Abstract

EATMON, DEDRA Evaluating Placement Algorithms with the DREAM Framework for Reconfigurable Hardware Devices *(Under the direction of Dr. Clay. S. Gloster, Jr.)*

The field programmable gate array (FPGA) has become one of the most utilized configurable devices in the area of reconfigurable computing. FPGAs have a large amount of flexibility and provide a high degree of parallel computing capability. Since their introduction in the 1980's, these configurable logic devices have experienced a dramatic increase in programming capabilities and performance. Both factors have been significant in the changing roles of configurable devices in custom-computing machines. However, the improvements in capability and performance have not eliminated the issues related to efficient placement of applications on these devices.

This thesis presents a tool that evaluates placement algorithms for configurable logic devices. Written in Java, the tool is a framework in which various placement algorithms can be executed and the performance and quality of each placement evaluated using a cost function. Based on devices that support relocatable hardware modules (RHMs), the tool places modules with user-specified placement algorithms and provides feedback that can be used for a comparative analysis. The framework manages module mappings to the logic device that are both independent of each other and do not require pin-to-pin routing connections. Such a tool is valuable for the identification of effective placement algorithms for real-time placement of RHMs

in run-time reconfigurable systems.

The Dynamic REsource Allocation and Management (DREAM) framework, has been designed and developed to evaluate FPGA placement algorithms/heuristics. A portion of the evaluation is based on a simplistic cost function that calculates the amount of contiguous unused space remaining on the device in two dimensions. In our experiments, we use an FPGA logic core generator to produce several rectangular RHMs. In addition to the rectangular RHMs produced by the logic core generation tool, our framework can handle arbitrary circuit profiles. Several scenarios consisting of approximately 500 insertions/deletions of both rectangular and non-rectangular RHMs are used as test data sets for placement. Three placement algorithms are presented to demonstrate the flexibility of the framework. The first algorithm tested in the DREAM framework is a random placement algorithm. The second algorithm is an adaptation of a traditional best-fit algorithm that we call exhaustive search. The third algorithm is a modified version of first-fit. Future work will involve the development of additional placement algorithms and the incorporation of placement issues that relate to requests for central reconfigurable computing resources originating from a remote site.

The DREAM framework answers the call for a tool that is sorely needed to identify placement algorithms that can be effectively used for real-time placement. In addition to providing results that can be used to benchmark the performance of placement algorithms in real-time on a configurable system, this tool also allows the end-user methods to store and load placements for future optimization. By taking full advantage of the partial and full dynamic reconfiguration capabilities of logic devices currently used in run-time reconfigurable systems, the goal of DREAM is to provide a tool with which the quality of placement algorithms can be quantified and compared.

Evaluating Placement Algorithms with the DREAM Framework for Reconfigurable Hardware Devices

by

Dedra Eatmon

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Department of Electrical and Computer Engineering

Raleigh, NC

July 2000

Approved By:

Chair of Advisory Committee

In an effort to honor the sister I never knew, Janice Faye Eatmon . . .

Biography

DEDRA EATMON was is the fourth of five children born to Edward Louis Eatmon and Beulah Barnes Powell. She is originally from Greensboro, NC. Dedra became actively involved in math and science programs in during junior high school at the urging of those interested in developing her potential.

At Ben L. Smith Senior High School, Dedra attended honors courses and participated in Student Government, Spanish Club and the Track Team. She was chosen to attend the North Carolina Governor's School East in Laurinburg, NC during the summer of 1989. That fall, she entered the North Carolina School of Science and Mathematics as a junior. At NCSSM, Dedra further developed interests in math and technical sciences. The courses there challenged and prepared her for the independent learning environment of a collegiate atmosphere.

After graduating from NCSSM in 1991, Dedra went on to attend North Carolina Agricultural and Technical State State University in Electrical Engineering. She entered A&T as a General Motors scholar. She was also a recipient of the Ronald E. McNair Scholarship. Dedra was a very active student and participated in many student organizations, including the Institute of Electrical and Electronics Engineers, the National Society of Black Engineers and the Society of Women Engineers. While at A&T , she was chosen to participate in the Research Experience for Undergraduates, sponsored by the National Science Foundation. Participation in that summer program led to her decision to pursue a graduate degree. Dedra received her Bachelor of Science in 1996.

Dedra entered the Master of Science program at North Carolina State University under the direction of Dr. Clay S. Gloster in 1996. Her area of research is digital signal processing with a concentration in adaptive computing. Her research has been presented at technical conferences and won the National Society of Black Engineers (NSBE) Technical Paper Contest at the 2000 National Convention. Dedra was a recipient of the Graduate School's Diversity Enhancement Award. As an active graduate student, Dedra served two terms as the NSBE membership chair and one term as vice-president of the Association for the Concerns of African-American Graduate Students in addition to general memberships in other organizations. She was also actively involved as one of the few graduate volunteers in the NC State Women's Center. While in graduate school, Dedra worked to familiarize African-American undergraduates with the possibility and benefits of pursuing a graduate degree.

Dedra taught several high school courses while at NC State, which sparked an interest in teaching. After completion of her M.S., Dedra expects to teach more courses in the sciences and engineering on both the high school and collegiate levels. The knowledge and support that Dedra received from NC State have motivated her to continue her work in teaching African-American children in an effort to help younger minority scholars reach their potential.

Acknowledgements

There are so many people who are responsible for my finishing the Master of Science in Computer Engineering ...

I have to begin my acknowledgements by thanking Dr. Clay S. Gloster, Jr. I appreciate the times when he pushed me and when he didn't push me. Most of all I thank him for helping me learn how to defend my ideas.

I am also thankful to Dr. Winser A. Alexander. He is one of the most helpful *people* I know. The discussions we have had helped me in many areas of my life. It took some time for me to realize why what is expected of me is expected. He pushed me in his own strong, silent way.

I am grateful to my Daddy for directly and indirectly funding my graduate education. He knows that no matter how old and independent I seem to be, I'm always his little girl. I also thank my Mama for all her words of encouragement and small tokens to get me through. I hope I have made my brothers and sisters proud, for we act as a family and the success of one affects us all. Edward, Deborah, Dwayne and Paula have brightened my day one more than one occasion during this arduous process.

I am indebted to my two favorite ladies, Dr. Nan Manuel and Dr. Nellouise Watkins. They opened so many doors and showed me possibilities I did not know existed. I can only hope to make them proud, for their love got me here. So did the nurturing of Ms. Jocelyn Foy, whose melodic voice and caring nature helped get me into and through NC State.

I am overwhelmingly grateful to my sisters . . . in life and the struggle for the past ten years. We go back to Skidz, biker caps, interviz and privileges! Nikki A. Rogers, Carol N. McDonald, Cheryl L. McKay, LeShawndra N. Price, Darice Witherspoon and Yalaunda M. Thomas, M.D. are some of the best friends one could ask for. Most people have one or two really close friends; I am fortunate enough to have all of them. My relationships with them have never faltered, neither has their faith in me. These women have been the most consistent people in my life and have helped me make it through so many rough spots. Many times it is because of them I know that we are all going to make it ... together! I truly appreciate their friendship through the years ... all ten of them.

I am so very thankful to those who have been my inspiration and examples for me to follow. These people have received their doctorates from NC State and have proven that it can be done. I thank Dannellia Gladden-Green because she never gave me the sugar-coated version of anything. I owe Fonda Daniels-Barnes because she was there for me when I needed it and by watching her I saw what a presentation really is supposed to be. No one knows how much I thank Mabel Watson for keeping Dr. Pamela Banks-Lee occupied and off my back! Donald Miles, Donovan Moxey, Vincent Wilburn, Darrell Simpson and Corey Graves all paved the way by achieving. Sonetra Howard Wilburn has been the single most influential person in my tenure at NC State. She was my first point of contact in many situations, and was usually right. Her words of wisdom helped me so much. My academic and personal relationships with her have grown and prospered, for which I am grateful.

I am also appreciative of the presence of Chadwin D. Young, who has been my friend, my sounding board and my ally. He has helped me grow and deal with change. His words have humbled me and his focus inspired me. For what he has done and been I am eternally grateful.

I also offer thanks to my partner in the graduate student struggle. Mrs. Elebeoba (Chi-Chi) May has shown up in several chapters in the book of my life. I apologize for leaving her. We will meet again . . . we always do.

The constant nagging of Mrs. Kym Fuller has also helped me make it. It is well worth finishing this degree to not hear her fuss at me. Sabrina Simmons Warren also knows that she has put me through graduate school with her support and words of encouragement. This is "our" degree and she earned it!

"Weeping may remain for a night, but rejoicing comes in the morning" Psalm 30:7 NIV

Contents

List of Figures

List of Tables

Chapter 1

Thesis Overview

The Dynamic Resource Allocation and Management (DREAM) framework is the result of research conducted to design and develop an evaluation tool for field programmable gate array (FPGA) placement. By supplying statistics for evaluation of individual algorithms, DREAM can be used to benchmark the performance of two-dimensional placement algorithms. This document is a chronicle of the conditions that made evident the need for such a framework, the position in current and future practice where the framework will be beneficial and the development of the tool.

The introduction of the FPGA for partial execution of compute-intensive applications has brought forth several areas of research and experimentation that work to provide problems to which this newly developed computing power can be productively applied. Research has been conducted to identify applications that can benefit from the use of configurable logic devices such as the FPGA while providing improved execution time and increased flexibility. There has also been work to efficiently place functions on configurable logic devices for maximum usage. Rather than offer a single solution for efficient placement, DREAM serves as a tool to be used for evaluating existing placement algorithms [1].

This document is carefully divided into chapters that each address individual issues in the design, development and simulation of the evaluation tool, among other subjects. This chapter introduces the problem statement, research motivations, research contributions and document organization.

1.1 Problem Statement

In recent years, the area of reconfigurable computing has provided high flexibility and the ability to execute several applications simultaneously. Processing elements whose hardware can be changed one or more times can be used to execute more than one application. Applications that were previously executed solely in software can be partitioned to run in both software and hardware. This type of co-simulation creates an environment where the user has the advantage of changing the application an infinite number of times without the fabrication delay of typical methods. As the size of hardware devices continues to grow, there must be a way to evaluate the way in which applications are placed and executed.

A programmable logic device (PLD) has internal circuitry that can be arranged by the end user to suit the needs of a particular application. This ability to change the design of a PLD creates an advantage over other types of computing devices.

The general purpose processor (GPP) is a computational unit designed to execute a wide range of applications. The GPP can execute many different types of applications without specifically targeting any one area. Most desktop and laptop computers contain a general purpose processing chip. This chip contains fixed hardware that cannot be changed by the end-user. The generality of the processor configuration can limit the performance for computationally intensive applications. The GPP has a low fabrication cost and can be produced in large quantities because of its generic architecture.

An application-specific integrated circuit (ASIC) is designed to execute a specific application. ASICs are best-suited for tasks that contain large amounts of computation. The ASIC typically operates at a higher clock speed than the GPP, but can be used to execute a limited number of applications. It also typically achieves improved performance over the GPP because the design specifications associated with the ASIC are tailored for a target application. Once designed and fabricated, the hardware of the ASIC cannot be changed. The design specification makes the ASIC much less flexible than a GPP. Its design is not very easily modified, and the fabrication of an ASIC is much more expensive because its design may not apply to a large number of applications.

The FPGA-based custom-computing machine (FCCM) is used to achieve an acceptable medium between general purpose processors and ASICs. FCCMs usually combine a host processor for executing portions of an application in software and a configurable logic device for executing other portions in hardware. As the requirements of an application change, the configurable device can be readily reprogrammed. The FCCM is much more flexible than the ASIC, and uses the generality of the GPP to produce a computing system with the ability to execute a large set of applications based on some end-user specification. The increased use of custom-computing machines for compute-intensive applications, i.e. image processing, signal transmission and image compression, along with the limited hardware resources of configurable logic devices has created situations where the placement of portions of the application in configurable hardware becomes a factor in the efficient execution of these applications.

Along with the reprogramming capability of the configurable hardware device comes the potential for significant performance improvement for compute-intensive applications. When partitioned wisely into software and hardware portions, execution time for an application can be improved by several orders of magnitude [2].

This potential performance improvement often justifies loading portions of applications onto one or more FPGAs to provide more efficient execution of the application.

Various methodologies exist to identify portions of applications that are optimal candidates for hardware execution [2], [3], [4]. These methods typically use a specialized version of a C or C++ compiler to partition applications for execution on an FPGA-based custom-computing machine. There also exist methods for configuring FPGAs based on profile mappings that can be placed in many locations on a single device [5].

Partitioning applications into hardware and software candidates can improve performance, but the performance improvement neither alleviates nor eliminates issues associated with placing the hardware portions on the hardware device. The placement problem is similar to floorplanning and bin-packing. Each of these issues can be solved using some sort of evaluation tool.

In floorplanning, a finite two-dimensional space is filled with objects of varying unit size. The objective in floorplanning is to place each object in a location that will result in the maximum amount of remaining space for the placement of future objects. With an FPGA, the amount of available space for allocation is limited and the time required to find optimal placement of an application can increase execution time. Increased execution time conflicts with the performance goals of a dynamic computing system.

The bin-packing problem is also one that addresses the attempt to fill a finite space efficiently. In bin-packing, it is desired to store a set of objects by using as few fixed-space bins as possible. Bin packing is applicable to a multi-device FCCM, because in such scenarios, an object will be placed in the next bin if it doesn't fit in the current bin. Several heuristics are used to solve the problem. However, the ultimate goal is to maximize the space used while at the same time minimizing the amount of space required.

The issue of placement efficiency is significant when applications are executed on reconfigurable hardware devices. Haphazard placement of application profiles can reduce placement efficiency and not fully utilize the device resources to execute applications simultaneously. Although previous work has found acceptable placement methods [6] [7] [8], none of these efforts provide techniques for determining an optimal placement method for a set of user-defined constraints.

Using a defined placement algorithm for the position of an application on FPGAs will give the operator a method that can be analyzed for efficiency and performance. The steps followed in a placement algorithm provide an outline for determining how to position applications on reconfigurable devices. There exist few tools to evaluate FPGA placement algorithms and return statistical information to allow the user to determine the best method for a specific application. The focus of this research is to provide a tool with which to evaluate the quality of placement algorithms on FPGAs. A tool to do so provides the data for comparative analysis and allows the user to choose the algorithm to best fit the performance requirements of a particular system.

