborated in detail in figure4.4.. The basic tools needed for building and installing a cross compiler on any machine consist of the following. *gcc* a GNU based C compiler forms the base for compiling libraries and binaries on the host machine. The *gcc* on the host machine is used in building and installing the cross compiler



```
/*KSV main 9*/
  ave_item_diff_pre =
/*KSV main 10*/ find_
/*KSV main 11*/
  sort_array (array_1,
/*KSV main 12*/
  sort_array (array_2,
/*KSV main 13*/
  dual_sort (array_1,
/*KSV main 14*/
  ave_item_diff_post =
/*KSV main 15*/ find_
/*KSV main 16*/
```

```
llel_136.c **** /*KSV main 14*/
llel_136.c ****    ave_item_diff_post
                 .LM11:
 133FFFD8                sethi %hi(-4
 901263C0                or %o1,%lo(-
 9207BFF8                add %fp,-8,%
 90024008                add %o1,%o0,
 153FFF63                sethi %hi(-1
 9212A268                or %o2,%lo(-
```

Figure 4.2. Illustration of Sample Code at Various Stages

for the target architecture which would be called the *cross-gcc*. The other tools required in the process include assemblers, linker combined with loaders and C-libraries. The last requirement for the construction of the cross compiler is the target architecture specific C language headers. C language header files are readily available for download for most commonly known target architectures in the embedded domain and they are also available for commonly used operating systems like linux, solaris etc. Most of the third party vendors shipping embedded processors include cross compilers and design tools for development for the architectures along with the processor. GNU is an open source community initiative, hence header files and libraries for most of the existing versions of the compiler can be obtained from free lance programmers and research groups organized into forums. After the build, the cross compiler is tested for correctness using simple bench marks. The application source code is then compiled on the host processor with the newly built cross compiler. The binary file obtained can be transferred to the target architecture for execution. The binary can also be used for obtaining profiling information. The binary file for the target architecture is passed through a set of instruction set simulators to obtain cycle count information. The instruction set simulators can be used to generate other profiling information including memory usage, power dissipation, cache usage patterns etc without actually executing the binary on the target architecture. This method of rapid prototyping is particularly useful in the design of distributed heterogeneous systems. The system architect has the opportunity to evaluate the performance of various configurations for the distributed system.

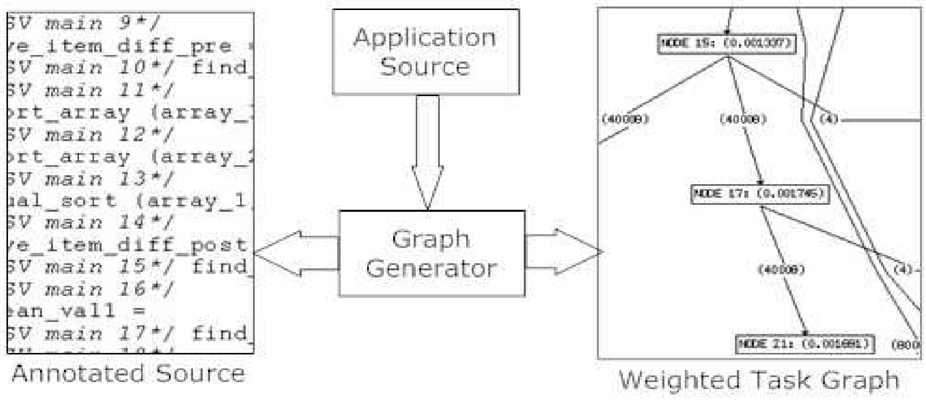
Cross compilers provide a convenient method to profile and test any prototype design that is designed for any processor (*especially embedded processors*) on the host processor. Design issues and performance bottle necks can be fixed in the prototypes even before physically testing on the intended processor. The primary advantage of this process is the huge amount of time saved in the test of the prototype. Once a single cross compiler tool chain has been set up for a target architecture, the tool chain can be used for compiling and generating executable files for the target architecture on the host architecture itself.

This process eliminates the need for transfer of the executable on to the target architecture for compilation or simulation purposes. Another advantage of the method is that, there is now scope for investigating into the behavior of the application with respect to the target architecture. Hence the architecture can be fine tuned to optimize the execution of the application. The above advantages proved to be the main motivation in the use of cross compilation techniques.

In this work all the experiments were design to be conducted on a sun SPARC IV processor. Cross compilers were constructed for three other architectures ARM processor for a linux based system, power PC processor for a linux based system and finally *m68hc11* a micro-controller. Inter ARM processor is a powerful embedded processor supporting complex instruction types. Power PC is a general purpose processor capable of a high degree of programmability. M68HC11 is a Motorola embedded micro controller with very limited level of programmability. The micro controller also supports few instruction types. Hence the combination of the processors chosen represents a good mix of processing elements with varying degrees of programmability and performance. The cross compiler is used compile the application source with specific compilation switches turned on. The gcc compiler can be used for generating debugging information about the application code being compiled. The combination of runtime options used in invoking the compiler decides what switches are turned on. The gcc compiler can generate assembly code for the target architecture from the application source. Hence using a cross compiler the application source code can be compiled with specific switches turned on so that the application source code is profiled for the target architecture. Application source code compiled with an ARM cross compiler built on a solaris machine can thus produce ARM assembly language code on the solaris machine. The generated assembly code can then used along with instruction set simulators to obtain further information about the application when particularly executing on the target architecture platform. The problem with this technique is instruction set simulators are usually tailored for complete programs and hence do not provide information pertaining to individual tasks. In order to over come this problem, the original application source code is passed through the graph generator. The result is modified application source with comments added in the code demarcating each task in the application source with specific task numbering. The cross compiler generates the assembly listing for the application source for each of the target architectures. The application source is now converted to assembly listing for each of the target platforms. The output of the cross compilation process is the application source code with annotated assembly language code for each instruction in the application source code. A data sheet is provided by the vendor for every target architecture that contains information about the instructions supported by the target architecture. From the support for the processor target architecture, it is easy to compile a library of all the instruction supported by the target processor and the average number of CPU cycles used for each instruction type.

The process of estimating the number of CPU cycles used by each task uses three inputs, annotated original application source code, annotated and cross compiled assembly listing of the application source code and finally the data sheet for all the target architectures that contains the cycle count information for different categories of instruction supported by each target architecture. The first step in the process of extraction the CPU cycles is the first pass of a parser on the annotated source code to identify the task numbers. The second step is a first pass on the assembly listing of the source code to identify corresponding numbers in comments. On identifying a numbered task the profiler cross references the data sheet for the target architecture to identify the number of the cycles used by the particular instruction and is maintained in a buffer. Since most applications are written using high level languages supporting high level design techniques like procedures and functions, there may remain undefined references in the assembly listing for which the profiler is unable to find a match in the target architecture data sheet. The function/procedure calls written in high level languages are usually stored as symbols for in a function table at the end of the main application program, in order to facilitate a jump to the address of the function tag on a procedure call routine. The profiler identifies the function/procedure names at the end of the first pass on the annotated application source code. The CPU cycle count information for each of the procedures is calculated by a set of secondary passes into the application source code and a unique file is created for each procedure definition and the cycle count information for that procedure is stored in that file. The profiler then makes a second pass on the assembly listing while maintaining the list of undefined references. This time when a function/procedure name is encountered, the CPU cycles used for that particular instruction is substituted from the separate list of files for each function call. At the end of the second pass the buffer for each numbered task contains the list of all the instructions comprising the task and the average cycle count for that particular type of instruction. The contents of each buffer are cumulatively added to obtain the total number of cycles needed for each task. Similarly a data sheet is compiled for all candidate target architectures that would be used in the distributed heterogeneous system. Cross compilers are built and installed for all the target architectures. The above process of computing the CPU cycles for each task is then repeated for each target architecture with the corresponding cross compiler tool chain. The results are collated and arranged in text buffers for fast access by the partitioning algorithm.

The key design considerations in the implementation of the profiling module include speed of execution and level of automation. The module should be really fast to accommodate real time speeds and the amount of human intervention should be kept at a minimum possible. Most of the high level programming languages are fast in the case of the computationally intensive applications. But when it comes to reading and writing huge amounts of data from disc files, high level programming languages are really slow. Hence powerful shell programs or *shell scripts* written using a combination of *csh scripting*, *gawk* and *sed* are used in the static profiling process. *Gawk* is the GNU version of the original UNIX programming language *awk* with some advanced functionality added. Gawk contains many of the string manipulation functions supported by the C programming language and in addition most of the C language constructs can be used within a gawk script. Since gawk is typically a scripting language, it is faster than high level programming languages especially for string/data manipulations from *streams*. A *stream* in this context can be defined as a stream of data that is obtained as output from one process or gawk command. Gawk scripting language and shell scripting languages like csh can operate efficiently on stream outputs and inputs. The output from one shell process can be directly streamed as input for another shell process. without actually being written onto any temporary disc files. The advantage of using streams is that the input/output wait involved in writing to temporary disc files is avoided. *Sed* - stream editor is ideally suited for operations on huge amount of data stream and is also exceptionally fast. The process of compiling and building each of the individual cross compilers is explained in detail in the appendices.



```
SV main 9*/
ve_item_diff_pre
SV main 10*/ find_
SV main 11*/
ort_array (array_
SV main 12*/
ort_array (array_
SV main 13*/
ual_sort (array_1
SV main 14*/
ve_item_diff_post
SV main 15*/ find_
SV main 16*/
ean_val1 =
SV main 17*/ find_
```

Figure 4.3. Illustration of Output from Task Graph Generator

4.2 Sparc-Linux-Arm Cross Compilation

This section provides in detail the scripts and packages needed for compiling and construction a cross compiler for an ARM processor running linux operating system on a Sparc processor running the solaris operating system. The building blocks required for constructing the compiler are the following.

- *binutils*- a collection of GNU binary tools containing *ld* the linker and *as* the assembler
- *glibc*- a library defining C language system calls for the kernel
- *linux-headers-arm*- ARM architecture specific kernel header files

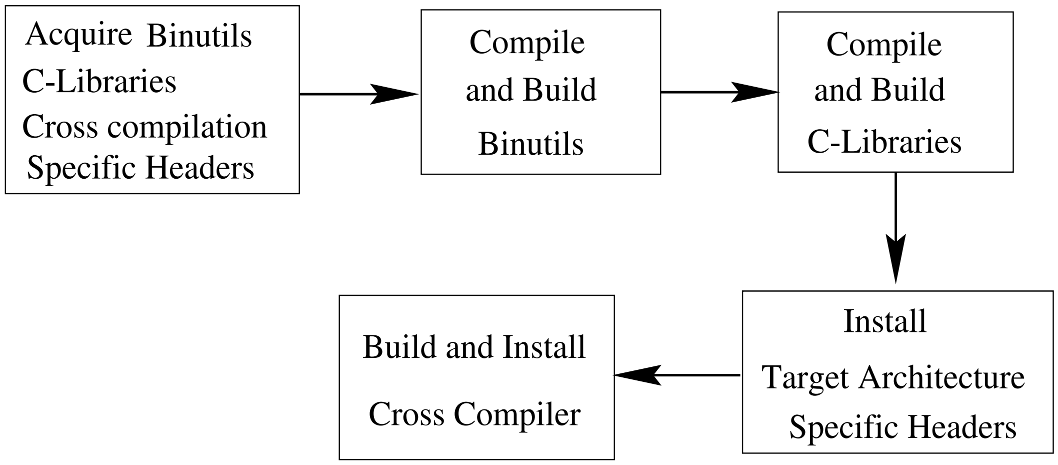


Figure 4.4. Overview of the Process of Building a Cross Compiler

The above tools can be downloaded from www.gnu.org or from forums promoting research in the ARM-linux cross compilers. The tools which are in *bzip* format can be extracted using a *bunzip* command. Some tools are bundled in *RPM* (Redhat linux Package Manager) files. In order to extract RPM files on a solaris operating system, a small *PERL* script *rpm2cpio* can be used. The PERL script converts the RPM packages to solaris archives file archives. The file archives can be now be extracted using the command *cpio*- copy file archives in and out. The next step after obtaining the tool packages is fixing the directory to install. Approximately about 45MB of free space is required for the installation. A working version of the GNU C compiler *gcc* is required. The version used for this cross compiler was gcc 2.95.3. The binutils need to be installed first in order to construct the compiler. The first step in installing the binutils is configuring the installation script. The source files directory contains a shell script *configure*- this file contains place holders for source file locations, library locations, installation location and installation options. The source files can be configure to build a cross compiler for a little-endian or a big-endian architecture. The place holders are completed and the configure script is executed. The script performs a check on the availability of all source files, read and write permissions for directories that would be used in the installation process and amount of free space on the disk. If all the conditions needed to build the compiler are satisfied, the script generates an installation script file. The installation script can the executed by using the commands *gmake* and *gmake install*. The command gmake compiles the source files and builds the libraries. The command with the install option links the library files, generates binaries as output and copies the binaries to their respective directories. The next step is to build the actual cross compiler using the *ld* and *as* tools which have been built.