1.2 Research Contributions

The objectives of this research project are: 1) to devise and implement a tool with which various placement algorithms can be evaluated for quality of placement; 2) to study techniques for reconfiguration of programmable logic devices; 3) to identify applications that benefit from reconfiguration.

Many computationally-intensive applications are partitioned for execution in both software and hardware. The hardware portions are often loaded onto the device during the execution of the application. The development of a tool to evaluate the performance of placement algorithms on hardware devices can better assist in identifying those algorithms that succeed in reconfiguring devices on demand during run-time.

The DREAM framework is a graphical tool that evaluates the performance of FPGA placement algorithms and returns the results in a statistical report. The framework takes advantage of the reconfiguration capabilities exhibited by configurable logic devices to graphically suggest placements for mappings. In addition, the basis of the evaluation attempts to maximize the number of users and space occupied on the device. DREAM uses a data request file, a module library, placement algorithm and device size to place applications and return a set of statistics.

The framework is used to evaluate the quality of placement for programmable logic placement management algorithms. The placement quality is based largely on a cost function that calculates the amount of contiguous space remaining in two dimensions after the placement of each module. The cost functions were created as part of the research as the basis for evaluation. The placement algorithms evaluated were modified versions of common algorithms or algorithms created by the researchers. DREAM has different modes to simulate several individual hardware device scenarios. In addition to the algorithms that are evaluated in the following research, other placement algorithms can readily be incorporated into DREAM to evaluate placement quality. The statistical report provided by DREAM can be used for a comparative analysis given a set of placement algorithms.

1.3 Document Organization

The following chapters of the document detail the project, the approach and results. The document is organized as follows: Chapter 2 provides relevant terminology and background in the area of reconfigurable computing and run-time reconfigurable systems. The chapter also justifies the need for the DREAM framework. Details of the DREAM framework implementation are presented in Chapter 3. Three placement algorithms are incorporated into DREAM in Chapter 4. Chapter 5 presents the experimental setup and results achieved. Finally, Chapter 6 presents conclusions and tasks for future research.

Chapter 2

Related Work and Current Practice

The area of reconfigurable computing is fairly new, hence terms used by different research groups often have different interpretations. A list of terms is presented to familiarize the reader with the terminology used. The terminology is followed by a section on the background and evolution of the configurable logic device and the advent of reconfigurable computing systems. In addition to specific applications, improvements in both performance and logic capacity are also discussed here. Numerous papers related to partitioning applications into hardware and software candidates are discussed, in addition to those that present individual placement methods.

2.1 Reconfigurable Computing Terminology

- Configurable logic: Digital logic circuits that can be programmed by the end user more than once to realize circuit designs.
- Reconfigurable logic: Digital logic circuits that can be programmed an infinite number of times to realize circuit designs.
- Programmable logic device (PLD): Generic name given to any digital logic device capable of being programmed by the end user to realize circuit designs.
- Logic capacity: Amount of digital logic that can be mapped into a PLD

- Programmable Logic Array (PLA): Logic device designed to implement logic with two levels. It contains a programmable AND-plane and a programmable OR-plane.

- Programmable Array Logic (PAL): Logic device designed to implement logic with two levels. It contains a programmable AND-plane and a fixed OR-plane.

- Simple Programmable Logic Device (SPLD): Term given to programmable logic devices with a small logic capacity (200 gates or less); many PLAs and PALs are considered SPLDs.

- Complex Programmable Logic Device (CPLD): Logic device that provides logic capacity up to the equivalent of 50 simple PLDs

- Mask Programmable Gate Array (MPGA): Highest capacity general purpose logic device available; pre-fabricated with a sea of logic gates to be personalized with one or more metal layers.

- Field Programmable Gate Array (FPGA): User-programmable equivalent of MPGA, with smaller logic capacity and less fabrication expense.

- Fully reconfigurable PLD: PLD in which the entire device must be configured regardless of which portions are to be changed.

- Partially reconfigurable: PLD in which portions of the device can be configured while other portions continue functioning.

- Reconfigurable Computing (RC) System: Computer systems that consists of a host processor connected to a PLD.

- FPGA-based Custom-Computing Machine (FCCM): Computer system comprised of a host processor connected to an FPGA for software/hardware co-processing.

- Run-time reconfiguration (RTR): Process of (re)configuring a PLD dynamically during the execution of a particular process or application.

2.2 Programmable Logic Devices

Configurable logic is a viable alternative method to the general purpose processor and the application specific integrated circuit (ASIC) for solving scientific and engineering problems. At one end of the hardware implementation spectrum is the general purpose processor (GPP) which provides acceptable performance for a broad range of applications. As the name implies, the GPP is manufactured to implement applications varying from very simple functions such as logic expressions to fairly complex functions, i.e. weather prediction. Though low in system cost with low development time, the GPP can be slow for compute-intensive applications. However, the main advantage to the GPP is the ease of programmability when simple revisions are required. Most desktop computers contain a GPP because they must be manufactured to accommodate a wide range of applications.

Alternatively, the ASIC can obtain significant performance improvement over the GPP, but for a concentrated set of applications. Each lot of ASICs is completely configured to the specifications of the end-user, but during the manufacturing process. Application development time is on the order of several months to a year and the amount of specification required to customize each device results in a relatively high fabrication expense. This expense can be justified by the increased performance of the ASIC. One disadvantage of fabricating an ASIC is the amount of work required to revise a design when changes are made to the device.

Programmable logic provides an option that is a hybrid between a GPP and an ASIC. With programmable logic, the end-user can customize hardware for specific applications and completely change the function whenever necessary. Programmable logic devices are configured by the end-user, an action that decreases the amount of specification, thus resulting in faster application development. Most PLDs can also be programmed a large number of times, which allows flexibility of application implementation approaching that of the GPP. The increased logic capacity and reprogramming capabilities, have made the PLD a very popular choice for many applications.

The Programmable Logic Array (PLA) was the first device developed for implementing programmable logic in hardware. This relatively small logic device contains two levels of logic: a programmable AND-plane and a programmable OR-plane. Logic functions represented in sum-of-products form are prime candidates for implementation on the PLA [9]. However, the restriction of using two levels of logic may result in poor performance when the number of gate inputs is large.

The Programmable Array Logic (PAL) was introduced by Advanced Micro Devices to counter the disadvantages of the PLA [9]. The PAL, also small, contains two levels of logic: a programmable AND-plane and a fixed OR-plane. Several versions of the device were manufactured to vary the number of inputs on the fixed OR-plane. Many PLAs do not contain flip-flops, thus accommodating only combinational logic. The PAL may contain flip-flops and increase its capabilities to support sequential logic. The PLA and PAL comprise a group of programmable logic called simple PLDs (SPLDs). The logic capacity of these devices is approximately 200 gates or less. A major disadvantage of SPLDs is that the active circuit area of the device grows at an unproportionately fast rate with increasing logic capacity.

Logic devices with high gate capacity, equivalent to more than 50 SPLDs, are called complex PLDs, or CPLDs. CPLDs not only contain programmable logic, but the structure of the device can be configured more than once to realize circuit designs. CPLDs consist of multiple logic blocks interconnected together via a programmable switch matrix. The CPLD has a larger number of registers available and virtually any logic function can be reproduced using these devices [10]. In this thesis, both Mask

Programmable Gate Arrays and Field Programmable Gate Arrays are considered CPLDs, due to their large gate capacity.

CPLDs can be programmed by several methods. Circuit designs realized with Erasable Programmable Read-Only Memory (EPROM) can be reconfigured using ultra-violet light. Similarly, Electrically Erasable Programmable Read-Only Memories, EEPROMs, uses voltages to replace the current contents of the device with a new circuit design.

The highest capacity of logic available in a general purpose chip is in the Mask Programmable Gate Array (MPGA). The MPGA is customized with wires connecting an array of pre-fabricated transistors. The specification of the metal gate interconnect customizes each chip to the needs of the user. Fabrication of the MPGA is less costly than that of the ASIC since many device fabrication steps are performed ahead of time. Development time is greater than the GPP because of the manufacturing steps required.

Like the MPGA, the Field Programmable Gate Array (FPGA) contains an array of logic blocks, and I/O blocks. In contrast to the MPGA, the FPGA contains prefabricated interconnect that is also reprogrammable. While the logic of the MPGA is configured by the manufacturer, the FPGA can be programmed by the end-user. It has less logic capacity than the MPGA, but justifies its use by the lower cost associated with user-programmability. As the user-programmable equivalent of the MPGA, the FPGA can be programmed more than once to implement and execute various circuitry. Development time for the FPGA is less than the MPGA. Although technically possible to program an FPGA with either EPROM or EEPROM, most FPGAs are either anti-fuse or Static Random Access Memory (SRAM) based.

The anti-fuse based FPGA is a one-time programmable device. The configuration of the FPGA is permanent once an electrical connection between the terminals is established. Anti-fuse based FPGAs are not volatile and remain configured when the system is powered off and restarted. Conversely, the programming flexibility of the SRAM based FPGA contributes to its common use in reconfigurable computing systems. The logic of the SRAM-based FPGA can be reprogrammed, or reconfigured, an infinite number of times to realize a different circuit design after its initial programming. Although volatile, the device configuration can easily be loaded from a ROM when the system is powered up. With its many advantages in programming flexibility and range of applications, the SRAM-based FPGA has become the "workhorse of many new reprogrammable applications." [11]

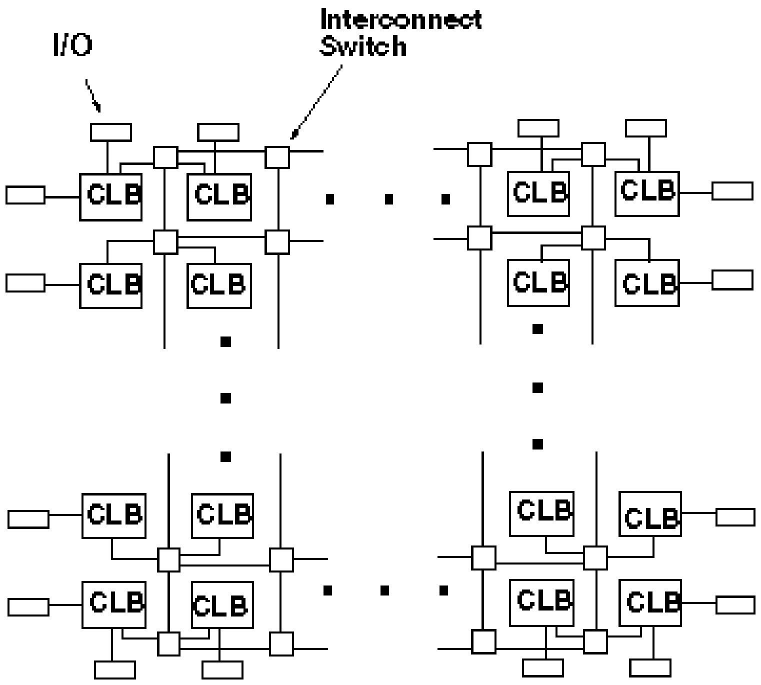


Figure 2.1: A Generic PLD Architecture

Figure 2.1 illustrates a generic PLD architecture. Each configurable logic block (CLB), which typically contains a few flip-flops, may implement any Boolean function of up to five variables. Each CLB is connected to all other CLBs in the device via programmable interconnect switches. These switches manage a two-dimensional mesh of wires to connect the CLBs to implement a user-specified application. The

I/O pins around the perimeter of the chip can be programmed as input, output or bi-directional.

2.3 Reconfigurable Computing Systems and Run-Time Reconfiguration

This research is based on application implementation on reconfigurable computing systems. Generally comprised of one or more FPGAs connected to a host processor, a reconfigurable computing system can be used to execute a number of applications that require both hardware and software resources for implementation. In instances where device configuration is altered during execution, the system is labeled as *run-time reconfigurable*. When a portion of the device can be configured while the rest of the device continues functioning (without interference), the device is classified as a partially dynamic reconfigurable device.

Many types of architectures have been developed as a result of the introduction of the reconfigurable computing system. These reconfigurable systems can be used to decrease execution time for compute-intensive applications. These reconfigurable systems use programmable logic to implement functions that were previously executed totally in software.

In [11], Hauck investigated the changing role of FPGAs in systems where reconfigurable logic is beneficial [11]. This research concentrates on SRAM-based FPGAs, the advantages of their use, and evolution of their significance as system requirements change. A key point delivered in the paper was the evolution of the FPGA from a logic emulation tool to an integral part of the reconfigurable computing system. In its early development, the FPGA showed the greatest performance potential for logic emulation, testing a circuit design to verify its behavior. An ASIC under development is mapped onto one or more FPGA devices and the logic emulation system is placed in the actual environment for which the ASIC is designed. Although the logic emulation system runs at a slower clock rate, the ASIC can be completely configured. This study suggests that the SRAM-based FPGA is also ideal for prototyping, and the FPGA has evolved from a device used for general glue-logic replacement to the foundation for a wide range of systems that require reconfiguration.

Run-time, or dynamic, reconfiguration occurs when a hardware device is configured one or more times during execution. An application is loaded onto a hardware device in stages, or portions, each of which is executed separately. Modifications to the existing configuration can be either partial or full reconfiguration, based on the abilities of the device. Run-time reconfiguration (RTR) systems are a specific subset of FCCMs and can be used to emulate ASICs with the FPGA-host processor combination.

Because of the real-time performance constraints, RTR systems require quick placement of application modules. Here, reprogramming time is crucial and any introduced delays may impede the performance of the applications. Communication delay can be reduced by tightly coupling the FPGA with the host processor. However, the inefficient placement of modules can hinder effective execution of an RTR system. The research in this thesis will provide a method for evaluating the quality of placement when using application modules based on several performance variables. The framework introduced builds upon current research that has investigated the performance improvement resulting from reconfigurable logic and the concept of creating methodologies to place application modules on FPGAs.

Many researchers have investigated the performance of applications separated into candidate modules for software and hardware execution [2] [12] [3]. Architectures resulting from such investigations have been used to prove the advantages of partitioning applications to improve performance.

One method for partitioning compute-intensive applications into hardware and software portions was created in [3] to investigate the effectiveness of such practices. One objective of this research is to "define the class of programs that are well-suited for acceleration" with software/hardware partitioning. The method presented in [3] is primarily based on three criteria: the collection of profiling data, a candidate preselection scheme, and a high complexity hardware partition. In addition to the criteria mentioned above, the estimated performance of the hardware portion is considered. Complex operations such as floating point operations are allowed to reduce execution time.

Software portions of the application are translated into VHDL processes after being identified as conducive to performance improvement using hardware/software co-execution. The resulting VHDL models are then processed by a synthesis system. This systems prepares the VHDL to be merged with the portions of the application to remain in software via communication code. The work concluded that speedup is a result of both choosing a small candidate region and minimized memory access frequency. Those portions of the application that can be executed with minimal memory access frequency result in less communication delay and better performance.

PRISM (Processor Reconfiguration through Instruction Set Metamorphosis) is a general-purpose computing platform that can speed up and improve performance of compute-intensive application execution [2]. The system is used to convert applications generally executed solely in software into hardware and software candidates. Based on previous research that reported orders of magnitude performance improvements to applications by converting small portions into hardware [13], the concepts presented in [2] can reduce execution time for software applications that are effectively partitioned and executed. It is desired to achieve a balance between the software and hardware candidate portions.

The methodology presented in [2] has four steps for partitioning software applications. The steps, which include identifying candidate segments for execution in hardware and converting the segments into functions represented by data flow graphs (DFGs), produce a hardware/software application that significantly reduces the execution time and improves performance. In identifying candidate segments, the focus is to extract those segments that will have the greatest impact on the performance of the entire application. As is the case in many compute-intensive applications, the majority of the required computations are contained in a small portion of the program code.

The candidate segments identified for hardware execution are ultimately converted into function descriptions represented by the DFG. An access definition file created from the DFG is merged with the remaining software portions of the application to create a new C program. This new program, compiled for optimization, automatically loads the hardware portions of the application onto the FPGA for execution. The automatic execution of the hardware portions is an action used to validate the feasibility of the framework produced in this thesis.

Athanas et. al verify the functionality of the PRISM architecture. For certain applications, reconfiguration of the FPGA requires less than one second. Partitioning of the hamming function demonstrated a speedup of 24 times the execution of the application in software alone. By isolating and translating compute-intensive sections of programs such as hamming distance and error correction, PRISM creates an environment where the user can achieve results similar to application specific platforms without extensive knowledge of hardware design.

The CC-DSP is presented by Bergmann et. al as a "custom-computing architecture . . . specifically designed for efficient implementation of DSP algorithms. " [12] In an attempt to address the weaknesses of other custom-computing systems [14] [15] [16] [17], this architecture can accommodate the high gate count often associated with arithmetic operations processed in many parallel DSP applications. The CC-DSP was compared to several other computer architectures to demonstrate the merit of the CC-DSP as an architecture more suited to the complicated operations involved in DSP algorithms.

The architecture comparisons are based on assessing the number of complex arithmetic operations as well as the "cost" of each board, based on the number of hardware devices. The CC-DSP was shown to provide a moderate performance improvement for floating-point operations. However, as a single-board approach, it gives no improvement over parallel processing architectures.

After the performance improvement of partitioning compute-intensive applications was shown, other research focused on the advantage of using reconfigurable devices to "page" data instruction sets in and out of the device itself. Due to the reprogrammable nature of configurable devices, instruction sets can be programmed when needed, removed and replaced by other instructions once executed and no longer needed. By taking advantage of the reprogrammable capability of such devices, a reconfigurable computing system can, in many ways, act as an application-specific machine. One such architecture is the Dynamic Instruction Set Computer, or DISC [5].

DISC creates an application specific processor by taking advantage of FPGA partial reconfigurable capabilities. Because the circuitry of a reconfigurable device can theoretically be modified an infinite number of times, the instruction data can be piped in and out as required by the demands of the executing program. The concepts of DISC show many similarities to the research presented in this thesis. The notions exercised are applicable to the guidelines and corollaries established.

DISC systematically replaces idle application-specific instruction modules with usable modules to provide an essentially limitless application-specific instruction set. Each instruction in the set is implemented as an independent circuit module and paged in as dictated by the program. The partial reconfiguration ability, existent in many PLDs, configures a section of the device without affecting the execution of existing logic. In this manner, the FPGA serves as a cache of the most recently used instructions by replacing the least used modules with more recently used ones. This results in the implementation of larger instruction sets than those allowed with traditional execution.

For added flexibility, DISC modules can be placed at multiple locations on a single device. This relocatable ability increases the placement options when instruction modules are paged in to the device. In a manner similar to the framework presented in this thesis, the physical layout of a module profile affects neither the profile nor placement of existing modules on the device.

The papers presented in this section verify the performance advantages of partitioning an application into software and hardware segments. Regardless of the computing architecture used, the framework presented in this thesis can be used once the hardware segments are profiled into mappings for placement. The framework does not attempt to partition the applications, but acts as a tool with which the quality of the placement algorithms can be evaluated and compared.

2.4 FPGA Placement

The programming flexibility and parallel computing potential of run-time reconfigurable systems prove the advantage of implementing computationally intensive applications on such a system. Although performance improvement for compute-intensive applications can be achieved with reconfigurable systems, other substantial issues remain in their development. Reconfigurable systems have introduced the feature of placing hardware profiles in numerous locations of a PLD, but without any guidelines for determining placement. Determining the quality of placement of circuits on reconfigurable devices can maximize usage of the device and may reduce the amount of reconfiguration required, i.e. removing or moving one module to add another.

Bazargan et. al present a placement heuristic for fast placement of logic modules on a programmable logic device, referred to as a reconfigurable functional unit (RFU) [6]. This *online* method is a hybrid between the traditional best-fit and first-fit placement algorithms. The goal of this work is "to devise fast methods for placing RFU operations on the chip tightly, hence placing more modules on the chip."

The fast, though not optimal, method presented is an extension of the best-fit, first-fit and bottom-left placement methods. After each placement, the remaining empty regions of the device are separated into "empty rectangles" to facilitate placement. The empty rectangles do not overlap any modules and can be grouped to place the next module. The RFU operations (RFUOPs) are independent of each other and can be placed anywhere on the device.

In a similar work, [7] the authors present a methodology for mapping large circuits to FPGAs. Current FPGAs have a logic capacity of more than one million equivalent gates. With such a large variety of gate designs available for configuring a device, creating macro designs before application execution can prove advantageous when attempting to alleviate some placement problems.

In [7], Emmert and Bhatia create a fast floorplanning algorithm for large FPGAs that can quickly utilize the large number of available gates for realizing circuit designs. The macro designs are used to provide better performance than profiles that are created during execution and may degrade system performance. A set of macros are created for the execution of each application that have either hard (permanent) or soft (variable) shapes and a unique location. The macro itself is a set of interconnected and relatively placed logic blocks. The set of macros are floorplanned on a two-dimensional array of available device locations.

The macros are planned in three phases: topological placement, legal floorplan and reshaping. The topological placement treats all macros as surface entities with no regard to the actual physical dimensions. The topological placement may contain some block overlapping. Once placed, a legal floorplan is created. At that time, all overlapping macros are repositioned to fix all blocks within the fixed area of the FPGA. Lastly, any available soft macros are reshaped to fix the confines of the fixed device area.

Mapped on a Xilinx 4000 series of FPGAs, the methodology offered supplies a solution to quickly and efficiently map application profiles on an FPGA. The utilization of macros in device placement cut down on the design time and has proven to be successful in implementation. The profiles and macros used have created benchmark examples for use in evaluating the effectiveness of the solution.

Another solution for improving placement on hardware devices is offered in [18]. The paper discusses the trade-off between placement quality and compile time on large capacity devices. As the equivalent gate size of programmable logic devices grows, compilation time has also grown. The increase in compile time has reduced the advantage of quick turnaround in PLDs.

The tool presented in [18] minimizes area of the PLD while at the same time reducing the compilation time for large circuits. Similar algorithms have succeeded in reducing the wiring area required, but with no improvement in compile time. The approach discussed here reduces compilation time by diving large problems into a smaller set of manageable problems to reduce the complexity of the circuit.

The smaller set of problems are handled as logic blocks clustered to create a larger application. The clustering of these blocks can be varied to balance the compile time/quality of placement trade-off. The goal is to create a clustering with the necessary parameters to result in the best trade-off. Implemented within a framework of the Versatile Place and Route (VPR) tool, the clustering algorithm reduced both the wirelength requirement and run-time of larger applications.

The previous research and related work have demonstrated the advantages of partitioning large computational-intensive applications into candidates for hardware and software execution. Other solutions offer ways to increase the efficiency of placement on a programmable logic device and show the performance improvement acquired in doing so. Though each of the works cited and summarized contributes to the improvement of programmable logic utilization and placement efficiency, the solutions may not be optimal for scenarios in which either the computer architecture or application type is not identical to that in the paper. The previous research has offered no method for the user to evaluate the performance of individual methods based on benchmarked experimental results.

In following advances made in the area of reconfigurable computing, this thesis presents a framework to take full advantage of the reconfiguration ability and increased logic capacity of PLDs. Additionally, the tool helps users to alleviate placement problems by providing a basis for determining quality of placement. The DREAM framework for reconfigurable computing systems can evaluate the quality of placement for any algorithm incorporated. DREAM offers a statistical report that can be used for a comparative analysis to choose the most effective placement algorithm for individual execution scenarios. The tool, its components, features and user modes are discussed in the following chapter.

Chapter 3

The DREAM Framework for Relocatable Hardware Devices

Previous research has identified a need for a framework for evaluating and managing placement of hardware application profiles on reconfigurable devices. The Dynamic Resource Allocation & Management (DREAM) directly addresses this need. DREAM takes full advantage of programming flexibility exhibited when implementing applications on reconfigurable systems. Written in Java, DREAM is a tool that evaluates placement algorithms for configurable logic devices. DREAM is a framework wherein placement algorithms can be executed and the performance and quality of each placement algorithm evaluated using a cost function.

DREAM places modules according to a user-specified placement algorithm and provides feedback that can be used for a comparative analysis of several placement algorithms. While the DREAM framework can support placement of typical modules that require pin-to-pin connections, it can also support FPGA devices that allow module placements that are not limited to those connection requirements. These devices hold both the input and output data for the application in internal registers. In the DREAM environment, such modules are defined as relocatable.

Relocatable hardware module (RHM) are completely independent of each other. That is, placement of one RHM will not affect the execution of another. Each RHM can be inserted or deleted at any time during the execution of one or more existing RHMs or applications. This mapping flexibility allows maximum utilization of FPGA resources. A tool like DREAM is valuable in the identification of effective placement algorithms for real-time placement of RHMs in run-time reconfigurable systems.

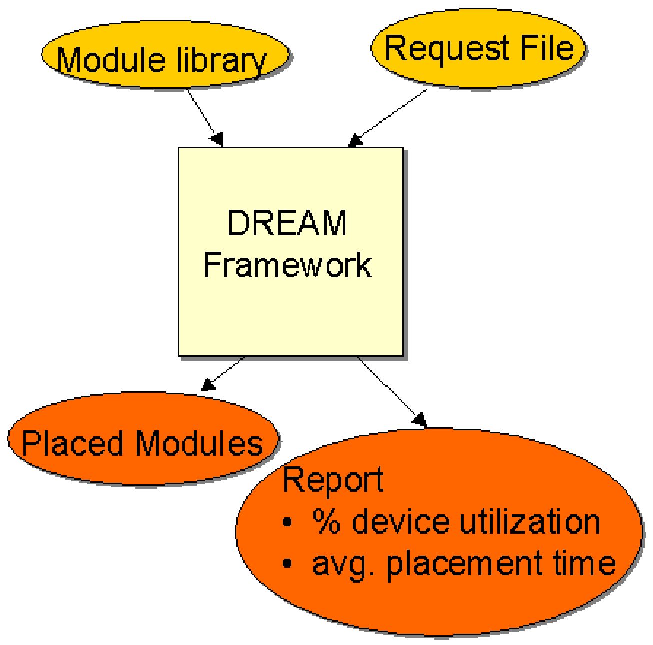


Figure 3.1: A High Level Overview of the DREAM Framework

Figure 3.1 presents a high level diagram of DREAM including the output produced by the software. The DREAM framework accepts as input a module library, defining the profile of all potential modules to be placed, and a data request file. The data file simulates the request of hardware resources in a real-time environment. DREAM produces both a statistical report and a final placement, after processing all requests for insertions and deletions found in the request file. This report includes the average device utilization, average time to place each module, total placement time and the number of denied insertion requests. The report and the final placement can be used to assist the user in selecting the *best placement algorithm* for a specific profile of user requests.

3.1 The DREAM Graphical User Interface

The DREAM graphical user interface has three main components which are shown in Figure 3.2. These are: the FPGA fabric, the placement algorithm selection area and the DREAM feature selection area. The FPGA fabric is the area where user-specified modules are placed. It is a two-dimensional graphical representation of the CLBs on an FPGA. Since DREAM typical FPGA CLB arrays are square, we designed the framework to handles square grids only. Hence an m x m grid is defined as a grid of size m (default = 100). In subsequent figures of this thesis please, note that the light gray areas on the fabric are unoccupied spaces where RHMs can be placed.

In Figure 3.2, several RHMs are shown in a sample placement. There are several instances of ADD2, a module used in our experiments. Instances of ADD2 exist in the upper left corner of the FPGA fabric, as well as in the upper right and lower left. A square module is placed three times in the FPGA fabric, in both the upper and lower right corners and the lower left corner.

The DREAM placement algorithm selection area lists all placement algorithms available and allows the user to select a particular algorithm. The feature selection buttons can also be used to step through a placement and to save a partial placement for completion at a later time. With the DREAM framework, it is simple for the user to add new placement algorithms.