4.3 Motorola m68hc11 Cross Compilation

The m68hc11 is a Motorola embedded controller with limited functionality. The advantage of using m68hc11 is that it is easy to program with a limited set of supported instruction types. The forums available on gnu.org provide open source tools and software for development on the m68hc11 and m68hc12 platforms. The bundled software can be downloaded and installed to get a working version of the gcc compiler, the glibc libraries, binutils and m68hc specific header files. The version of compiler used in this work was gcc 2.95.3. The software was installed on a PC running Microsoft windows XP. The cross compiler generated m68hc11 assembly code annotated with task numbers. This assembly listing was transferred to the sun Sparc machine running solaris 8. The profiling tools integrated into the framework then parsed the assembly listing to obtain specific information about the number of CPU cycles used for each task and other details including memory usage and power usage.

4.4 Power PC Cross Compilation

The Power PC processor is a generic processor capable of executing complex instructions. The cross compiler for Power PC was set up on the Sun Sparc machine and the assembly listing was generated. The profiling of tasks was done on a powerbook G4 running MacOsX 10.4.11. The X-Code framework was used to profile the application. The *Shark* performance analyzer was used to generate task related information.

4.5 *C++* Based Implementation

In this section we describe in detail the *C++* programming language based software framework developed for implementing the partitioning algorithm. The software framework can be divided into two major parts. The first part is the entire set of objects that comprise the framework. This part includes object behavior modeling using attributes and methods. The second part contains generic functional routines for partitioning and graph traversal. Since the code partitioning problem has been mapped to a graph partitioning problem, graph traversal and partitioning algorithms are implemented and can be invoked independent of the nature of the objects. The advantage of this development methodology is the flexibility provided to the user in deciding the exact partitioning algorithm that the user prefers to use. Partitioning algorithms based on clustering, simulated annealing and tabu search approaches have been implemented in this work. The framework has been designed in such a manner that any new partitioning algorithms that need to be utilized, can be invoked using API's provided in the object framework.

4.5.1 Motivation for Using *C++*

The *C++* programming language was the ideal choice for the development of the entire software architecture framework. The language is object oriented hence provides data hiding or encapsulation mechanisms. The behavior of all the individual components of the framework are modeled using objects. Hence all the operations pertaining to each of the individual components are encapsulated into the object itself as *methods.* The various partitioning algorithms were implemented as generic methods. Since the implementation of the objects is transparent, the user can switch from one partitioning algorithm to another without any programming change. In addition, the framework can also accommodate additional partitioning algorithms with minimal changes to the existing framework. Once the partitioning algorithm is implemented, just addition of another choice will enable the partitioning algorithm to utilize the existing API's of the framework in order to integrate into the system.

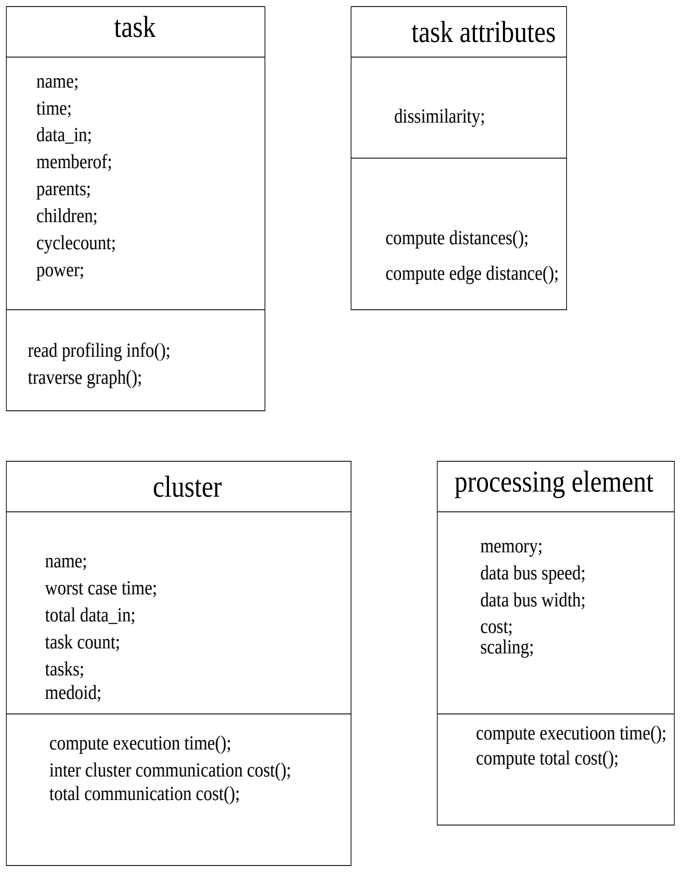


Figure 4.5. Object Based Design of the Main Entities

4.5.2 Object Descriptions

The individual entities in the partitioning problem exhibit behavior similar to objects. In this section we describe in brief the main objects that form the core components of the software framework. The *task* object represents the model of a task in the application or equivalently a node in the task graph. The attributes of the task class include the following: the name of the task (usually node number), the time taken in seconds to complete the execution of the task , the amount of data (in KB) needed by the task for completion, the id of the cluster to which this particular task is a part, list of all the nodes having an directed edge with their end point on this particular task (or parent nodes), list of all nodes to which there are directed edges initiating from this particular task (or child nodes), the total number of CPU cycles that the task would need on each processing element, and the amount of power needed to execute the task. The object encapsulates methods for providing complete access to the task graph. These methods include:

- *Parentnodelist* $\Rightarrow$ Obtaining a list of all the parent nodes
- *Childnodelist* $\Rightarrow$ Obtaining a list of all the child nodes
- *Distance* $\Rightarrow$ Computing the edge distance to another task node (if connected)

The object also contains methods to interact with the library of processing elements and profile information repository to obtain the number of CPU cycles and power consumption. The information pertaining to each task is stored in the corresponding object. The individual attributes of each task are then normalized and converted to dissimilarities and stored in the task attributes class.

The object *cluster* is representative of a cluster of tasks. The attributes of the object include the following: name of the cluster, total amount of data incoming data (in KB), worst case execution time for the cluster obtained by comparing the execution times for the same cluster on all processing elements, total number of tasks comprising the cluster, the node numbers of all tasks grouped with this particular cluster and the node id of the medoid for this particular cluster. The object provides methods for traversing the task graph and computing the execution time for any sub-graph of the entire graph. Methods are also provided for computing the cost of data transfer between a pair of clusters represented by the amount of time taken to transfer data (measured in seconds) and also in addition to compute the entire cost of a solution measured as the total amount of time spent in data transfers.

4.5.3 Tree Traversal and Cost Computations

The intermediate solution at each step of the partitioning process is a set of clusters of code. Each code cluster consists of inter connected tasks. The steering logic of the partitioning approach produces the intermediate solution. If the cost of the new intermediate solution is better than the previous, the cluster configuration is updated with the new set of medoids. The partitioning algorithm is then steered in the new direction. The cost computation module plays a vital role in deciding the new direction of progress at every intermediate step. The software framework consists of tree traversal algorithms to identify the member nodes for every medoid in the new solution. Once the cluster medoids and the member nodes are identified, the methods of the aggregate object *cluster* are utilized to compute attributes like total execution time, total number of CPU cycles and total energy consumption of every cluster on all the processing elements. At the end of this estimation process each *cluster* object contains all the attributes on the different processing elements. The comparison of the cost of one solution to another is thus simplified by a huge margin, since the implementation of the entire object is transparent to the user.

4.5.4 Building Blocks

The entire software framework is constructed with a lot of scope for expansion and scalability. Since all the major components of the system are modeled as objects, additional behavior the components can be easily be captured by the addition of new attributes to the object. The programming changes needed to model the behavior are minimal, since the partitioning algorithm and the cost computation modules are isolated from the implementation of the object itself. Further investigation into newer partitioning methodologies or including any improvements to the cost computation module is extremely simplified. The algorithms can be implemented separately and the API's provided with the objects can be invoked in order to switch the algorithms. The entire architecture was designed with scope for further improvements and expansion.

CHAPTER 5
CODE GENERATION

This chapter presents an automatic code generation tool for partitioned procedural language applications without making any changes to the original application source code. The tool does not involve any manual intervention. The partitioned code clusters are identified before hand. Any form of communication between the partitioned code clusters is then achieved by specialized communication primitives inserted into the original source code. This automatic approach to code generation has significant potential. This approach drastically simplifies the process of partitioning applications for heterogeneous distributed systems. This tool has been implemented as a part of a masters thesis [27].

The design and implementation of the proposed code generation tool is shown in Figure 5.1.. The first step involves gathering data and the creating a metadata information repository. This is the most crucial stage as the accuracy of the code generator depends on the data accumulated. The second step is the actual partitioning of the program into individual clusters of code and the addition of programming constructs to convert the clusters into standalone programs. The third step involves the inclusion of a communication mechanism. Data communication and synchronization are implemented via message passing. Optimization methods used to improve the quality and performance of code such as using multicast or broadcast communication, using aggregate communication and elimination of unnecessary communication are also explained in detail.

5.1 Need for a Data Repository

In order to attain synchronization and correct execution of the independent sub-programs of a distributed application, it is critical to maintain accurate communication between the

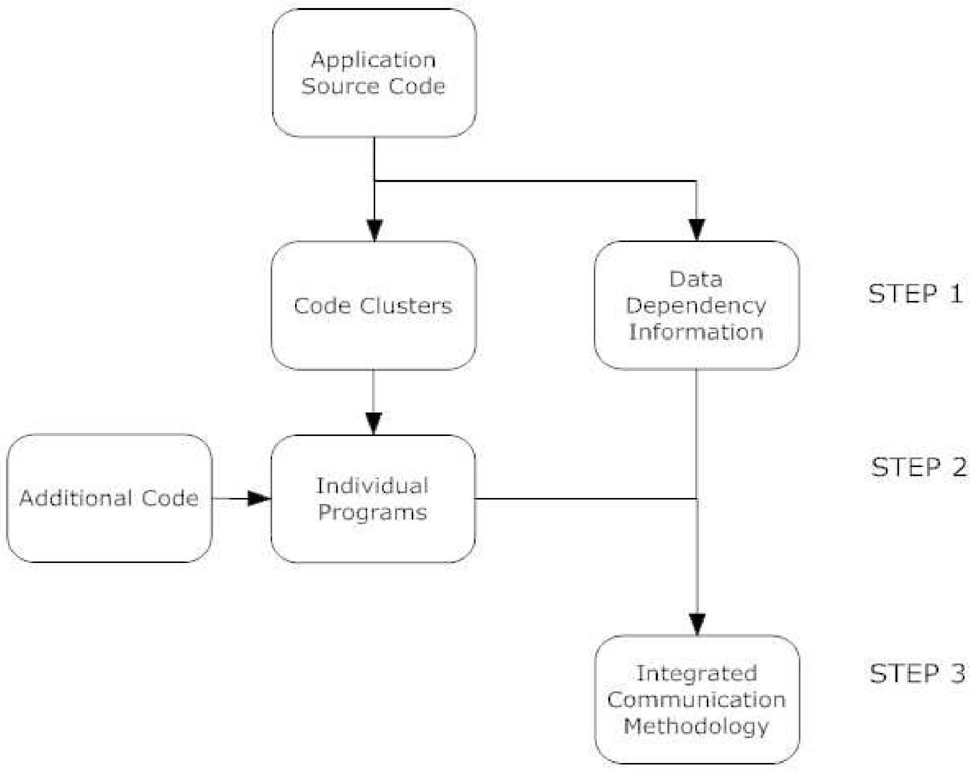


Figure 5.1. Steps Involved In Code Generation

code clusters. A strong communication methodology is necessary for the scheduling the instructions in the independent sub-programs. In order to implement this communication layer, detailed profiling information of the source code is necessary. Some of the factors that influence the scheduling of the individual sub-programs include

- Control Flow
- Data Dependency
- Software Functionality

These details are not available from the code itself. Subsequently, reverse engineering is applied to acquire this data about the program. The importance and resourcefulness of the reverse engineering process is explained in further detail in the following subsection.