An RHM, or module, is a graphical representation of a hardware macro that has been previously mapped onto an FPGA. The list of potential modules is stored in

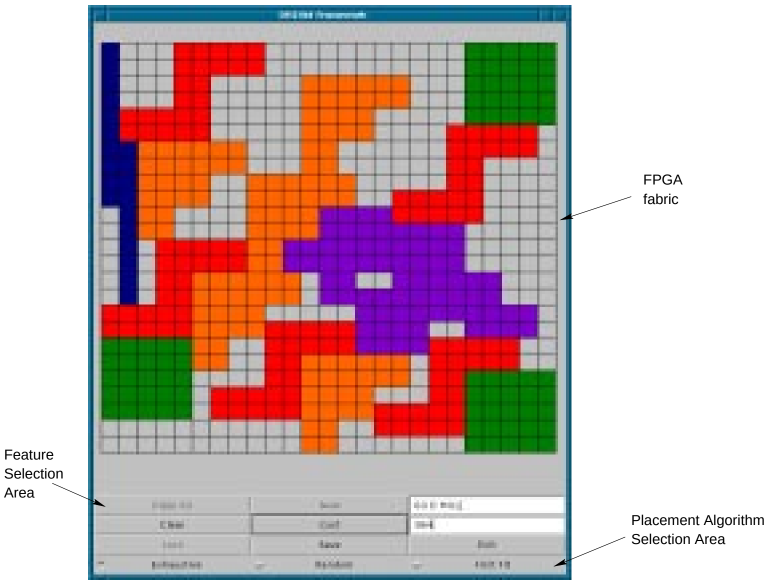


Figure 3.2: The DREAM Framework Window Showing a Sample Placement

the file “Module.lib”. A color assignment and the total number of CLBs used by the module are included as a part of the module definition. The profile of the module is represented by a set of ordered pairs that indicate the (x,y) coordinates of each CLB used in the module, along with a unique module name. The module origin (0,0) is the upper left-hand corner of the module.

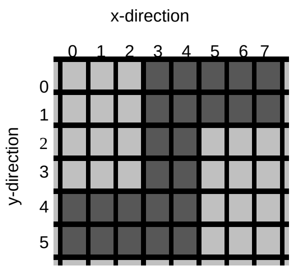


Figure 3.3: An Example Module Profile

For example, the module shown in Figure 3.3 has the following profile associated with its definition:

```
ADD2
5 24
```

`(3,0), (4,0), (5,0), (6,0), (7,0), (3,1), (4,1), (5,1), (6,1), (7,1), (3,2), (4,2), (3,3), (4,3), (0,4), (1,4), (2,4), (3,4), (4,4), (0,5), (1,5), (2,5), (3,5), (4,5)`

The name of the module is ADD2. ADD2 is red, the color assigned to 5, and occupies 24 CLBs, listed by (x,y) coordinate pairs. In Java, each module can have one of 768 possible colors. The module definition typically includes wiring associated with the RHM mapping, but this is not a restriction.

The FPGA fabric, or place where the modules will reside, is a two-dimensional grid representing the CLBs on an FPGA. Figure 3.4 shows an FPGA fabric of size $m = 10$, (10 x 10), and a module placed on the fabric. The origin of the grid (0,0) is the upper left corner of the FPGA fabric. Similarly, the upper left corner of the bounding box containing the module is the origin of the module. In this example the module ADD2 is placed at grid location (1,2). Neither the modules nor the grid ever contain negative coordinates.

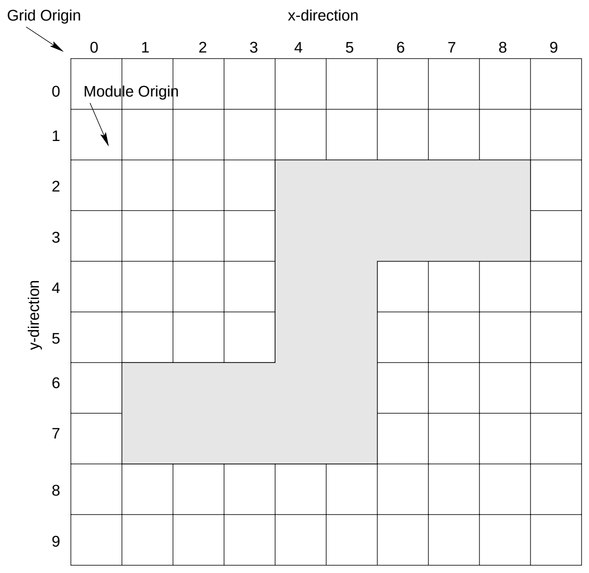


Figure 3.4: An FPGA Fabric and Module Origins

3.2 Features of DREAM

The DREAM framework has several features to facilitate the successful evaluation of placement algorithms. These features are controlled by the user via a placement algorithm selection area The feature selection area and various command line options also allow the user to change the parameters and look of the DREAM framework.

3.2.1 User Selection Area

The DREAM framework contains a selection area to allow the user to choose different options for program execution (shown in Figure 3.5). With these features the user can perform a number of tasks while evaluating placements.

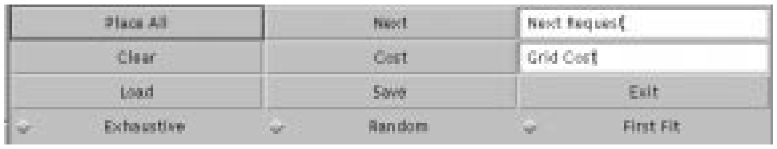


Figure 3.5: The DREAM User Selection Area

The “Place All” button processes the entire request file and shows the final placement after all requests/deletions have been processed. This button can be used at any time before exiting the program. The “Next” button processes the request file one request/deletion at a time while the line being processed is shown in the text field to the right of the button. The FPGA fabric is updated after each request in the data request file is processed. “Clear” erases the entire grid and disposes of any previously placed modules. This is used in situations where the user may choose to simulate removing all previous modules from a hardware device and placing another set of applications or to simply start over. The cost of the current placement can be calculated on demand and shown in the adjacent text box by using the “Cost” button.

The current placement can be saved to a file ("Modules.dat") for future use with the "Save" button. Any previously saved placement can be retrieved with the "Load" button. (Please note: This button can be used only on an empty fabric.) The "Exit"button simply stops execution of the program and closes the application. At the bottom of the user selection area are the placement algorithm choices. As the user incorporates other placement algorithm, the new algorithm can be added easily to the user selection area. The steps for incorporating algorithms are discussed in Chapter 4.

3.2.2 Placement Modes

The DREAM framework can be executed in two placement modes. The first mode maps RHMs requested by multiple users to a single hardware device (MUSD). In this mode, the DREAM framework places all RHMs on the same device and simulates an environment where only one device is connected to the host processor. This is the default mode in which the DREAM framework operates and is shown in Figure 3.6.

The second mode is the multiple user, multiple device (MUMD) mode, shown in Figures 3.7. In MUMD mode, RHMs are requested by multiple users and can be placed on one of many devices. This multiple device scenario is similar to an environment where one or more FPGA boards, each containing multiple devices, are connected to each other and the host processor via I/O interfaces. In MUMD mode, DREAM has been limited to twenty-five (25) devices. Larger numbers of devices will be difficult to view. The user can modify the grid size during execution of the program. An upper limit of the size of the grid has been set at $m = 150$. Larger FPGA fabrics will be difficult to view on a typical monitor. displayed accurately on a general computer monitor. For example, Figure 3.7 shows one FPGA board containing four FPGA devices.

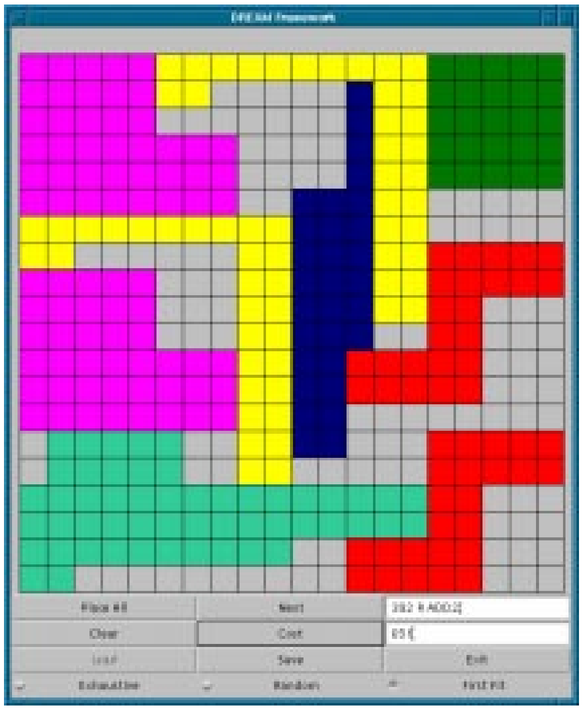


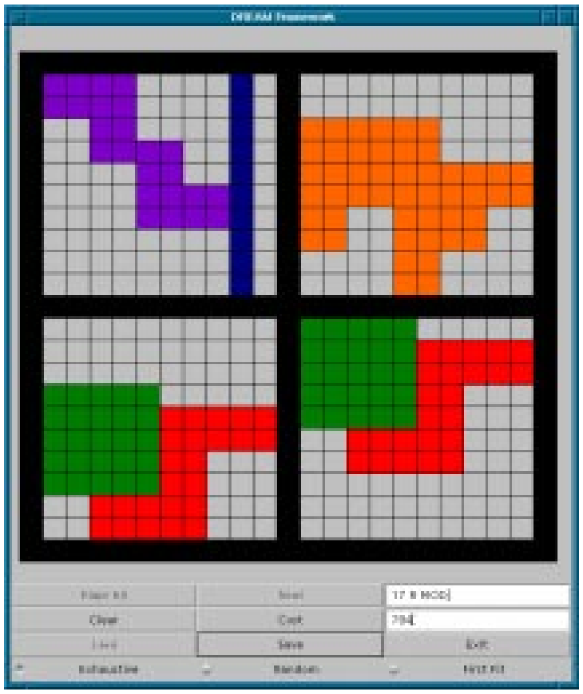
Figure 3.6: A Sample Placement using MUSD User Mode

3.2.3 DREAM Command Line Options

The DREAM framework responds to several instructions that can be implemented from the command line. These command line options allow the user to determine the mode in which the framework operates, the size of the FPGA fabric modeled and manual or automatic placement of the candidate modules.

PARAMETER	COMMAND	ADDITIONAL REQUIRED VALUE
Help Option	-help	
User Mode	-musd	
	-mumd *size*	Individual device size (natural numbers)
Process Mode	-man	Placement Algorithm
	-auto	Placement Algorithm
Fabric Size	-size *size*	Size of fabric (whole numbers)
Tentative Placments	-tent k	Number of tentative placements (natural numbers)

Table 3.1: DREAM Command Line Options



17 8 MOD

794

Figure 3.7: A Sample Placement using MUMD User Mode

Table A.1 lists each of the parameters that can be modified on the command line as well as any additional values that are needed. Executing DREAM with the help option prints information on how to use all available features. With the command line options, the user can choose in which mode the DREAM framework will be executed. For example, the user can choose from MUSD, the default mode or MUMD. As explained above, each mode simulates device usage in different scenarios.

The DREAM framework displays a list of valid commands when the user enters "-help" on the command line. This list is also displayed when an invalid command is entered. This help options gives the proper commands to be used as well as any additional required value for a command option.

The way in which the data request file is processed can be determined before execution of the DREAM framework begins. The user can also decide whether to manually process the insertions/deletions in the file or to have the entire file processed automatically by using the command options "man" or "auto". To allow the user to process each line of the request file individually, the “man” command is used. This is the default mode for the DREAM framework. If the request file is to be processed entirely before displaying the final placement, the “auto” command is used. However, at any time during manual processing, the “Place All” button on the feature panel can be pressed to achieve the same results as using the “auto” option.

When exercising the process mode option, the user also must select an algorithm for the framework to use. Unlike the graphical interface where the placement algorithm is chosen before placing any modules, the command line option must choose the algorithm without any type of panel available. The options for the placement algorithm are “exhaust” for exhaustive search, “first” for first-fit, and “rand” for random placement. The random placement algorithm will search a default number of locations for a module. The user can change the default value with the “search” parameter on the command line. If no options is chosen on the command line, the framework defaults to the first-fit algorithm. In the following command line, the random algorithm is selected to process the entire data request file. For this run, 75 tentative locations will be tried for each module to be placed.

```
> java Dream -auto rand -tent 75
```

The size of the entire FPGA fabric can be determined by the user of the DREAM framework. By entering “size” followed by the size of the fabric, as a positive integer, the user can alter the size of the fabric from the default fabric size of 100.

The command line options allow the DREAM framework user to modify the execution parameters to fit the evaluation of each specific application. If the options are not exercised for any of the parameters, the default settings are used. The command line options can be entered in any order after the requisite Java commands are are used to begin execution of the software. For example, if the size of the entire FPGA fabric is to be changed to 50, the command line would read:

```
> java Dream -size 50
```

Similarly, if the software is to be executed in MUMD mode with 10 CLBs in each device, the changes to the command line would be as follows:

```
> java Dream -mumd 10
```

To combine two or more commands, the user can add them to the end of the command line. To automatically run the software with the random placement algorithm and a fabric size of 75 CLBs, the user would type:

```
> java Dream -auto rand -size 75
```

3.3 DREAM Data Request Files

The data request files used in the DREAM framework follow a specific, pre-determined format. Each line of the file contains one request and ends with a semicolon. Each user request consists of three fields. The user identification number if the first field of the request. Each user must have a unique ID number. The next field of the request is the insertion/deletion field. It must contain either an 'R' denoting a request to place a module or 'D' to delete an existing module. The third and last field of the request is the name of the module. Excerpts of an actual data file containing 512 lines for a library of 8 possible RHMs, are shown below.

```
0 R REG;
158 R DIV;
1 R ADD;
1 D ADD;
2 R MUL;
155 R ADD2;
```

```
125 R DIV;

509 R MOD;

4 R ADD2;
```

Chapter 4

Evaluating Placement Algorithms with the DREAM Framework

The DREAM Framework for Reconfigurable Hardware Modules (RHMs) is designed to evaluate the quality of placement for any placement algorithm incorporated within the framework. The details of each evaluation are determined by the individual constraints and requirements of the executing application. However, the foundations of the evaluation are standard. A simple cost function serves as the basis for evaluating the quality of each placement. The procedure and requirements of each placement algorithm will vary the resulting placement for identical sets of data.

Each time a portion of the FPGA is requested by the user for module placement, DREAM uses several factors to determine final placement of the module. The most significant factor is the algorithm. This will identify a potential placement for each RHM. The cost function assists in determining precise placement of the module. For the statistical report, cost along with device utilization are reported.