5.1.1 Reverse Engineering

Reverse Engineering [13] is the process of extracting information from the existing software system using a bottom-up approach. During this process, the source code is not altered; although additional information about it is generated. The subject software system is represented in a form where many of its structural, functional and behavioral characteristics can be analyzed.

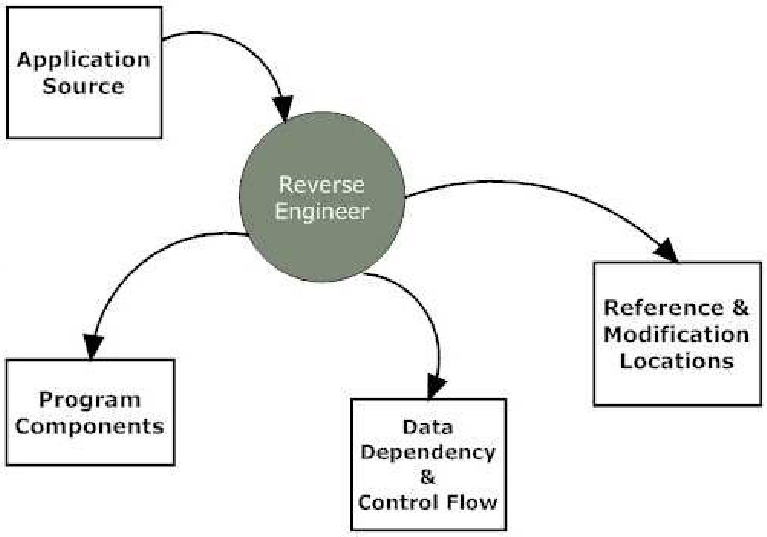

Application Source
Reverse Engineer
Reference & Modification Locations
Program Components
Data Dependency & Control Flow

Figure 5.2. Reverse Engineering Process

The source code contains all the information needed about the entities used in it. In order to extract the information about the entities used in it, the source code is passed through the reverse engineering process[36]. Relevant information obtained from the process include details such as:

- Identification of the entities used in the code.
- Modification locations.
- Reference Locations.
- What entities do they depend on?
- What other entities depend on them?

5.1.2 The "Understand C" Tool

In order to gather functional and behavioral information of the application that needs to be partitioned, we reverse engineer it using the UNDERSTAND C Tool developed by Scientific Toolworks, Inc. [33]. Several other tools like inSight, Essential Metrics and Imagix were tested and found unsuitable. The "Understand C" tool was preferred because of its ease of use, speed of generating reports and accuracy.

5.1.2.1 Functionality and Utility of Tool

The UNDERSTAND C tool analyses the application code to identify the program components and their interrelationships and creates a higher level representation of the software program that is easily understandable. It uses a graph model to visualize the flow of the code. The application to be reverse engineered is given as input, the RE tool profiles the code creates a complete documented report giving intricate details of the application at hand.

This tool gives two basic reports. The first contains a list of the program components i.e. a list of all the variables and functions with complete details of their data types. The second report contains a list of the locations where the program entities where referenced or modified. The control flow and the data dependency of the software program are also illustrated by the tool.

5.2 Creation of Metadata Table

The Code Generation tool proposed requires knowledge about the working of the application, before it can be distributed over several processors. This information includes the data dependency, control flow and list of program components. Although the reverse engineering tool provides a multitude of information about the application, there is a need to glean out and store appropriate information that will aid the distribution. This is done with the help of maintaining a Metadata table.

The steps involved in the creation of the Metadata table are as follows:

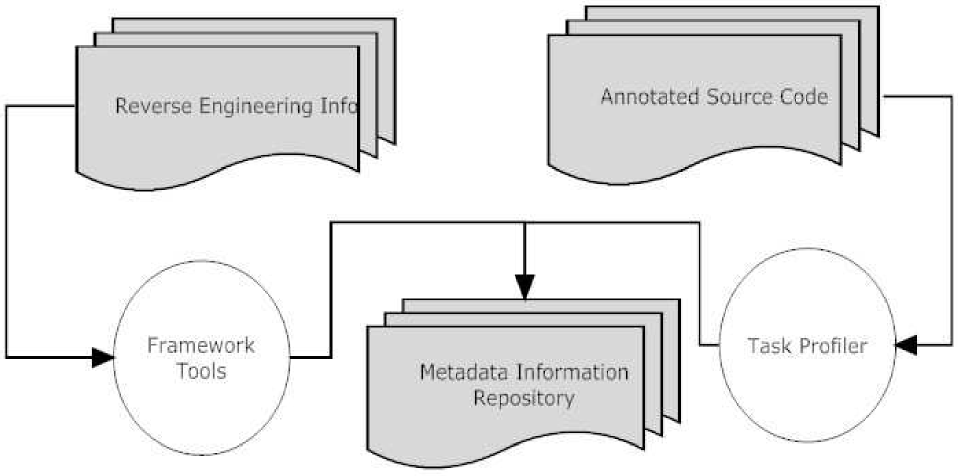


Figure 5.3. Creation of Metadata Table

- Output from reverse engineering tool is provided as input for the Framework.
- Modification locations of program entities are tabulated.
- The annotated source is passed through task profiler.
- The information from the two previous profiling steps are combined.

This process is illustrated in figure 5.3..

At the end of the preprocessing step the metadata table contains the following:

- List of program entities
- Reference locations
- List of tasks
- Entities used in each task
- Dependency information

Although the creation of a Metadata table accounts for an overhead, this process is required to be performed just once. Thereafter, distributing the application by varying the number of clusters does not require this process to be performed again.

5.3 Partitioning Program into Independent Sub-programs

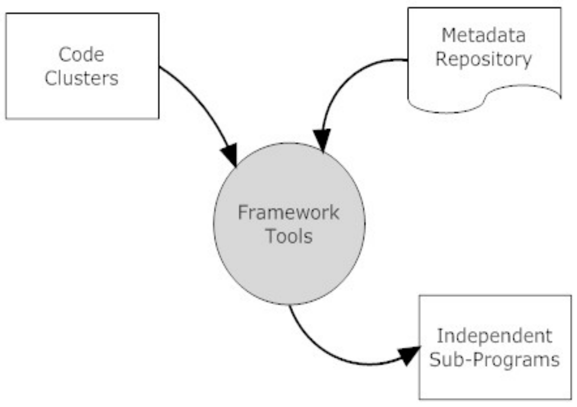


Figure 5.4. Clustering

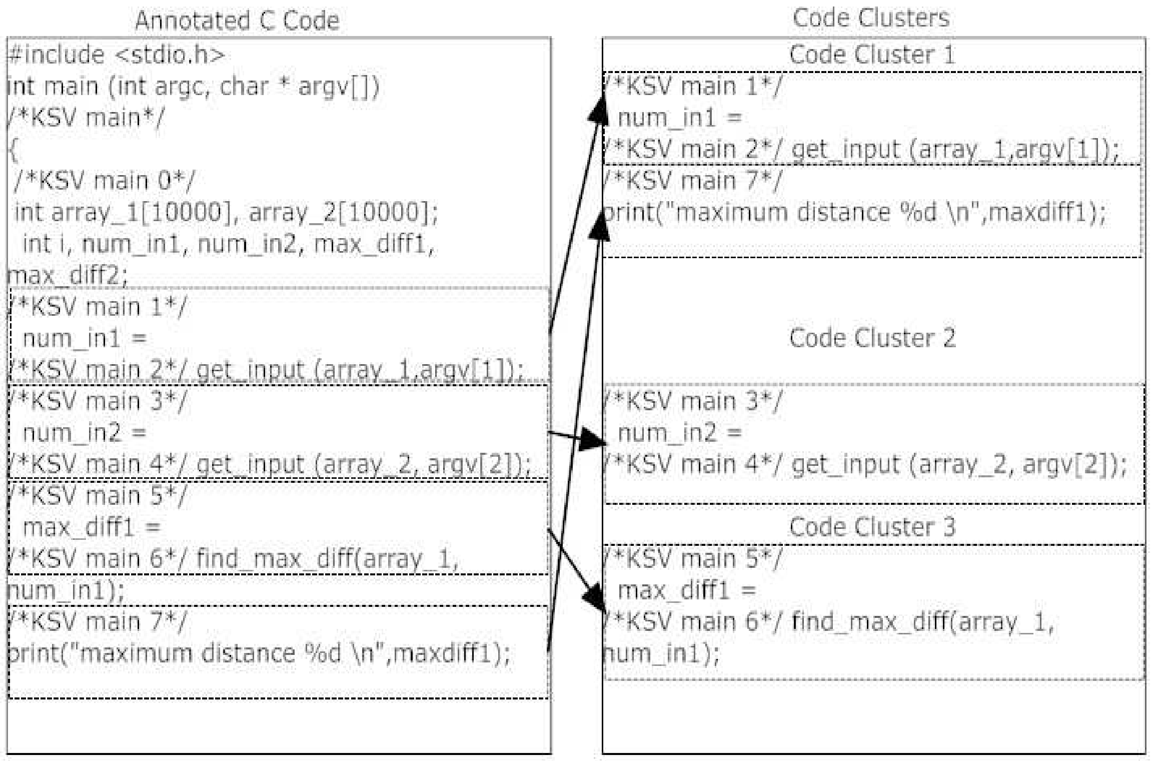
The annotated C-Language application is parsed though a separator program and partitions are created using clustering. These partitions or blocks of code are called code clusters. In order to execute the code clusters on specified processing elements, it is necessary to convert the code clusters into independently executable programs. An Illustration of creating independent subprograms is provided in figure **??**. The process of clustering and program completion is explained in following sections.

5.3.1 Clustering

The annotated application source code is passed through the task profiler. Each cluster is a group of several tasks.

We will refer to an example shown in Figure 5.5.. In the example the application is partitioned into three clusters:

- Code Cluster 1 containing Task 1 and Task 7
- Code Cluster 2 containing Task 3
- Code Cluster 3 containing Task 5



```
#include <stdio.h>
int main (int argc, char * argv[])
/*KSV main*/
{
 /*KSV main 0*/
 int array_1[10000], array_2[10000];
  int i, num_in1, num_in2, max_diff1,
max_diff2;
/*KSV main 1*/
 num_in1 =
/*KSV main 2*/ get_input (array_1,argv[1]);
/*KSV main 3*/
 num_in2 =
/*KSV main 4*/ get_input (array_2, argv[2]);
/*KSV main 5*/
 max_diff1 =
/*KSV main 6*/ find_max_diff(array_1,
num_in1);
/*KSV main 7*/
print("maximum distance %d \n",maxdiff1);
```

```
/*KSV main 1*/
 num_in1 =
/*KSV main 2*/ get_input (array_1,argv[1]);
/*KSV main 7*/
print("maximum distance %d \n",maxdiff1);
```

```
/*KSV main 3*/
 num_in2 =
/*KSV main 4*/ get_input (array_2, argv[2]);
```

```
/*KSV main 5*/
 max_diff1 =
/*KSV main 6*/ find_max_diff(array_1,
num_in1);
```

Figure 5.5. Code Cluster Formation

When partitioning is dynamic, the user can select the tasks that need to be grouped in the same cluster as well as the number of clusters that the application is to be partitioned into.

5.3.2 Adding Constructs

The basic goal is to convert the code clusters into independently executable programs. This requires adding constructs like declaration information and functions to the code clusters.

A naive approach to adding constructs would be to simply duplicate the functions in all the partitions. However, as the number of sub-programs increase (the number of partitions in which the program is distributed), the code size would explode.