4.1 An Efficient Cost Function to Measure the Quality of Placement

The DREAM framework is built around a cost function that is used to indicate the quality of placement. This simplistic cost function calculates the amount of contiguous space left in two dimensions on an FPGA after the placement of each RHM. This function is based on the assumption that *a placement of the current module resulting in a large contiguous space in two dimensions will increase the probability of successfully placing the next module whose shape is unknown at the current time.* An empty grid has zero cost and a full grid has cost of $2m^2$ for a grid of size m. The three equations for the cost function are as follows:

$$Cost_T = Cost_x + Cost_y \tag{4.1}$$

where the total cost of the FPGA fabric, $Cost_T$ is the sum of the cost of the placement in the x-direction and the cost of the placement in the y-direction. The cost in the x-direction is calculated as follows:

$$Cost_x = \sum_{i=0}^{m-1} (m - C_i^x) \tag{4.2}$$

where C_i^x is the maximum number of contiguous empty or unoccupied cells in the x-direction of the i^{th} row of an m size FPGA fabric. Since row i of an m size FPGA fabric has m cells, $m - C_i^x$ is the smallest when all cells in row i are empty or when $C_i^x = m$.

The DREAM program scans each row for empty CLBs. Unoccupied CLBs located next to one another are *contiguous*. An array *Contiguous(j)*, j = 0, 1, 2, ..., m-1, stores the number of contiguous cells in row i, beginning with column j. The maximum value over all j is C_i^x.

Figure 4.1 shows a sample placement for an FPGA with size $m = 20$. On row i = 2, there are two sections with 2 contiguous CLBs [Contiguous(1) = 2, Contiguous(5) = 2]and one section with 1 unoccupied CLB [Contiguous(18) - 1]. Since all other values of the array (contiguous) are zero, $C_2^x = 2$. Therefore, for row 2, the value of $m - C_2^x$ = 20 - 2 = 18.

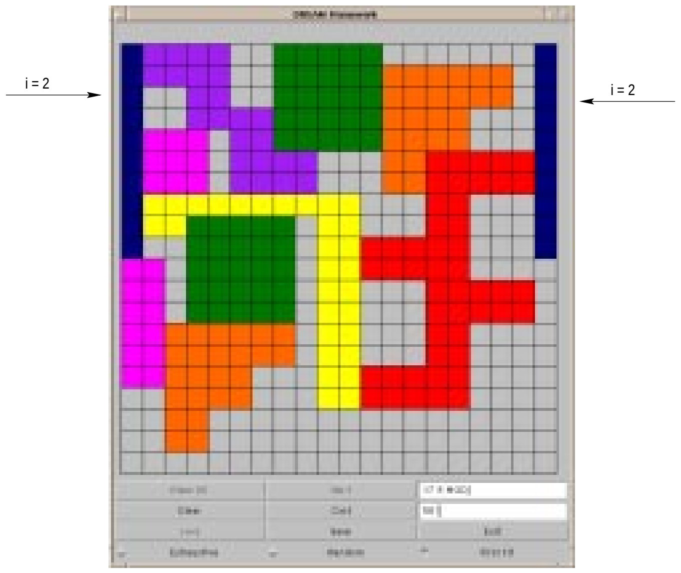


Figure 4.1: Cost Function: Evaluating the Cost in the X-Direction (m = 20)

In a similar fashion, the cost in the y-direction is calculated over all columns as follows:

$$Cost_y = \sum_{i=0}^{m-1} (m - C_i^y) \tag{4.3}$$

where C_i^y is the maximum number of contiguous cells in the y-direction of the i^{th} column of an m size FPGA fabric.

Similarly, C_{11}^x is shown in Figure 4.2. The FPGA fabric is scanned from top to bottom in the same manner it was scanned from left to right. Here, the maximum number of contiguous blocks, C_{11}^y, equals 4. For this example, the value of $m - C_{11}^y$ equals 16.

The ideal module placement will have minimal cost based on the constraints given. The cost is used to assist in identifying the best placement of one or more modules. The maximum cost $Cost_{T_{MAX}}$, is $2m^2$ and the minimum cost, $Cost_{T_{MIN}}$ is 0. A new cost function can be easily incorporated into DREAM by simply modifying the function computeCost() found in the DREAM source code.

As an example, the cost ($Cost_T$) of the placement shown in Figure 4.3 is calculated.

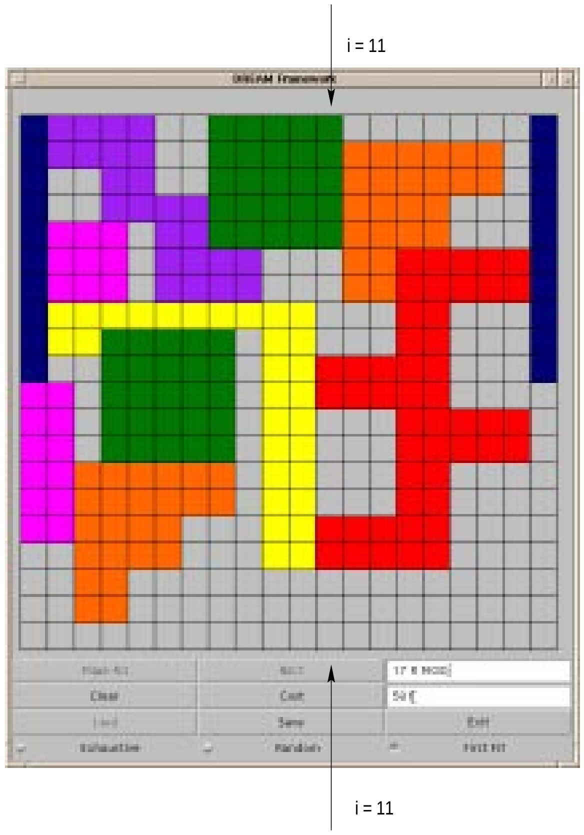


Figure 4.2: Cost Function: Evaluating the Cost in the Y-Direction (m = 20)

The framework first calculates the maximum number of contiguous CLBs in each row of the grid. The value returned for each row is accumulated to find the complete cost in the x-direction. For the example $Cost_x = 290$. The process is then repeated for each column of the grid. For this example, the cost in the y-direction is 304. The sum of both the cost in the x-direction and the cost in the y-direction results in the total cost ($Cost_T = 290 + 304 = 594$).

The breakdown of the cost function calculation for Figure 4.3 is shown in Table 4.1. For each indexed row or column (i) the number of contiguous cells is shown in table column C_i^x and C_i^y. The values of $m - C_i^x$ and $m - C_i^y$ are also shown, along with the cumulative total cost for the current row/column. The final row of the table shows the cumulative cost for each direction (x and y), as well as the total cost of the given placement.

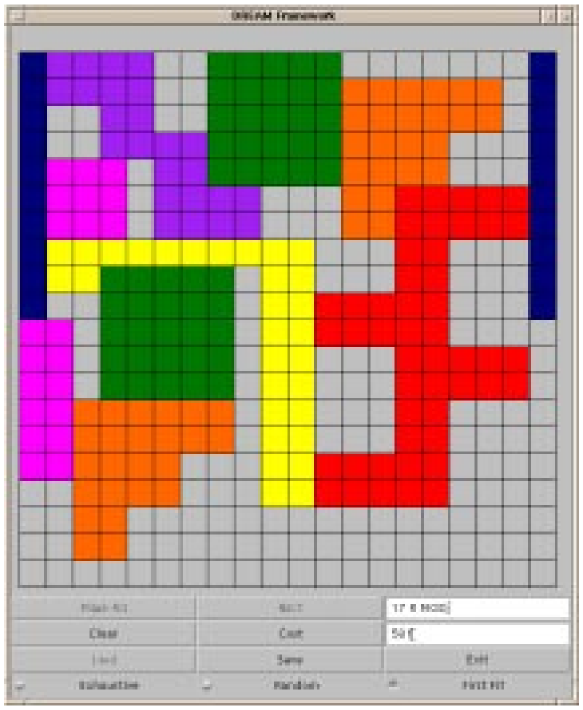


Figure 4.3: An Example Placement

During a typical DREAM execution, if the cost button on the feature panel is depressed, the cost is calculated at that precise moment. Throughout execution, cost is calculated as required for the device utilization value in the statistical report. Device utilization is the ratio of the total area used by the RHM modules to the FPGA device area. This parameter gives a measurement of the percentage of actual FPGA fabric used at different stages during execution. If the cost button is not depressed by the user, the final cost is shown as is the final placement of all insertions/deletion from the data request file.

4.2 Placement Algorithms Used in DREAM

A placement algorithm is a set of decisive steps that determine the location of a module based on a specific heuristic. In each algorithm, a module is submitted for placement. The placement algorithm produces several tentative locations for placing

i	C_i^x	$m - C_i^x$	C_i^y	$m - C_i^y$	C_T
1	7	13	4	16	29
2	2	18	4	16	34
3	2	18	4	16	34
4	3	17	1	19	36
5	3	17	3	17	34
6	3	17	3	17	34
7	3	17	5	15	32
8	3	17	5	15	32
9	3	17	12	8	25
10	3	17	3	17	34
11	4	16	3	17	33
12	3	17	4	16	33
13	3	17	4	16	33
14	4	16	4	16	32
15	4	16	3	17	33
16	4	16	3	17	33
17	4	16	7	13	29
18	16	4	7	13	17
19	16	4	7	13	17
20	20	0	10	10	10
Total	110	290	96	304	594

Table 4.1: An Example: Illustration of Cost Function Calculation

the module. Tentative placement of the RHM in each of k finite placement attempts is evaluated using the cost function. The algorithm then returns the placement of the module resulting in the minimal cost from k (default = 50) placement attempts. The module is placed in the location with minimal cost.

Three placement algorithms are presented to demonstrate the flexibility and functionality of the DREAM framework. The algorithms are versions of commonly used algorithms [8], [19] . A random placement algorithm was tested, in addition to a first-fit algorithm and a modified version of best-fit (exhaustive search). Although the individual steps of each algorithm vary, certain portions remain constant. In all placement algorithms, modules may not overlap. Hence, the availability of a location on the FPGA fabric is based on both the (x,y) location where the origin of the module is placed and the size and shape of the unoccupied space. Modules may not extend outside the boundaries of the FPGA fabric. Additionally, the execution of the DREAM ceases once all insertions/deletions have been processed. For each algorithm, a number of tentative placements are evaluated and the placement with minimal cost is returned.

4.2.1 The Random Placement Algorithm

The first placement algorithm evaluated in DREAM is random placement. This simple algorithm was created to demonstrate the functionality of the DREAM framework. The pseudo-code for the random placement algorithm is shown below. The random placement generates several random locations and places the module at the location with minimal cost. For this algorithm, k (default = 50) is the maximum number of tentative random placements attempted for each module. Placement of the current module is denied when all k tentative placements fail.

```
1. Get module from data request file
2. Generate next tentative (x,y) location
3. If (x,y) is not available goto 2
4. Tentatively place module at tentative location and compute cost
at that location
5. If cost at current location is minimal, save location and its associated
cost as minimal cost
6. If additional tentative locations are available go to 2
7. If no more locations to compute cost, place module at (x,y) location
with minimal cost
8. If more modules, goto 1
```

The flow of the random placement algorithm is shown in Figure 4.4. With each RHM insertion request, k (x,y) locations are randomly generated. For each location, if the generated location is available, the cost of tentatively placing the RHM there is computed. With the cost computed at each available location, the RHM is then placed at the location with minimal cost. If all tentative placements are unsuccessful or there is no available location, placement of the RHM is denied, the RHM is discarded and the next line in the data request file is processed.

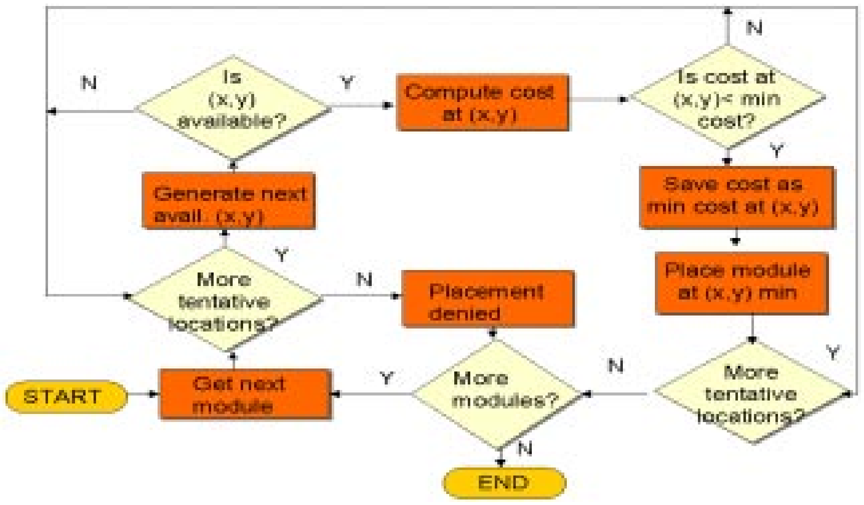
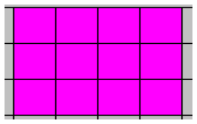
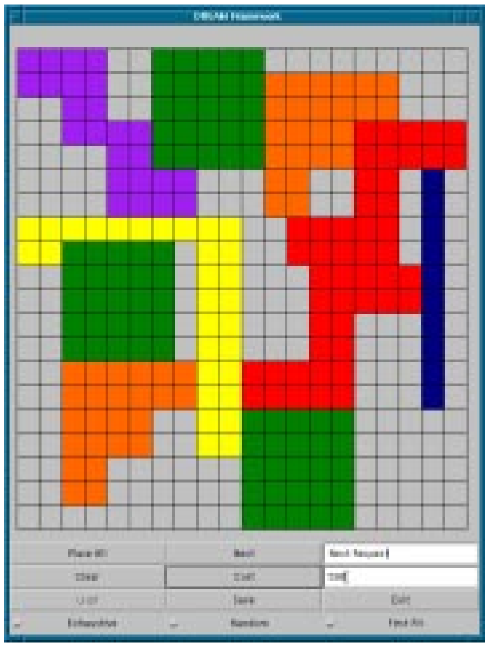

N
Is (x,y) available?
Y
Compute cost at (x,y)
Is cost at (x,y)< min cost?
N
Y
Generate next avail. (x,y)
Save cost as min cost at (x,y)
Y
More tentative locations?
N
Placement denied
Place module at (x,y) min
START
Get next module
Y
More modules?
N
More tentative locations?
Y
N
END

Figure 4.4: Algorithm: Random Placement

Figure 4.5a shows a candidate module that is to be placed in the partially filled FPGA fabric in Figure 4.5b. The cost of the existing placement is 598. The candidate module is placed using the random placement algorithm. The placement resulting from the algorithm is shown in Figure 4.6. The placement of the candidate module of Figure 4.5a in the FPGA fabric of 4.5b using the random placement algorithm is shown in Figure 4.6. The module is placed at location (4,17) on the fabric of size 20. This was the minimum cost for 50 tentative placements. The total cost, $Cost_T$, is 608, with $Cost_x = 324$ and $Cost_y = 284$.





(a) (b)

Figure 4.5: A Candidate Module for Placement

4.2.2 The Exhaustive Search Algorithm

The most extensive and complex of the three placement algorithms tested in the DREAM framework is exhaustive search. Exhaustive search is a version of best-fit modified for two-dimensional hardware devices [19]. Shown in Figure 4.7 and presented in the pseudo-code below, the exhaustive search algorithm is an adaptation of the traditional best-fit placement algorithm for a finite space.

```
1. Get module from data request file
2. If no locations available, deny placement and go to 1
3. Tentatively place module at next available location and compute cost
at that location
4. If cost at current location is minimal, save location and its associated
cost as minimal cost
5. If additional locations are available go to 3
```

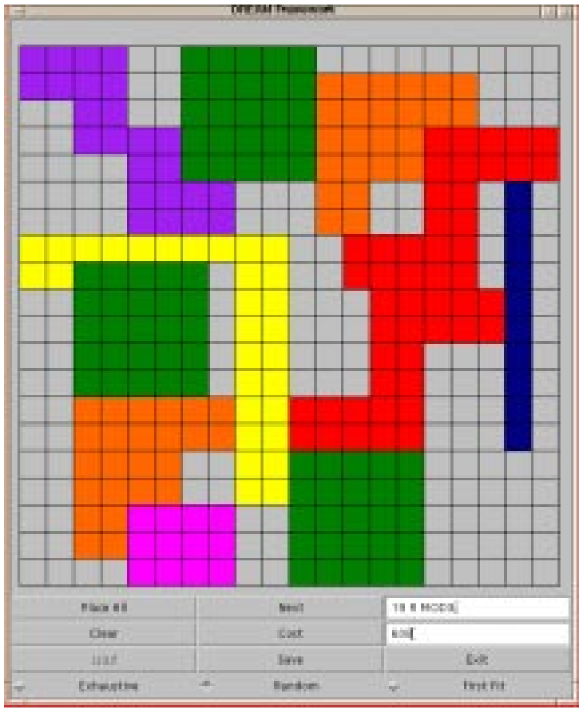


Figure 4.6: Final Placement of the Candidate Module using Random Placement Algorithm

```
6. If no more locations to compute cost, place module at (x,y) location
with minimal cost
7. If more modules, goto 1
```

Exhaustive search is named so because this algorithm checks each unoccupied location on the FPGA fabric and computes the cost of placing the current module at that location. Once a minimum location is determined, the module is placed at the location with minimal cost (x_{MIN}, y_{MIN}). If no such location can be found, request for placement of the current module is denied and the next request in the file is processed.

The placement of the candidate module of Figure 4.5a in the FPGA fabric of Figure 4.5b using exhaustive search is shown in Figure 4.8. The minimal cost returned was found at location (0,4). For graphs of smaller sizes, the first fit and exhaustive search algorithms will frequently return the same placement location. Though the

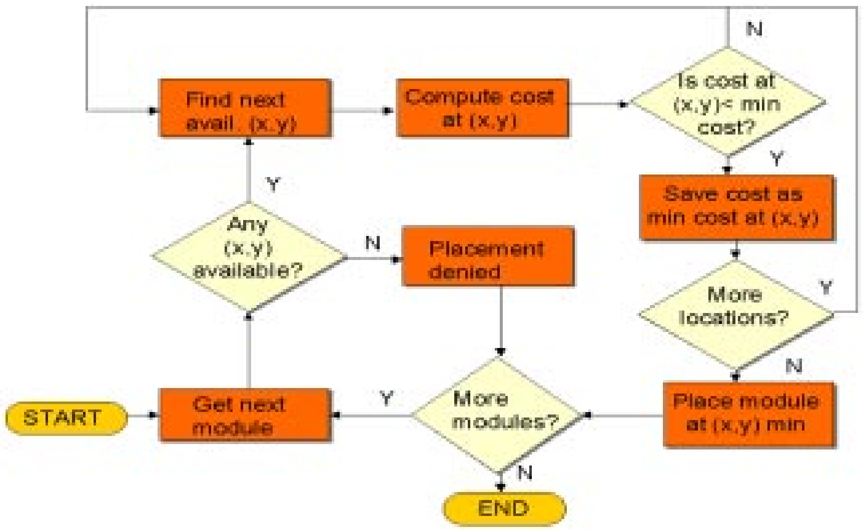


Figure 4.7: Algorithm: Exhaustive Search

final placement of the candidate module is different, the total cost of the placement is 608.

4.2.3 The First-Fit Algorithm

The placement algorithm shown in Figure 4.9 is called the first-fit algorithm. The first-fit algorithm used in this research is a modification of other first-fit algorithms in a two-dimensional space [8]. It is fairly simple to implement and executes relatively quickly. When an RHM request is processed, the FPGA fabric is scanned row by row beginning at the origin. The RHM is tentatively placed in the first k available locations (default $k = 50$). The cost of each tentative location is computed and the RHM is placed at the location with minimal cost. If no location can be found in which to place the RHM, placement of the current module is denied and the next line of the data file is processed.

The pseudo-code below outlines the process of the first-fit placement algorithm. If we set the number of tentative locations, k, to one, we have the traditional first-fit algorithm. This first-fit algorithm should give results that approach exhaustive search

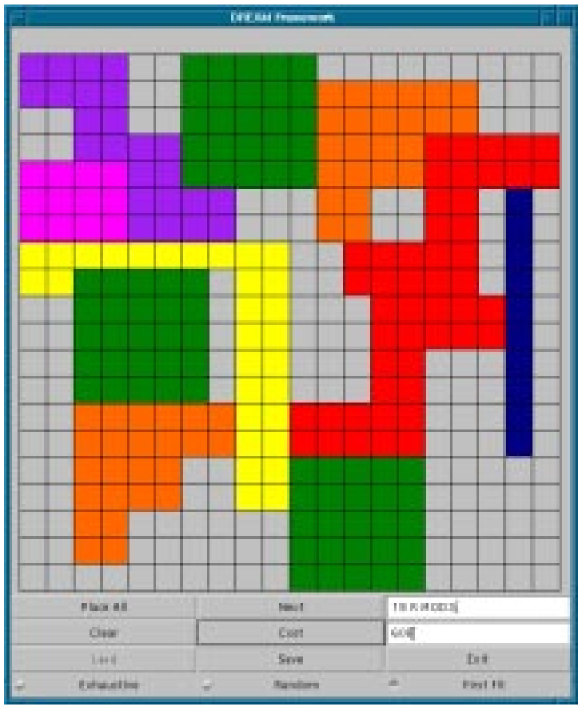


Figure 4.8: Final Placement of the Candidate Module using Exhaustive Search Placement Algorithm

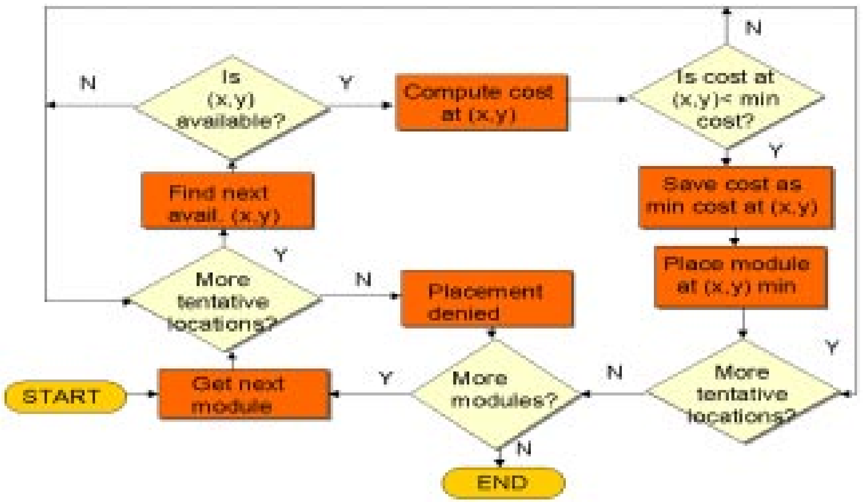

N
Is (x,y) available?
Y
Compute cost at (x,y)
Is cost at (x,y)< min cost?
N
Y
Save cost as min cost at (x,y)
Find next avail. (x,y)
Place module at (x,y) min
Y
More tentative locations?
N
Placement denied
Y
START
Get next module
Y
More modules?
N
More tentative locations?
N
END

Figure 4.9: Algorithm: First-Fit

for large k, while requiring less execution time.

```
1. Get module from data request file

2. Find next available location
```

```
2. If no locations available, deny placement and go to 1
3. Tentatively place module at next available location and compute cost
at that location
4. If cost at current location is minimal, save location and its associated
cost as minimal cost
5. If additional locations are available go to 2
6. If no more locations to compute cost, place module at (x,y) location
with minimal cost
7. If more modules, goto 1
```

In a fashion similar to the placement of the candidate module on the FPGA fabric using the random placement algorithm, the candidate module is placed using first fit. The resulting placement is shown in Figure 4.10. With first fit, the candidate module is placed at location (0, 4). Although the algorithm is different, first fit also results in a total cost of 608.

4.3 Incorporating a New Placement Algorithm into DREAM

Adding a new placement algorithm to the DREAM framework is fairly simple. The Java source code must be modified by a person that is familiar with the Java programming language. The way in which the algorithm determines a location for each module placement is the most challenging of tasks to understand. The code for each modification can be created by effectively following the format of the existing code.

A checkbox for choosing the new algorithm must first be added to the graphical interface through the main program (Dream.java). With this button, the user must also create an event for choosing the algorithm as shown in the following code segment.

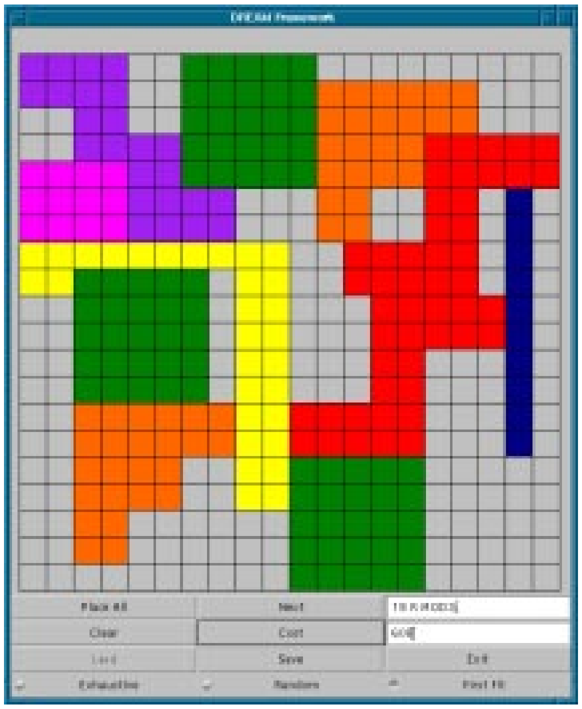


Figure 4.10: Final Placement of the Candidate Module using First Fit Placement Algorithm

```
if (evt.target.equals(exhaust))

        {

                algFlag = 1;

        }

        else if (evt.target.equals(random))

        {

                algFlag = 2;

        }

        else if (evt.target.equals(firstFit))

        {

                algFlag = 3;

}

        else if (evt.target.equals(newAlg))
        {
                algFlag = 4;
}
```

Next, modifications to the class of placement algorithms (PlacementAlg.java) are made to incorporate the new algorithm. Each time a module requests insertion, PlacementAlg is called. A case statement exists for each algorithm. A new case is added for the new algorithm as shown in the following code segment.

```
switch(algFlag)
{
  case 1:
          placeFlag = exhaustiveSearch(device, modToPlace);
          break;
  case 2:
          placeFlag = random(device, modToPlace);
          break;
  case 3:
          placeFlag = firstFit(device, modToPlace);
          break;
  case 4:
          placeFlag = newAlg(device, modToPlace);
          break;
}
```

Finally, the steps for executing the algorithm are added to PlacementAlg.java. In each case, an FPGA fabric with an existing placement of modules and a candidate module to be placed are passed to the algorithm. The steps of the algorithm, determined by the author of the new placement algorithm, are then executed to place the module. As an example, the code for the exhaustive search algorithm is shown below:

```
  public boolean exhaustiveSearch(HardwareResource device, Module thisMod)
  {
        boolean checkit;
        int i,j;
        int placement = (2*device.blocksPerRow*device.blocksPerRow)+1;
        int minCost = (2*device.blocksPerRow*device.blocksPerRow)+1;
        int minRowLocation = -1;
        int minColLocation = -1;

        for(i=0;i<device.blocksPerRow;i++)
        {
          thisMod.x_location = i;
          for(j=0;j<device.blocksPerRow; j++)
          {
            thisMod.y_location = j;
            checkit = device.checkLocation(thisMod,
thisMod.x_location, thisMod.y_location);
            if(checkit == true)
            {
              device.drawModule(thisMod);
              placement = device.computeCost();
              if(placement<minCost)
              {
                    minCost = placement;
                    minRowLocation = thisMod.x_location;
```

```
         minColLocation = thisMod.y_location;
      }
      device.clearModule(thisMod);
     }
    }
   }
   thisMod.x_location = minRowLocation;
   thisMod.y_location = minColLocation;
   if((minRowLocation == -1)||(minColLocation == -1))
    return(false);
   return(true);
}
```

Chapter 5

Experimental Results

5.1 Experimental Setup and Constraints

Based on the cost function defined in Chapter 4, each placement algorithm was used to place RHMs corresponding to user requests on the FPGA fabric. Two types of RHMs, rectangular and non-rectangular, were used for the experiments. An FPGA logic core generator was used to produce the rectangular RHMs. Non-rectangular asymmetric profiles were generated using typical FPGA placement and routing software [20] for a small set of benchmark circuits. These circuits included a floating point adder, a register, a divider and a multiplier. DREAM was developed using version 1.1.8 of the Java Development Kit (JDK).