The approach used in this work is to obtain certain information about the tasks included in each partition. Then, using the metadata table, find the corresponding entities and

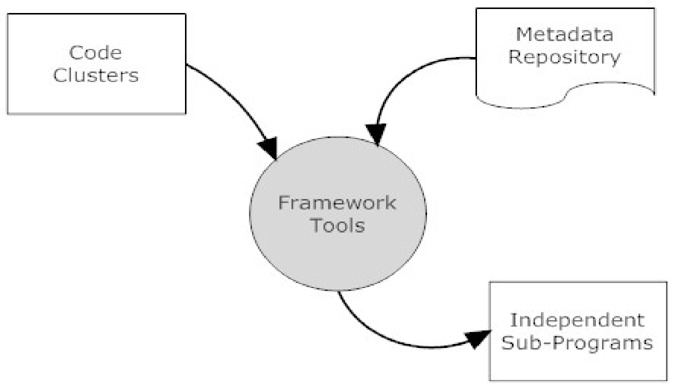


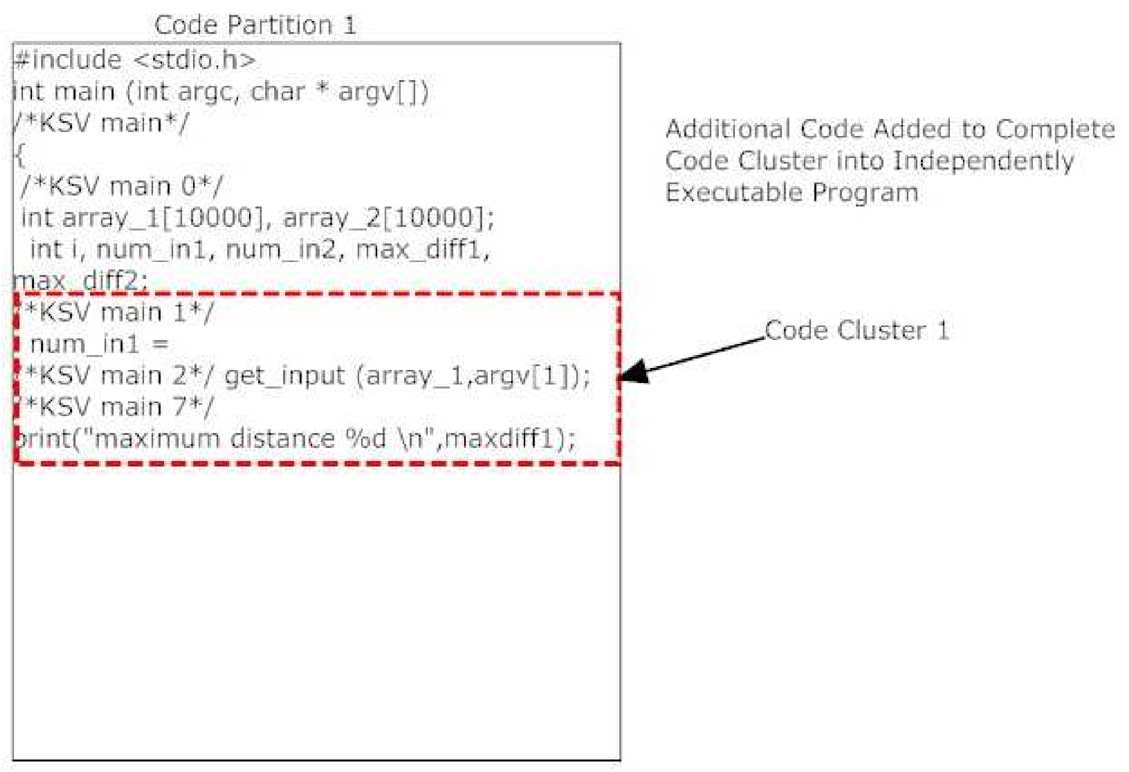
Figure 5.6. Adding Constructs

functions used in each of these tasks. Based on this information, declaration and functions for the entities present in the partition are added. This optimization helps in maintaining the code size and reducing redundant code duplication.

As seen in Figure 5.7., code cluster 1 is converted into a program Partition 1 by adding constructs. This is accomplished by adding the appropriate header files and functions that are being referenced by the tasks in the code cluster. Later, introducing the declaration information of the entities in order to errors on compilation.

5.4 Communication Methodology

The figures 5.8. and 5.9. illustrate the abstraction of the communication between two code clusters. The main function of code generation is to produce code for each distributed processor such that every task is correctly executed. Before a task is executed, the task with precedence to it, needs to complete execution and the data items referenced by the task are made available in the local memory. Additionally, after each new data item is computed, it must be sent to the appropriate processors whose tasks need this data for further computation. This requires the implementation of a communication layer.


Code Partition 1
Additional Code Added to Complete Code Cluster into Independently Executable Program
Code Cluster 1

```
#include <stdio.h>
int main (int argc, char * argv[])
/*KSV main*/
{
 /*KSV main 0*/
 int array_1[10000], array_2[10000];
  int i, num_in1, num_in2, max_diff1,
max_diff2;
/*KSV main 1*/
 num_in1 =
/*KSV main 2*/ get_input (array_1,argv[1]);
/*KSV main 7*/
print("maximum distance %d \n",maxdiff1);
```

Figure 5.7. Creation of Sub-Programs

5.4.1 Selection of PVM as the Communication Mechanism

Most common communication technologies include MPI, Java RMI and PVM. Java RMI is more suitable for Java applications, hence is not considered for this approach. We choose PVM [11] over MPI for its robustness and suitability for heterogeneous architectures.

5.4.2 PVM Communication Process

The PVM message passing process is explained in the steps given below:

Initialize send buffer: Before any message can be sent out from one processor to another, we need to allocate and initialize a send buffer.

Placing data into send buffer: This second stage of PVM message passing involves placing the data that is to be sent into the newly initialized send buffer.

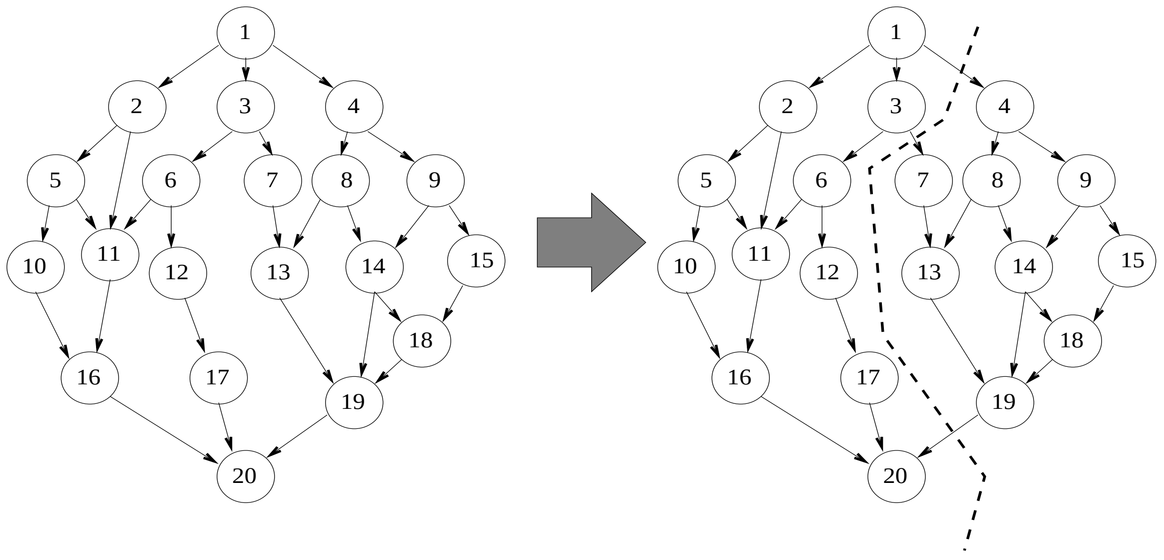


Figure 5.8. Illustration for Partitioning Using Clusters

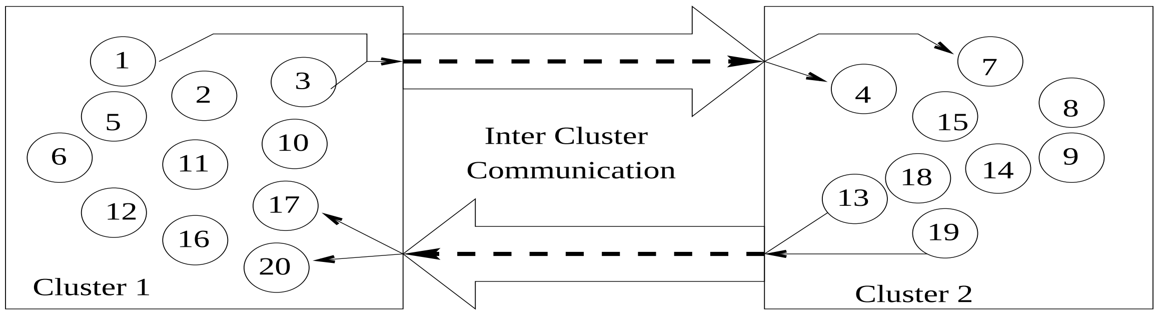


Figure 5.9. Abstraction of Communication Between Nodes from Different Clusters

Send out data in send buffer: This step involves sending the data in the send buffer to the destination processor.

Retrieve and place message in receive buffer: The destination processor receives the data from the sending processor and caches it in the receive buffer. There are two distinct type of receive commands, one is a blocking receive, wherein the destination processor will stall till the data arrives. The other type of receive command is non-blocking and resumes execution if the data does not arrive.

Unpack receive buffer: The final step includes unpacking the data from the receive buffer.

5.4.2.1 Asynchronous Messaging Primitives

Several asynchronous PVM message passing primitives used in this work are mentioned below:

- pvm_recv(int tid, int msgtag)

  Executing a receive command will receive a message with ID *msgtag* if it is in the communication buffer of the processor. Otherwise it blocks the processor idle till the message arrives.

- pvm_mcast(int *tids, int ntask, int msgtag)

  Executing a multicast command will send the data in the active message buffer *ntask* number of tasks within *tids.* This is useful for broadcasting the data.

- pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)

  A process named *task* will be initiated in the specified processor. The task ID of the spawned process will be sent to the parent process.

- pvm_initsend(PvmDataDefault)

  The routine *pvm_initsend* clears the send buffer and prepares it for packing a new message.

5.4.3 Implementing PVM Communication

PVM communication primitives are automatically interleaved in the sub-programs in a transparent fashion without changing the order of the original source. The technique used is fast, error free and does not involve intervention by the programmer unlike traditional methods that involved manual coding of the communication layer. However, it is necessary for the architecture to be PVM compliant.

In the transparent mode the mode, tasks are automatically executed on the most appropriate computer. Nevertheless, PVM gives the flexibility to allow the user to specify the processors to which the tasks need to be mapped in the architecture-dependent mode. In low-level mode, the user may specify a particular computer to execute a task. In all of these modes, PVM takes care of necessary data conversions from computer to computer as well as low-level communication issues.

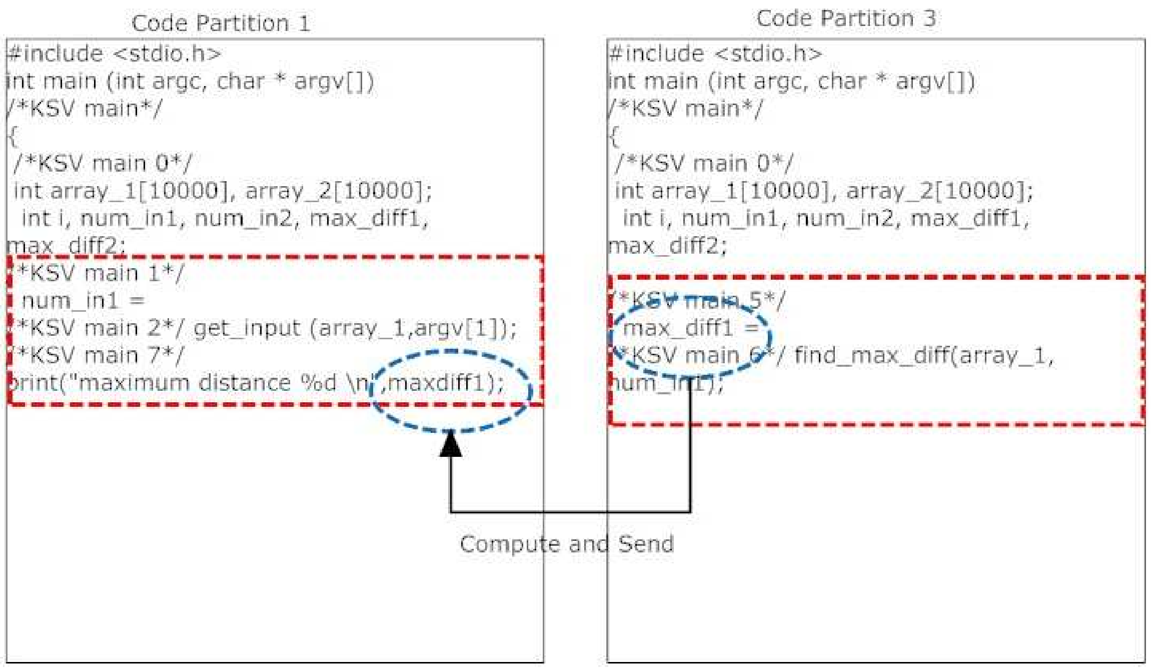
Once the centralized application is split into equivalent independent subprograms, the parent partition is selected. PVM constructs are then inserted into the parent partition to mobilize spawning of the other partitions on the specified appropriate processors. Then each partition is carefully profiled and message passing passing constructs are conservatively appended.

The message passing constructs are inserted into the subprograms based on the following:

- For every task that modifies an entity the metadata table is searched to check if there are other tasks that will be affected. If the tasks being influenced by the modification belong to different partitions the new value is broadcast to them.
- Similarly, before computing a task, the program checks whether the entities in that task have been modified previously. If the entities have changed, the new values are received from the respective partitions.

As the send command is non-blocking, the processor that sends the message can proceed with the execution. But the receive command is a blocking command and will stall till the message is not received. This aids any inherent parallelism in the program on the other hand ensures the correct scheduling and execution of the distributed program.

Figure 5.10. shows the need for communication. After the preprocessing and partition process is complete we obtain the above partitions 5.10. (Code Partition 1 and Code Partition 3). The tasks have been segregated in a manner that an entity that is computed in one partition needs to be printed in another partition. The entity needs to be sent to Code partition 1 as soon as it is computed for correct execution.



```
#include <stdio.h>
int main (int argc, char * argv[])
/*KSV main*/
{
 /*KSV main 0*/
 int array_1[10000], array_2[10000];
  int i, num_in1, num_in2, max_diff1,
max_diff2;
/*KSV main 1*/
 num_in1 =
/*KSV main 2*/ get_input (array_1,argv[1]);
/*KSV main 7*/
print("maximum distance %d \n",maxdiff1);
```

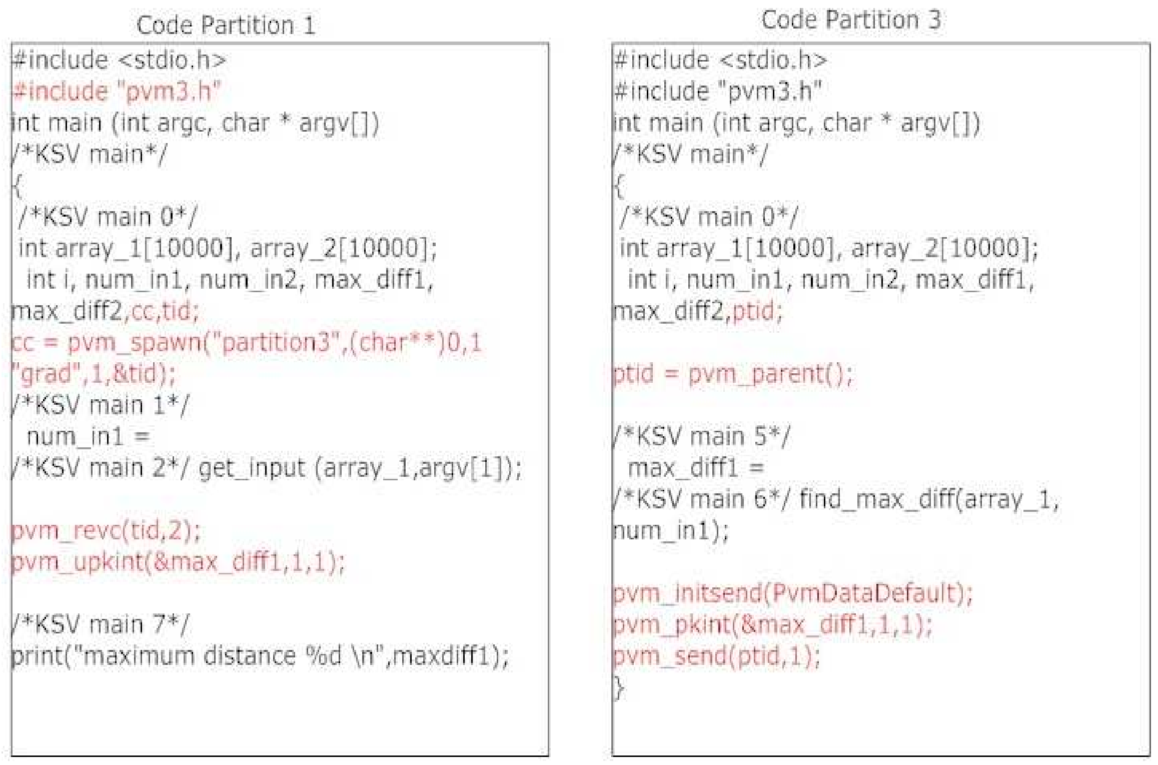
```
#include <stdio.h>
int main (int argc, char * argv[])
/*KSV main*/
{
 /*KSV main 0*/
 int array_1[10000], array_2[10000];
  int i, num_in1, num_in2, max_diff1,
max_diff2;

/*KSV main 5*/
 max_diff1 =
/*KSV main 6*/ find_max_diff(array_1,
num_in1);
```

Figure 5.10. Communication Between Sub-Programs

The communication between the two partitions is facilitated by addition of PVM primitives as shown in Figure 5.11.. In this implementation, Code Partition 1 is chosen as the parent process which spawns another process (Code Partition 3).

The PVM header files are included in all the partitions. The program finds that the entity modified in task 6 of partition 3 is required by task 7 of partition1. PVM instructions to initialize the send buffer, pack the new value and send it to the parent processor are



Code Partition 1

```
#include <stdio.h>
#include "pvm3.h"
int main (int argc, char * argv[])
/*KSV main*/
{
 /*KSV main 0*/
 int array_1[10000], array_2[10000];
  int i, num_in1, num_in2, max_diff1,
max_diff2,cc,tid;
cc = pvm_spawn("partition3",(char**)0,1
"grad",1,&tid);
/*KSV main 1*/
  num_in1 =
/*KSV main 2*/ get_input (array_1,argv[1]);

pvm_revc(tid,2);
pvm_upkint(&max_diff1,1,1);

/*KSV main 7*/
print("maximum distance %d \n",maxdiff1);
```

Code Partition 3

```
#include <stdio.h>
#include "pvm3.h"
int main (int argc, char * argv[])
/*KSV main*/
{
 /*KSV main 0*/
 int array_1[10000], array_2[10000];
  int i, num_in1, num_in2, max_diff1,
max_diff2,ptid;

ptid = pvm_parent();

/*KSV main 5*/
  max_diff1 =
/*KSV main 6*/ find_max_diff(array_1,
num_in1);

pvm_initsend(PvmDataDefault);
pvm_pkint(&max_diff1,1,1);
pvm_send(ptid,1);
}
```

Figure 5.11. Interleaving PVM Primitives

appended. Similarly, in code partition1, PVM instructions to receive and unpack the new value are automatically appended.

Figure 5.12. depicts the algorithm for inserting PVM instructions in the partitioned code.

5.4.4 Optimizations

This section deals with the optimizations handled for improving communication time. The techniques of using broadcast communication and elimination of redundant communication are explained in the following sections.

5.4.4.1 Broadcast Communication

A task could send the same message to many other tasks in different processors. The incorrect way to do this would be using repeatedly one to one sending, which could create

For each task:

- Check the metadata table for all the entities used in the task.
- For each entity in the task:
  - Check if the entity was modified in an earlier task.
  - If the entity is modified check if the task in which the entity is modified is within the same partition.
    - If within the same partition then no need to have a receive statement.
    - If not then receive from the other task.
  - Similarly, if an entity is being modified in the task, broadcast the new value if the entity is being used in a different partition.

Repeat for all tasks

Figure 5.12. Communication Algorithm

communication contention in the processor network. Thus a broadcasting scheme should be used to take advantage of the network topology in order to alleviate network message traffic.

In this implementation, after an entity is modified, the control flow information in the metadata table is searched to check if all other partitions require the modified entry. If they all require the modified value instead of having multiple send receive commands we replace it with a single broadcast command.

5.4.4.2 Optimizing Communication Overhead

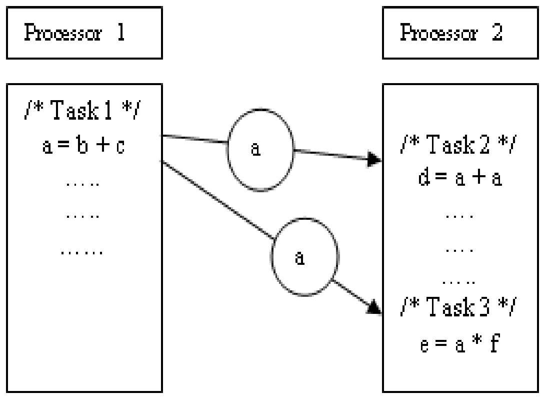


Figure 5.13. Local Memory Optimization

Local Memory: If two tasks are assigned in the same processor, there is no need for communication between them. This is because even if one task modifies an entity, the new value still exists in the local memory. For example in the Figure 5.13. two tasks (task 2 and task 3), assigned to the same processor (processor 2), are each receiving data from another task (task 1) in a different processor (processor 1). Processor 1 issues two sends with the same message to processor 2, and processor 2 issues two receives for the task 2 and task 3.

An optimized approach should eliminate such redundant communication. Processor 2 does not require receiving the same data for task 3 that it already has received for task2, as the data can be stored in the local memory. For task 3, data can be fetched from the local memory itself, reducing he communication expense. Thus a redundant receiving can be detected if the data item has already been in the local memory.

Dead Messages: Another optimization is to eliminate communication if an object is not going to be used any further. This is because even if the data item is modified it is not required by any other task.

5.4.4.3 Use Aggregate Communication for Consecutive Tasks

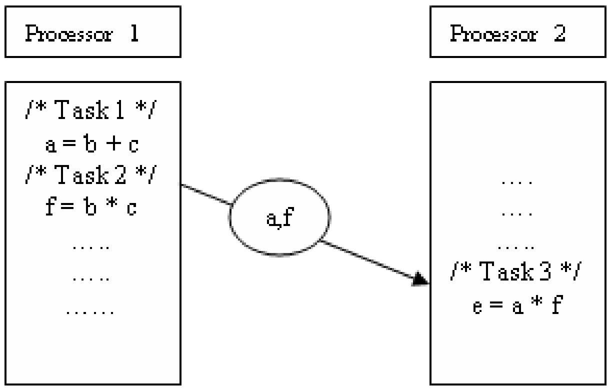


Figure 5.14. Aggregate Communication Optimization

Using aggregate communication for consecutive tasks, when required, improves performance. For example, in the Figure 5.14., Task 3 in processor 2, requires two entities that are being changed in processor 1. One approach could be two send the entities separately, each time the entity was modified using two send and receive messages. This approach uses an optimized method of packing both the entities in a single send command. The basic aim is to send less number of long messages.

5.4.5 Code Size Increase

This sub section presents the resultant increase in code size as an effect of code generation. A comparison between the commonly used broad cast approach and the optimized version of sending information is performed. The final simulation results of both the approaches are presented in table 5.1..

Table 5.1. Overhead Increase in Terms of Lines of Code

	% increase in lines of code	
# Clusters	naive approach	optimized approach
3	35.78	18.97
3	38.63	19.97
5	42.81	20.88
6	45.24	21.31
7	47.22	22.56

5.5 Summary

This chapter covered the implementation of the automatic code generator tool. Important program information was collected and stored in a metadata table to aid interleaving the communication layer. Additionally, the application is segregated into independent subprograms by conservatively adding constructs to the code clusters. The rationale behind using PVM as the communication methodology was discussed. The chapter also covered the implementation and design of PVM as the communication methodology in our work. A detailed description of the optimization techniques employed to increase performance of the distributed code was presented in the last section of the chapter. The techniques include usage of aggregate communication, broadcasting information instead of using one-to-one communication and elimination of redundant communication.