Several scenarios consisting of 512 insertions/deletions of both rectangular and non-rectangular RHMs were used as test data sets for placement. For our experiments, we used randomly generated data request files containing 512 requests and corresponding deletions for 512 users. To create the data request file, 512 unique user IDs were generated, each with a randomly selected module from a library containing 8 possible modules. These 1024 requests, each one line of the file, were scrambled 512 times and the first 512 requests were stored in the data request file, “requests.dat”. A total of 10 request files were used for each placement algorithm for a comparative analysis.

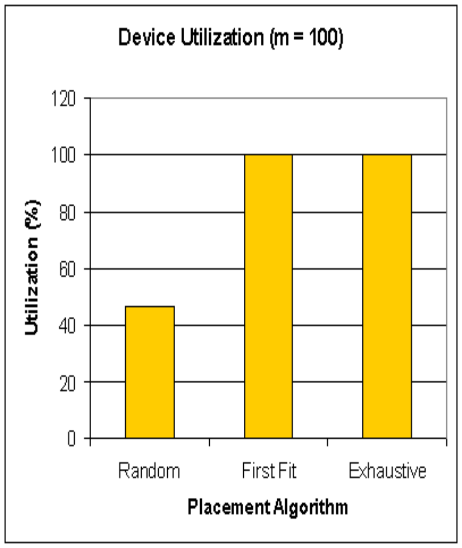
The framework was executed in both the MUSD and MUMD modes. An FPGA fabric of size 100 (default mode) was used. In MUSD mode, all 100 x 100 CLBs were available for module placement. The fabric was divided into four (4) 50 x 50 devices for execution in MUMD mode. The experiments were executed on a 699 MHz Pentium^{TM} running Windows 98 operating system.

5.2 Experimental Results

This section of the document contains the experimental results, graphs and tables for each placement algorithm incorporated into DREAM. A small, symmetrical module was used to demonstrate the functionality of the DREAM framework and the cost function upon which quality of placement is based. Each algorithm was executed using 10 unique data request files. The same ten files were used for each algorithm in order to provide data for the comparative analysis.

Figure 5.1 shows a comparison between the device utilization and module capacity for each algorithm in MUSD mode. A 5 x 5 square module was placed as many times as possible in an FPGA fabric of size $m = 100$. Figure 5.1a is the device utilization of each algorithm in the fabric. The first fit and exhaustive search algorithms have a device utilization of 100% compared too a less than 85% value for the random placement algorithm. The total cost of placements that achieve 100% device utilization are 20,000, by definition of the cost in Equations 4.1 - 4.3. This graph demonstrates how the nature of the placement algorithm can vary the device utilization for a symmetrical RHM.

Figure 5.1b is a comparison of the module capacity of each placement algorithm, also in MUSD mode. Due to the random generation of locations by the random placement algorithm, only 187, 5 x 5 modules were successfully placed in the fabric. The first fit and exhaustive search algorithms returned successful placement of 400



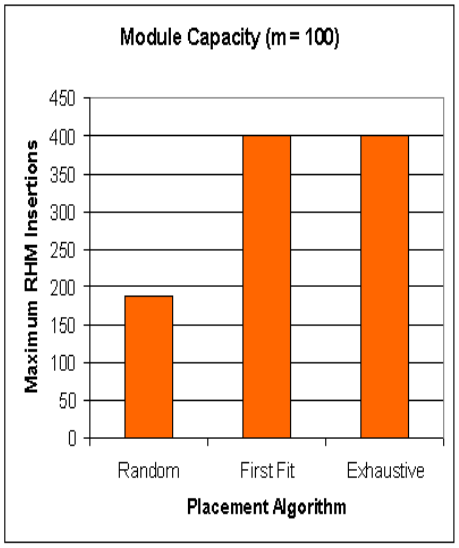


Figure 5.1: Device Utilization and Number of Insertions for a 5 x 5 Module

modules, which occupy the entire FPGA fabric. Though each algorithm returns the location with minimal cost of a set of tentative placements, the random placement does not have a consistent method of generating the best tentative placement of the RHMs.

5.2.1 Total Execution Time

The time required to process an entire request file is called the total execution time. The total execution times for each placement algorithm are shown in this section. A total of 10 runs were made, each with a unique user request file.

Figure 5.2 shows the total execution time of the random placement algorithm in both MUSD and MUMD user modes. The time required to process the entire request data file is slightly less in the second mode, as is expected. Due to the inability to partition application profiles across multiple boards in MUMD mode, the amount of contiguous space is slightly decreased, resulting in less available area and decreased placement time.

The total execution time for the first fit algorithm is shown in Figure 5.3. The

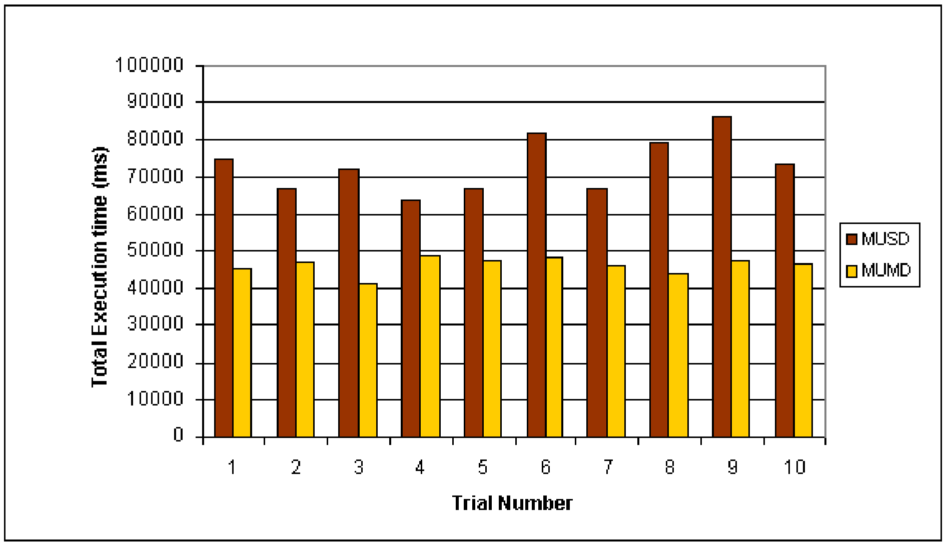


Figure 5.2: Random Placement: Total Execution Time

execution time in MUSD mode is on average 120000 ms. Execution time in MUMD mode is also consistent at approximately 85000 ms. Again, the amount of time used in searching for a placement location is reduced by the division of the FPGA fabric into four separate devices.

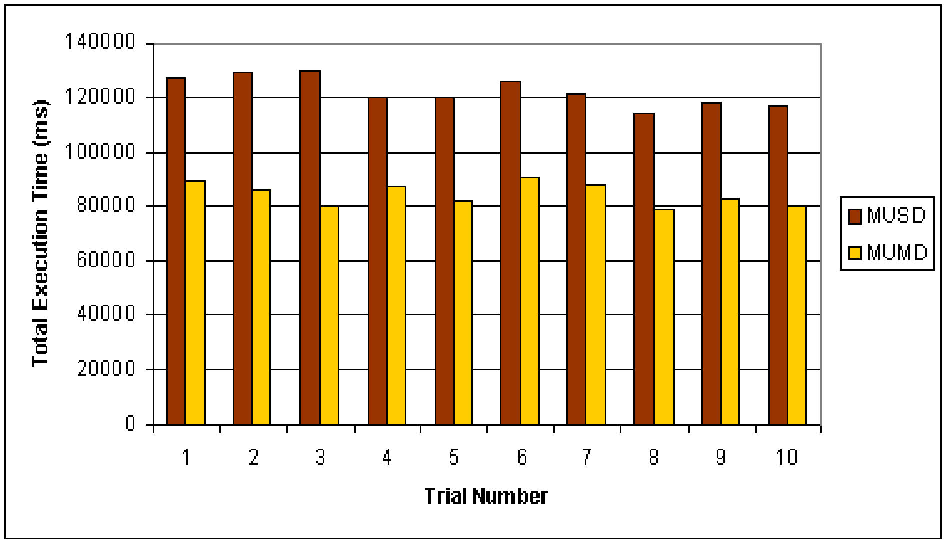


Figure 5.3: First Fit: Total Execution Time

The total execution time for the exhaustive search algorithm is shown in Figure 5.4. The execution time in both user modes is the highest of the algorithms tested. The extensive nature of the algorithm requires a great deal of time spent on determining the optimal location for placement. However, the execution time is significantly higher in MUSD mode than MUMD.

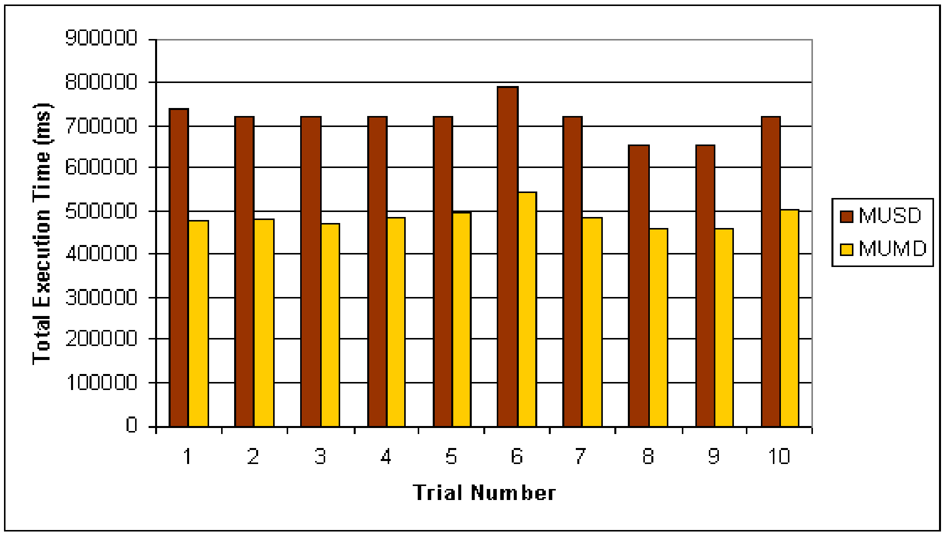


Figure 5.4: Exhaustive Search: Total Execution Time

Here, the total execution times for each placement algorithm have been shown. Random placement required the least amount of execution time. First fit and exhaustive search required the largest amount of time to process the data files. This is to be expected and is due to the differing natures of each algorithm.

5.2.2 Device Utilization

The device utilization is define as the ratio of the total area used by the RHM modules to the total FPGA device area. It is a metric used to determine how well a placement algorithm makes use of the available FPGA device area. The graphs below show the device utilization in both MUSD and MUMD modes for each algorithm.

Figure 5.5 shows the device utilization for the random placement algorithm in both MUSD and MUMD user modes. The MUMD mode results in a higher device utilization than MUSD mode.

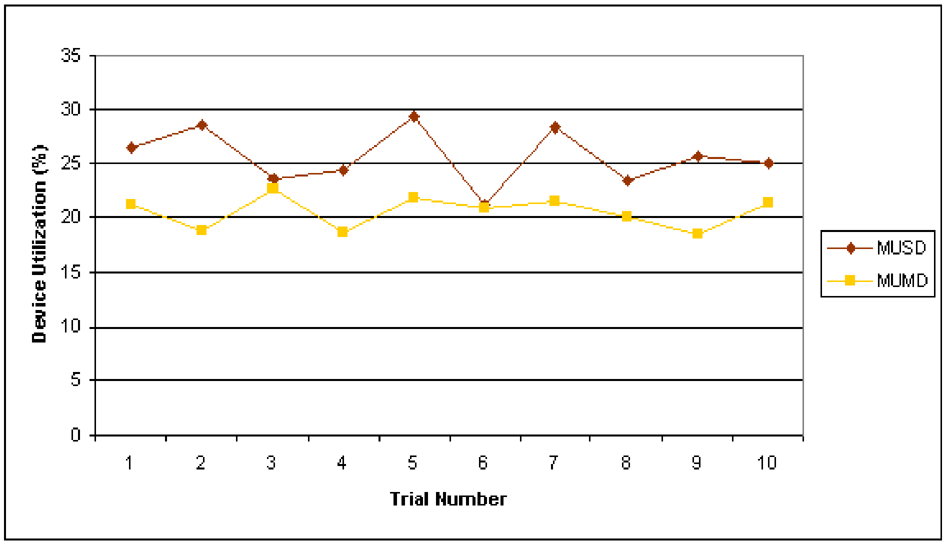


Figure 5.5: Random Placement: Device Utilization

The first fit algorithm exhibits similar device utilization in both user modes. Shown in Figure 5.6, the device utilization for the first fit algorithm is approximately 90% in MUSD and MUMD user modes.

The exhaustive search algorithm device utilization is fairly similar in both user modes. In Figure 5.7, the MUMD mode demonstrates a more consistent device utilization at approximately 60%. The device utilization percentages of the MUSD mode vary from 62 - 66%.

The device utilization of each algorithm in MUSD is shown in Figure 5.8. The exhaustive search shows the highest device utilization of the three algorithms. All three algorithms are consistent in exhibition of device utilization. Random placement consistently uses the least amount of available FPGA fabric for placement of RHMs.