CHAPTER 6
A FRAMEWORK FOR ARCHITECTURAL EXPLORATION

In this chapter a novel approach and framework is described for architectural design space exploration based on Transaction Level Modeling(TLM) for SoC's A system level design/synthesis framework based on TLM models is proposed.

The main components of ESL design include application specific processors, embedded software and high speed Intellectual Property (IP) cores or third party processing elements. Fast and efficient system-Level design and synthesis of distributed heterogeneous architectures is critical for new product development especially in the fast paced and ever changing consumer electronics segment of the market. Maintaining a strict time to market for a new product is critical for the success of the product launch and the survival of the product line. A brief overview of the classic design flow is now presented.

6.1 Electronic System Level Design

The key components of the system level design process include an accurate model of the system and an implementation tool for verifying the model. System level models are usually described using high level programming languages like C or C++. The models generated are then mapped onto existing processing elements from third party vendors or in special cases some parts of the model are implemented in hardware as ASIC's. One of the major issues in system level design is the problem of identifying the appropriate processing element for mapping a particular task. Heterogeneous systems have multiple processing elements and the task of the system architect is further complicated by the numerous choices of architectures. System level models described using high level programming languages lack implementation details. Hence the designer's tasks includes making assumptions about implementation details that were not available in the initial design phase The models at the system level also do not include cycle time information.

System level design process has been automated using CAD tools. The design process starts with the requirements specification document. System level design models are implemented using a high level programming language for design verification.

6.2 Classic Design Flow

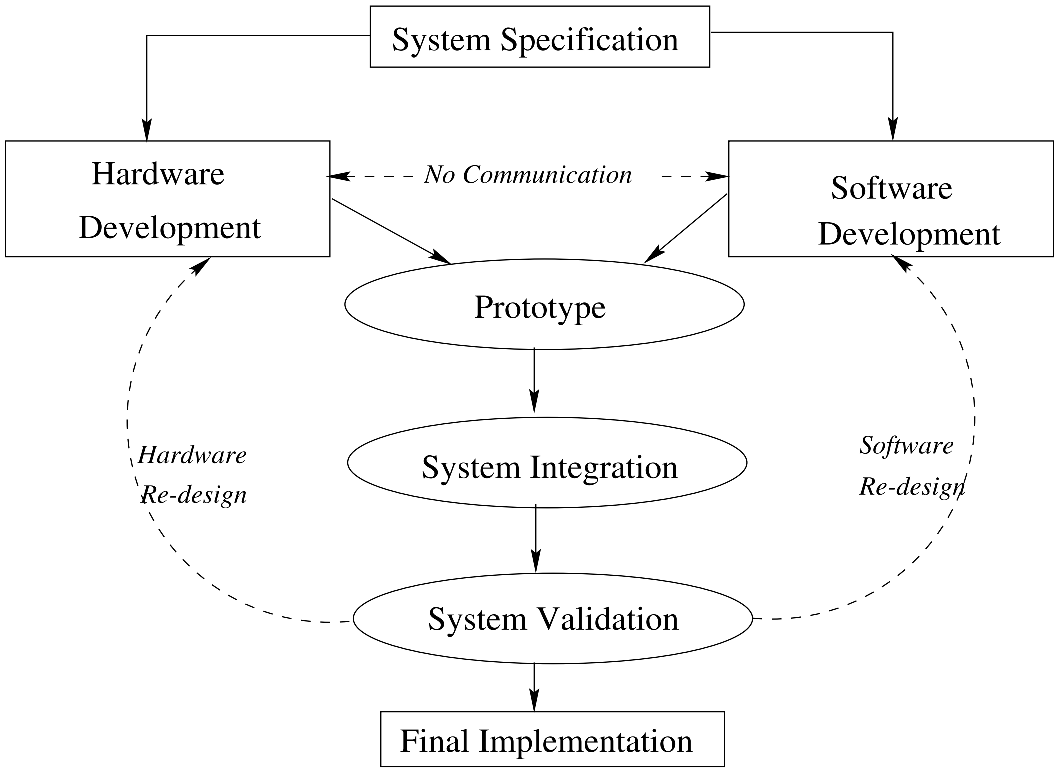


Figure 6.1. Classic Design Flow for Electronic System Level Design

The steps involved in the classical design flow for system level design [1] is illustrated in figure 6.1. The process starts with the collection of requirements from the customer. The requirements are then translated into an intermediary form known as the functional specification document or the system specification. Hardware and software development processes are initiated in parallel with the system specification document as the input.

During the entire process of complete hardware and software development, there is no communication between the hardware development and software development teams. Once the development on both tracks have been completed, the entire system is integrated together into one module. The system is then validated for functional correctness and then for other performance based issues. All issues arising after the stage of validation introduce another cycle of independent hardware and software development. This process independent development continues until all issues have been resolved and the system model has been validated. The next step in the design flow is implementing the prototype and testing the implementation.

The primary reason that this design flow worked initially was due the limited number of available choices for system architects. The existing processing elements available for use in the system design were usually of two categories, general purpose processors or micro controllers. General purpose processors could not used in embedded system designs since most of the systems do not include operating system support. The cost of the general purpose processors could not justify the cost of the entire product after development. In the case of micro controllers the degree of programmability was very less and each micro controller had a specific function. Eventually an ASIC needed to be custom designed in order to interface with the micro controllers and achieve the functionality of the system design. The functionality of the ASIC was clear from the start of the design. Hence independent development of the hardware and software was still viable and the number of iterations required to resolve the issues in the functional verification of the system design was limited to an acceptable number.

Diminishing feature sizes introduced a new complication in the design flow. The choices available to system designers were large and the system level design decisions played a critical role in the development of an efficient and optimal system. The advent of SoC's (System on Chips) introduced new design choices and challenges in the design process. The traditional design flow could not support the design of the new SoC's. The independent development of hardware and software is clearly an impediment for fast and efficient generation of a prototype. The problem can easily be fixed with the integrated development of hardware and software in parallel. The increased communication between the development processes assures a very low volume of issues created in the design. The above observations were the motivation for the hardware/software codesign process. Distributed heterogeneous systems design adopted the hardware/software codesign methodology and all the steps including partitioning, scheduling, simulation and verification were performed at the hardware and software domains in parallel with effective communication and feed back.

6.3 SoC Design Flow

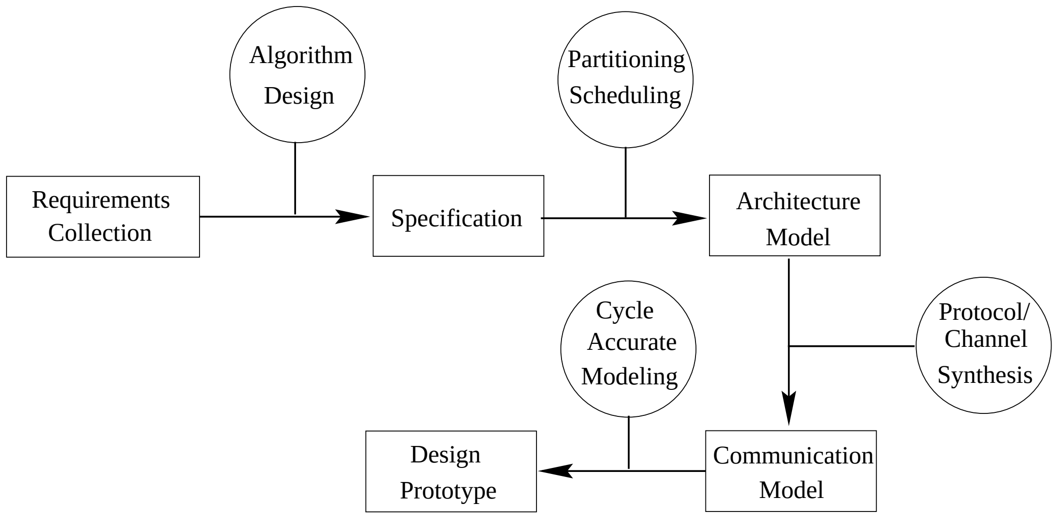


Figure 6.2. Design Flow for Modeling SOC's

The steps involved in a SoC design flow are illustrated in figure 6.2.. The process starts with the collection requirements from the customer. An appropriate algorithm is chosen for modeling the requirements specification and the functional model of the system is developed. Partitioning and scheduling are performed on the functional model in order to obtain the architectural description model of the system. At this stage the functional units in the system design are finalized. The hardware and software processes are identified. The next step in the design flow is identifying the channels of communication and finalizing the protocol configurations that would be used in the communication synthesis of the model. The communication model is finalized after all communication based requirements have been identified and fulfilled. The final step in the design flow of the SoC is using cycle accurate modeling to implement the prototype. This prototype is simulated and tested for functional requirement completion. It is evident from the steps involved in SoC design that traditional design flow would not be efficient enough to generate a optimal prototype in a short time frame.

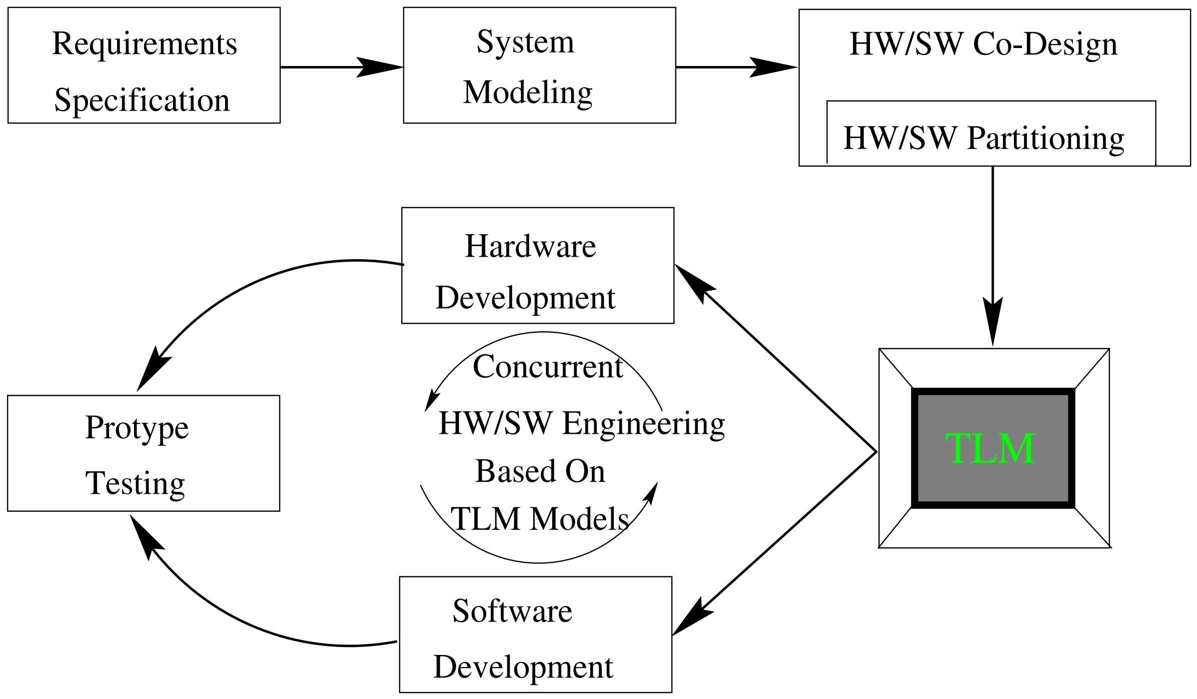


Figure 6.3. Novel Approach for Modeling SOC's

In order to over come the short comings of the traditional design flow in the development of SoC's a novel design flow was proposed. Figure 6.3. illustrates the steps involved in the novel design flow . The design flow now includes the use of TLM models. The author in [25] provides an elaborate discussion on transaction level modeling and how it interacts with system C modules. The novel design flow introduces numerous challenges for the system design engineer. The design space for architectural exploration is now huge. We can observe this from the following factors.

- The choice of reusable Intellectual property(IP) cores available is huge
- The parameters that affect design performance are numerous
- The needs of applications are specific, hence fine tuning can be performed only on a case by case basis.

Observing from the above mentioned reasons we can see that it is difficult to come up with one standard for the different levels of abstraction. Hence most of the research in this area is currently carried out on case study basis, particular for one specific scenario.

6.4 TLM/SystemC Based Exploration for SoC Architectures

TLM can be classified according to use cases in a broad sense.

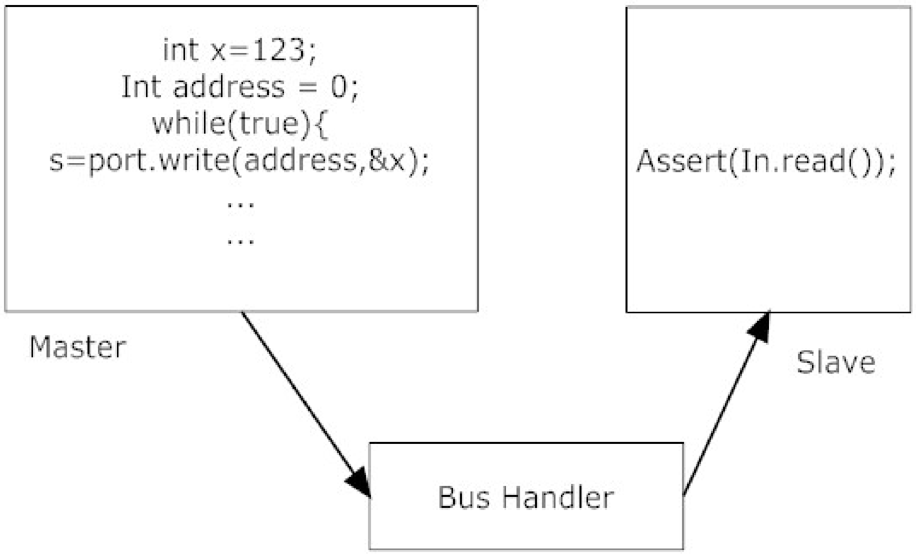
- *Functional view:* Requirements/Specification of the application
- *Architect's view:* Architectural exploration
- *Programmer's view:* Embedded software design
- *Verification view (could also be combined as part of architect and programmer view):* System validation and HW-SW co-verification

A Transaction can be defined as an event that involves a data exchange between the entities of the systems. The entities in the system include both communication objects and processing elements [20]. Each transaction is considered to be atomic, that is either complete or has not occurred at all. Transaction level models can be used at different levels of abstraction. Broadly used levels of abstraction include the following:

- Untimed models - highest level of abstraction - mostly functional
- Approximately timed - pin accurate models

- Accurately time - cycle accurate models - mostly RTL generated code for design verification.

An illustration of communication architecture modeling using TLM is shown in figure 6.4.. The entities in the system include processing elements and communication elements. TLM



```
int x=123;
Int address = 0;
while(true){
s=port.write(address,&x);
...
...
```

```
Assert(In.read());
```

Figure 6.4. Sample Communication Code Using TLM

is used describe both the categories. Both processing objects and communication objects can be modeled at any level of abstraction using TLM. In this proposed approach we discuss the scenario of modeling the communication architecture for bus-based systems using TLM in the exploration of SoC's. In the particular case of communication architecture modeling the abstraction levels of TLM can be further defined as follows.

- *Cycle accurate:* - A high level abstraction of Register Transfer Level (RTL) detail, includes cycle accurate and pin accurate information

- *Pin Accurate / Bus Cycle Accurate (PC-BCA):* - is an abstraction over the CA approach where in behavior inside the components are not modeled for every cycle, only the bus is still cycle accurate.

- *Transaction based Bus Cycle accurate:* - TLM based models are used for communication exploration.

We can observe from the above description of the different levels of abstraction for modeling the communication architecture using TLM that there are specific advantagesin using a particular level of abstraction. The cycle accurate models provide detailed and accurate information. They can modeled using C/C++ or any high level programming language. The obvious down side in using cycle accurate models is that simulation takes a lot of time for completion. The pin accurate models are 1000x faster than RTL and require only one tenth of the modeling effort of RTL. Transaction based models provide the highest level of abstraction and most of the smaller implementation details are transparent. Most low level bus signals are replaced by high level functions. Simpler Interface reduces modeling effort at transaction level. The speed of simulation is about 1000x RTL and they involve only about a twentieth of RTL modeling effort.

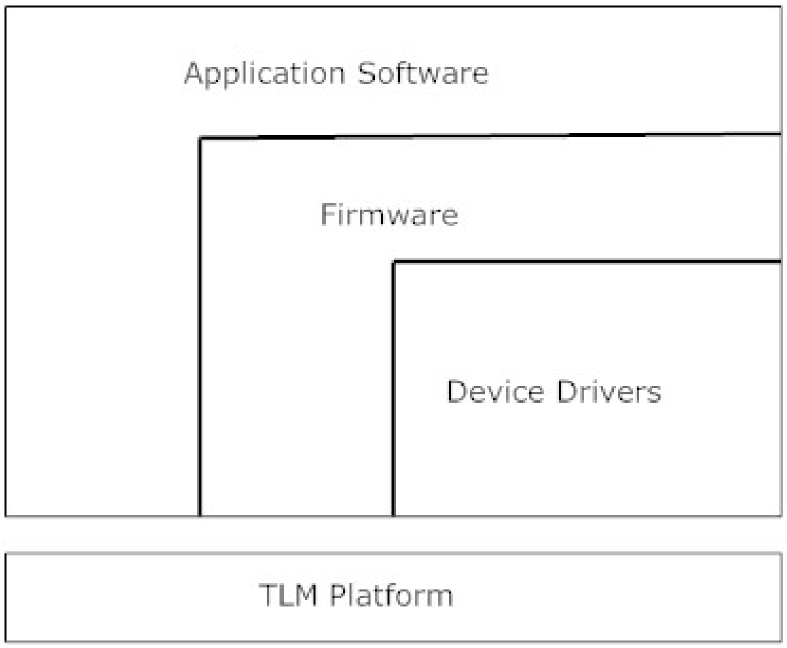


Figure 6.5. Software Areas that can be Modeled using TLM

The versatility of TLM models extend to all areas in the software description. This is better illustrated by the figure 6.5.. We can observe from the figure that since TLM operates at different levels of abstraction. The models can be generated at any level in the software architecture hierarchy. Depending on the amount of detail required for the model different level of abstraction can be chosen by user. Design exploration and optimization is main goal of the TLM methodology. The functional specification can be modeled using TLM and the design is verified using generated RTL code. The optimization process can now be performed in iterations with feed back after every iteration, between the TLM plane and the RTL plane. Figure 6.6. illustrates the process of continuos feed back and optimization of the system design using TLM. Hence TLM is ideally suited for architectural exploration and communication architecture modeling. The framework proposed would take the requirements specification from the customer in a generic format like UML(unified modeling language). Code generation tools like rational rose can be employed to generate C/C++ functional specification from the UML requirements document. The functional level code is subjected to hardware software partitioning and then scheduling. TLM is applied with SystemC as front end to model behavior for simulation. The system designer can then fully utilize the power of abstraction of TLM to verify and synthesize the system design with the least amount of effort involved for design and testing.

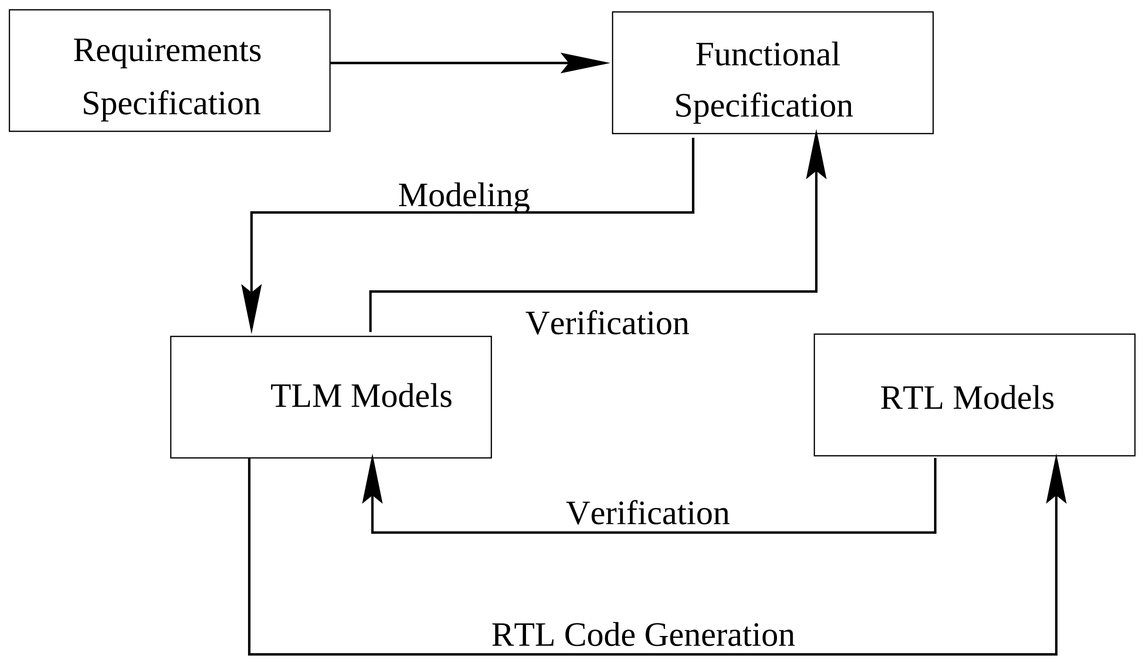


Figure 6.6. Optimization using TLM Models

CHAPTER 7
CONCLUSION

In this work we presented a complete automated framework to acheive code partitioning and code generation for distributed heterogeneous systems. The frame work consists of profiling tools bundled with the partitioning algorithm and the code generation module. The tool is end to end automated and requires no human intervention. The frame work can be used for rapid prototyping in mapping applications onto distributed heterogeneous systems. The approach can be used for providing insight into the behavior of applications on distributed systems and in turn can promote efficient mapping of parts of the application onto the different processing elements of the system in order to decrease execution time, optimize communication latency and even acheive low power dissipation.

The partitioning algorithm developed was based on clustering. Clustering based methods were found to produce good quality results with a extremely low execution time. The low run times for the clustering based approaches is preferrable over all other partitioning approaches in the case of distributed applications requiring fast response time. The partitioning algorithm simulates the application for a range of cluster numbers and generates the corresponding mapping. The user can conveniently choose the optimal mapping configuration based on the amount of savings in communication latency or the metric that the cost function is tuned for.

The frame work includes a completely automated tool for code generation for mapping applications onto distributed heterogeneous systems. The code generation approach performs communication synthesis and completes the different clusters of code into individually executable independent programs. The communication between the individual segments is acheived by synthesizing PVM system calls. Inter cluster data dependencies are satisfied by sending PVM calls packed inter cluster dependent data. The code generation approach works especially for ANSI C based distributed applications.

The advantages of algorithm and the code generation approach are that they are both scalable. Increasing the number of nodes in the application has little effect on the execution time in the case of clustering. Both Tabu search and simulated annealing based approaches suffer adversely with the effect of increase in the number of nodes. The frame work is also very easily extendible. Support for new processing elements with different processor architecture in the distributed system can easily be acheived with the generation of a cross compiler for the new architecture. The system requires no other change to accomodate the new architecture. The frame work has been designed with this key idea in mind. The profiling tools of the application have been implemented using powerful shell programming tools. The arrangement of modules in the design flow of the profiling process is such that support for new set of profiling tools can be implemented easily by including the new profiling module in the flow. The results of the new profiling module are also cumulatively collected alongwith the results of other modules without introducing any major changes in the profiling process flow.

Further scope for this work is in the direction of TLM (Transaction Level Modeling) for ESL - Electronic System-Level Design. The entire frame can then be extended from heterogeneous distributed systems to the domain of embedded systems. Rapid prototyping and advancements in the speed product life cycle development are the key factors that is affecting most of the embedded system designers. TLM tools provide fast and efficient mechanisms to visualize, design implement and test prototypes for new market ideas with a very short span of time and with very little amount of effort. The level of abstraction in the TLM models controls the amount of implementation specific information that needs to be an input for the design of the distributed embedded system. The embedded system designers can utilize the short design turn around time for exploring the numerous choices available in terms of choosing processor architectures, communcation topologies and arbitration protocols for synthesized communication architectures.

A TLM based design strategy and communication synthesis approach has been proposed. The approach is yet to be tested for the full design flow. The first step in the new approach would be translation of the application specification to a system level functional description using a high level programming language. The second and the third step step in the proposed novel design flow are the same paritioning and code generation steps. The final step in the flow would be the communication synthesis of the architecture. Different levels of abstraction can used to represent the details of the communication architecture. The entire frame work would represent a modular design flow for mapping applications to a distributed heterogeneous system even without operating system support.

REFERENCES

[1] A. Rettberg, M.C. Zanella and F.J. Rammig. *From Specification to Embedded System Applications*. Springer Publications, 2005.

[2] A. Sinha and A.P. Chandrakasan. Joule Track - A web based tool for Software Energy Profiling. In *IEEE/ACM Design Automation Conference* , pages 220–225, 2001.

[3] A. Spiegel. Pangaea: An automatic distribution front-end for java. In *4th IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 93–99, 1999.

[4] A.C. Nacul and T. Givargis. Code Partitioning for synthesis of Embedded applications with phantom. In *Proceedings of the IEEE/ACM International Conference on computer aided design*, pages 190–196, 2004.

[5] C. Chakrabarti and D. Gaitonde. Instruction Level Power Model of Microcontrollers. In *International Symposium on Circuits and Systems*, pages 76–79, 1999.

[6] D. Chandra, Ch. Fensch, W.K. Hong, L. Wang, E. Yardimci and M. Franz. Code generation at the proxy: An infrastructure-based approach to ubiquitous mobile code. In *Proceedings of the Fifth ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS 2002), Malaga, Spain*, 2002.

[7] D. Wu, B.M. Al-Hashimi and P. Eles. Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems. In *IEEE Proceedings-Computers and Digital Techniques*, volume 150, pages 262–273, 2003.

[8] F. Vahid and D. Gajski. Closeness metric for system-level functional paritihtioning. In *Proceedings of IEEE International Conference on European Design Automation 1995*, pages 328–333, 1995.

[9] F. Vahid and T.Givargis. *Embedded Systems Design- A Unified Hardware/Software Introduction*. John Wiley and Sons, 2002.

[10] G. Stitt and F. Vahid. Hardware/Software Partitioning of Software Binaries. In *IEEE/ACM International Conference on Computer Aided Design*, pages 164–170, 2002.

[11] G.A. Geist, J.A. Kohla and P.M. Papadopoulos. Pvm and mpi: A comparison of features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

[12] G.C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 187–200, 1999.

[13] H.A. Muller, S.R. Tilley and K. Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 217–226, 1993.

[14] H.Z. Zebrowitz. *GEDAETM - A Graphical Programming and Autocode Generation Tool for Signal Processor Applications*. Lockheed Martin Corporation.

[15] J. Henkel. A Low Power Hardware/Software Partitioning for Core-based Embedded. In *Proceedings of the 36th ACM/IEEE conference on Design Automation Conference*, pages 122–127, 1999.

[16] J. Henkel and Y. Li. Energy Conscious Hardware Software Partitioning of Embedded Systems: A case study on an MPEG-2 Encoder. In *Proceedings of the Sixth International workshop on Hardware/software codesign*, pages 23–27, 1998.

[17] J. Kim and I. Lee . Modular code generation from hybrid automata based on data dependency. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003)*, pages 160–168, 2003.

[18] J.B. Peterson, R.B. O'Connor and P. M. Athanas. Scheduling and partitioning ANSI-C programs onto multi-FPGA CCM architectures. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 178–187, 1996.

[19] K.S. Vallerio, and N.K. Jha. Task Graph Extraction for embedded system synthesis. In *Proceedings of IEEE International Conference on VLSI Design*, pages 480–486, 2003.

[20] L. Cai and D. Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, 2003.

[21] L. Guthier, S. Yoo and A. Jerraya. Automatic generation and targeting of application specific operating systems and embedded systems software. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 679–685, 2001.

[22] L.R. Welch, B. Ravindran, J. Henriques and D.K. Hammer. Metrics and Techniques for Automatic Partitioning and Assignment of Object-based Concurrent Programs. In *Proceedings of The Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 440–447, 1995.

[23] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R.K. Brayton and A. Sangiovanni-Vincentelli. Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES), Estes Park, Colorado, USA*, pages 151–156, 2002.

[24] M. Lopez-Vallejo and J. C. Lopez. On the hardware-software partitioning problem: System modeling and partitioning techniques. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):269–297, 2003.

[25] M. Moy. *Techniques and tools for the verification of Systems-on-a-chip at transaction level.* PhD thesis, INPG, Genoble France, 2005.

[26] N. Liogkas, B. MacIntyre, E. Mynatt, Y. Smaragdakis, E. Tilevich and S. Voida. Automatic Partitioning for Prototyping Ubiquitous Computing Applications. *IEEE Pervasive Computing*, 3(3):40–47, 2004.

[27] N.S. Singh. An automatic code generation tool for partitioned software in distributed computing. Master's thesis, University of South Florida, 2005.

[28] P. Eles, Z. Peng and K. Krzysztof. System level hardware/software partitioning based on simulated annealing and tabu search. *Kluwer Journal on Design Automation for Embedded Systems*, 2(1):5–32, 1997.

[29] P. Prabhakaran and P. Banerjee. Parallel algorithms for force directed scheduling of flattened and hierarchical signal flow graphs. *IEEE Transactions on Computers*, 48(7):762–768, 1999.

[30] P.C. Mahalanobis. On the generalised distances in statistics. In *Proceedings of the National Institue of Science*, 1936.

[31] R. Canal, J-M. Parcerisa and A. Gonzalez. Dynamic Code Partitioning for Clustered Architectures. *International Journal of Parallel Programming*, 29:59–71, 2001.

[32] S. Nikolaidis. Instruction - level energy characterization of an arm processor. In *Marlow workshop*, 2003.

[33] Scientific Toolworks, Inc. *Understand C.*

[34] T. Yang and A. Gerasoulis. Pyrros: static task scheduling and code generation for message passing multiprocessors. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 428–437, 1992.

[35] V. Kindratenko. Code Partitioning for Reconfigurable High-Performance Computing: A Case Study. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 143–149, 2006.

[36] W. Li and C. McDonald. Full support for textual editing in a syntax-directed editor. Technical Report 94/8, Department of Computer Science, The University of Western Australia, 1994.

[37] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 507–512, 2000.

APPENDICES

Appendix A Clustering Background

Hierarchical clustering mechanisms are used for identifying clusters in data items that need to be compared on multiple dimensions. This work uses a variation of the K-means clustering. In this chapter, a breif overview of the different clustering methods is provided. A comparison of the methods, is also provided. The motivation for choosing K-means clustering over other clustering approaches is elaborated. The rest of the chapter provides an overview on the different ways to compute the distance between the nodes of the set and a comparison of the methods. The trade offs involved in using individual methods are explained further.

Clustering algorithms can be categorised as hierarchical or non-hierarchical approaches. Hierarchical clustering algorithms can be further classified as divisive or agglomerative. In divisive approaches usually the algorithm begins with the entire set as one cluster and then multiple clusters are formed by seperating out elements that do not match the membership property of the set closely. In the case of agglomerative based approaches all the elements are considered as individual clusters and then the elements with close distances are combined together to form clusters. This work uses K-means clustering, which is a non-hierarchical approach. The major difference between a hierachical approach and a non-hierarchical approach is that in the case the K-means clustering an initial number of clusters is provided as input to the algorithm. The advantage of the method is that the elements can be divided into any number of clusters as needed by the user. The disadvantage is that, the number of clusters chosen by the user is not necessarily the optimal number. This chapter provides an overview of the k-means clustering and the metrics used to compute the distance factor between the nodes in the cluster.

K-means Clustering: The K-means algorithm assigns each point to the cluster whose center (also called centroid or medoid) is nearest. The center is the average of all the points in the cluster, that is, its coordinates are the arithmetic mean for each dimension separately over all the points in the cluster... Example: If the data set has three dimensions and the cluster has two points: $X = (x_1, x_2, x_3)$ and $Y = (y_1, y_2, y_3)$. Then the centroid Z becomes $Z = (z_1, z_2, z_3)$, where $z_1 = (x_1 + y_1)/2$ and $z_2 = (x_2 + y_2)/2$ and $z_3 = (x_3 + y_3)/2$. The algorithm steps as formulated by J. MacQueen(1967) are as follows:

- Choose initial clusters, k.
- Randomly generate k new cluster centers
- map points to closest cluster center
- compute cost to verify if the new cluster centers form a better mapping
- continue the process until stopping criteria is met

The main advantages of this algorithm are its simplicity and speed which allows it to run on large datasets. Its disadvantage is that it does not yield the same result with each run, since the resulting clusters depend on the initial random assignments. It minimizes intra-cluster variance, but does not ensure that the result has a global minimum of variance.

The most common form of the algorithm uses an iterative refinement heuristic known as Lloyd's algorithm. Lloyd's algorithm starts by partitioning the input points into k initial sets, either at random or using some heuristic data. It then calculates the mean point, or centroid, of each set. It constructs a new partition by associating each point with the closest centroid. Then the centroids are recalculated for the new clusters, and algorithm repeated by alternate application of these two steps until convergence, which is obtained when the points no longer switch clusters (or alternatively centroids are no longer changed). Lloyd's algorithm and k-means are often used synonymously, but in reality Lloyd's algorithm is a heuristic for solving the k-means problem, but with certain combinations of starting points and centroids, Lloyd's algorithm can in fact converge to the wrong answer. There are two factors that are important in order to obtain an optimal solution with the K-means algorithm. The first factor the distance metric used in deciding the membership of nodes to a given cluster. The second factor is the cost function used to steer the algorithm.

Distance Metric: In this section a breif overview of function use to compute the distance metric is described. Clustering approaches work by grouping elements with similar attributes into clusters. The similarity(or dissimilarity) between nodes is quantified by the distance metric. The formulation of the distance metric depends on the data elements itself. Each element in the entire set can be considered equivalent to a point. The dimension of the space containing the point corresponds to the number of different attributes of the element that is under consideration. One of the earliest techniques for computing the distance metric was proposed by Mahalanobis[30].

$$D_m = \sqrt{(x-\mu)^T.S^-1.(x-\mu)} \tag{A.1}$$

is the distance of the multivariate vector x $= (x_1, x_2, x_3, ...x_n)^T$ from the group of values with mean $\mu = (\mu_1, \mu_2, \mu_3, ...\mu_n)^T$. For any two random vectors, x and y the dissimilarity measure between them is defined as follows:

$$d(\vec{x}-\vec{y}) = \sqrt{(\vec{x}-\vec{y})^T.S^(-1).(\vec{x}-\vec{y})} \tag{A.2}$$

The distance metric thus computed is very effective in grouping, data elements with a large number of attributes and a wide range of values. Hence this work adapts the mahalanobis distance metric. Instead of applying the metric directly on the data elements, itself. The metric is modified to account for the distance between clusters.

Cost Function and Termination Criteria: The cost function steers the clustering algorithm. Improper formulation of the cost function could lead to either a sub optimal solution or in the worst case an infinite loop where the algorithm is not able to stop at all. The clustering algorithm is based on a greedy approach, hence one important factor to take into account when evaluating the solution is, the solution could be a local optimum. The key check points that have been added in order to ensure that the algorithm does not encounter any of the above mentioned drawbacks are described now. The first change applied to the cost function was to compute the change in the cost of the solution based on the total cost of all clusters in addition to variations in cost of each cluster. Traditional approaches would include a node in a cluster based totally on the cost of the individual cluster under consideration. The second important change applied to the cost funtion in order to avoid an infinite search for the solution was the addition of a maximum number of iterations of change for the clusters. There is a probability of an inferior quality solution, because of the restriction imposed on the number of iteration. A third change is applied in order to eliminate such a probability. The entire clustering algorithm is repeatedly executed a hundred times with each run executed with a restriction of a random number of iterations. The final solution is the best of the hundred runs.

ABOUT THE AUTHOR

Viswanath Sairaman completed high school in Chennai , India in 1994. Received his undergraduate degree in Bachelor of Science, majoring in Math from the university of Madras in 1997. Completed his Masters in Computer Applications from Regional Engineering College, Warangal - currently National Institute of technology , Warangal in the year 2000. He was an intern for Honeywell India in the year 2000 for about six months. He then joined HCL technologies in the role of a software engineer in July, 2000. He has about one year experience in software development and testing, of which four months were in Fairfield, Connecticut and the role involved developing software for hardware devices for IPC corporation. After successfully gaining invaluable experience he was admitted into the Ph.D program at the University of South Flordia, Tampa where his area of research was architectural design space exploration, code partitioning and mapping partitioned applications onto distributed systems