The device utilization for each of three placement algorithms, in both user modes

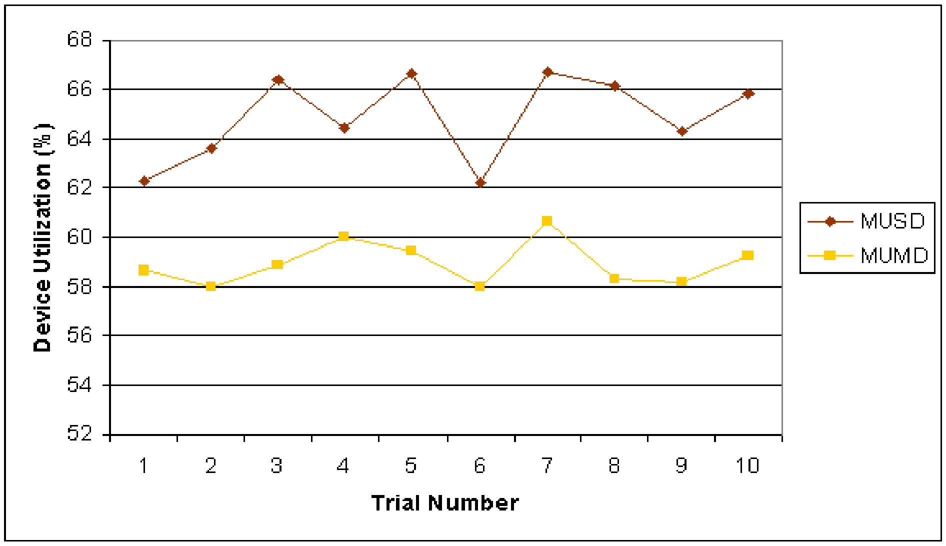


Figure 5.6: First Fit: Device Utilization

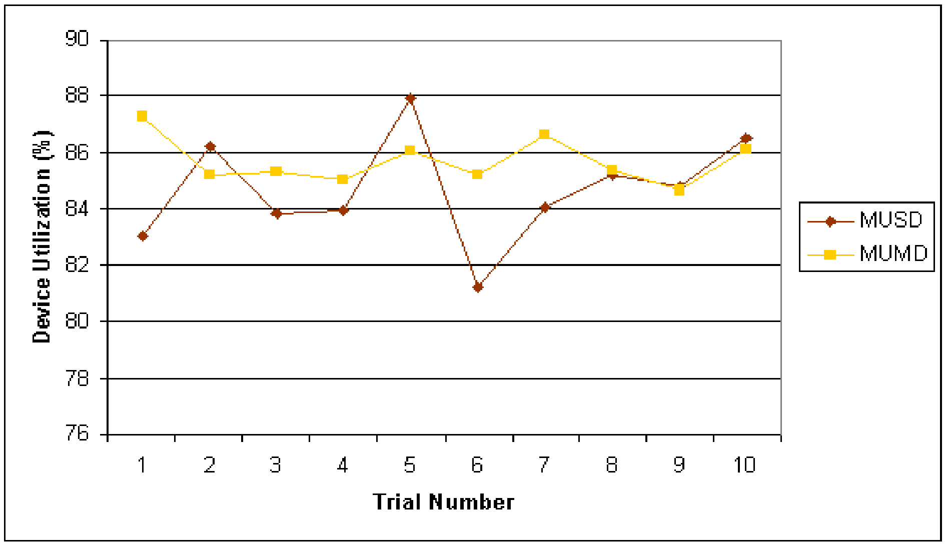


Figure 5.7: Exhaustive Search: Device Utilization

were provided. The random placement algorithm proved to be the least efficient method of placing RHMs. Exhaustive search averaged the highest device utilization, with values on average above 85%.

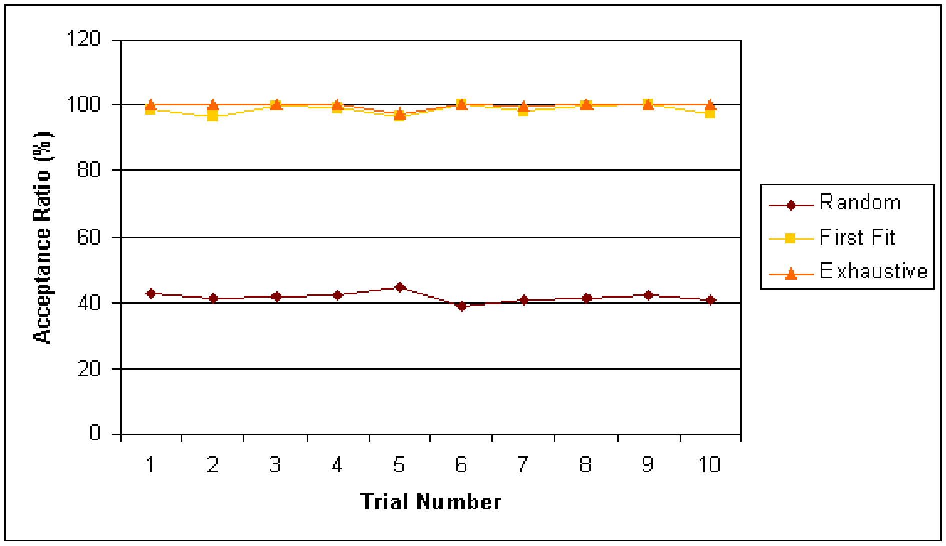


Figure 5.8: Device Utilization in MUSD Mode

5.2.3 User Request Acceptance Ratio

The number of RHM requests in the data file successfully processed and placed are monitored using a parameter called the *request acceptance ratio*. This is a ratio of the number of RHMs successfully placed to the number of actual RHM requests made in the data file. The values are shown in both modes for each placement algorithm.

Generally, the acceptance ratio for the random placement algorithm is higher in MUSD mode than in MUMD mode, as shown in Figure 5.9. The acceptance ratio for the majority of the data sets in MUSD is approximately 42%. MUMD mode has a slightly lower percent acceptance with values ranging from 33% to 41%. In general, no more than half of the requested FPGA fabric was granted with the random placement algorithm.

The MUSD mode of the first fit algorithm on average accepts more data requests than MUSD mode, as is shown in Figure 5.10. In comparison to MUSD, which reaches 100%, the MUMD mode of first fit is above 94%. With less than 10% of requests denied, the first fit algorithm performs very well.

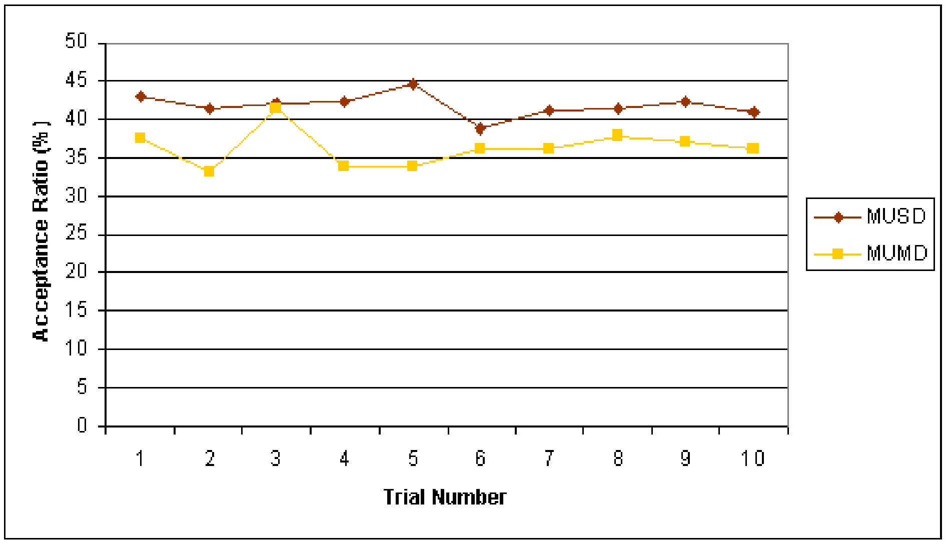


Figure 5.9: Random Placement: Acceptance Ratio

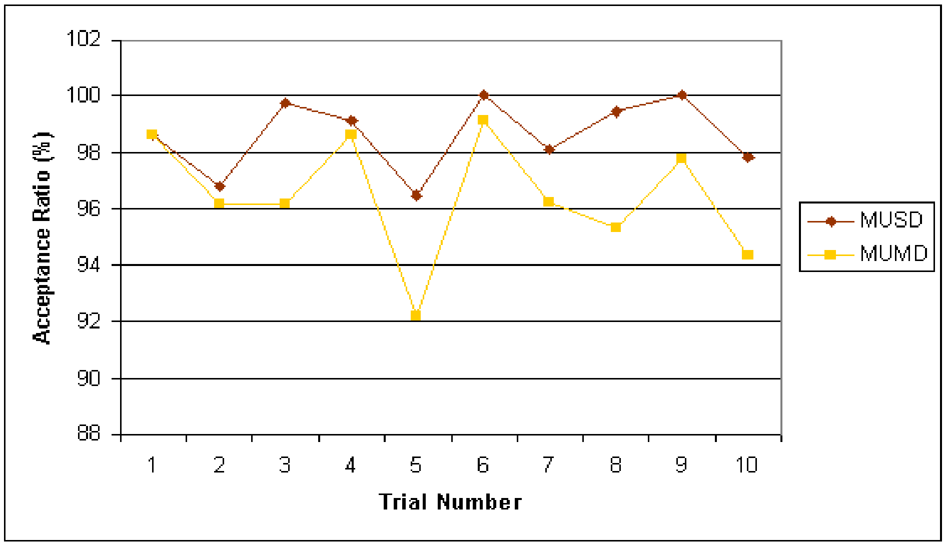


Figure 5.10: First Fit: Acceptance Ratio

The exhaustive search algorithm exhibits the best acceptance of data requests in MUSD mode, shown in Figure 5.11. It has a consistent acceptance ratio of approximately 100%. Though above 90%, the acceptance of data requests in MUMD mode does not prove as consistently high as MUSD mode.

The percent acceptance of each placement algorithm executed in MUSD mode is

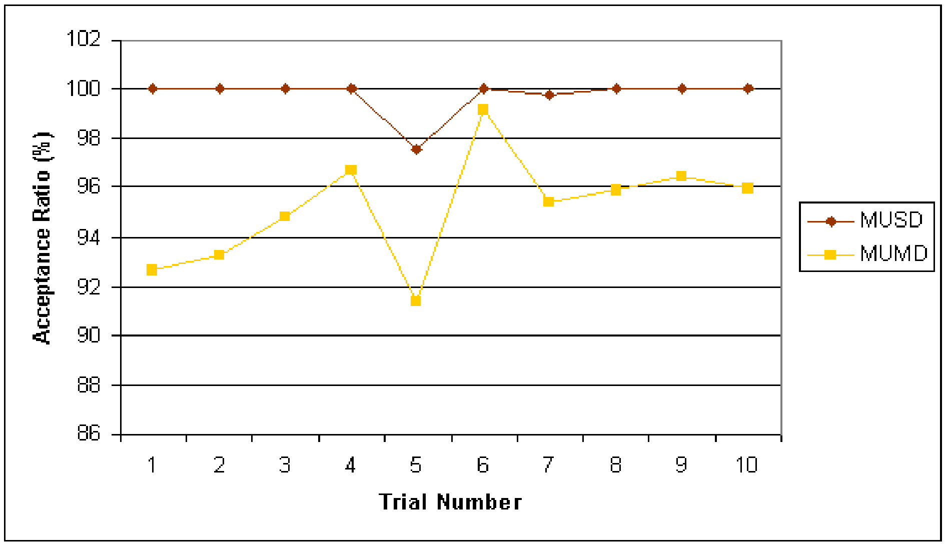


Figure 5.11: Exhaustive Search: Acceptance Ratio

shown in Figure 5.12. Exhaustive search proves to accept the largest percentage of the data request file, the entire file in most cases. The first fit and random placement acceptance ratios are also shown.

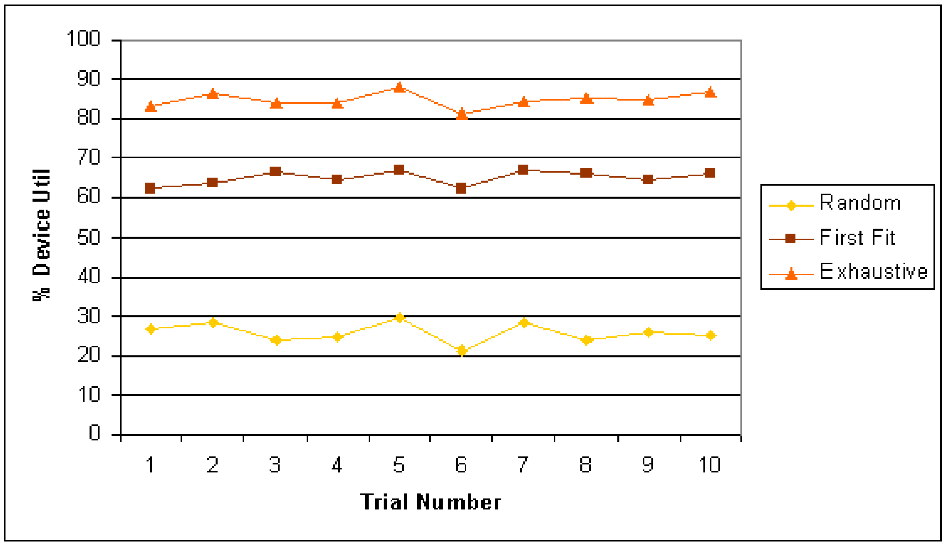


Figure 5.12: Acceptance Ratio in MUSD Mode

A compilation of the average performance statistics is listed in Table 5.1. The values shown are averaged over ten data sets and reported here. Each placement algorithm was executed in both MUSD and MUMD modes. The default values for the framework were exercised. The information provided by the DREAM statistical report is for the user to identify the best placement algorithm based on the specific needs of each user.

Algorithm	User Mode	Execution Time (ms)	Percent Acceptance	Device Utilization
Random	MUSD	73186	41.762	25.599
	MUMD	46112	36.373	20.539
First Fit	MUSD	122292	98.619	64.86
	MUMD	84468	96.467	58.937
Exhaustive Search	MUSD	715878.6	99.72981	84.67649
	MUMD	486570.6	95.159	85.66842

Table 5.1: Placement Algorithm Average Performance Statistics

Chapter 6

Conclusions and Future Work

The DREAM framework provides a tool to evaluate the quality of placement algorithms for RTR systems. Based on a cost function that measures the amount of unoccupied space in two dimensions, DREAM offers a method for determining the best placement algorithm given a set of user-defined constraints. This evaluation tool creates a user-driven environment.

Previous research on placement algorithms for hardware devices offers methods for *fast* placement on RTR systems [6]; for mapping large circuits to FPGAs [7]; and for minimizing PLD area while reducing compile time [18]. However, these studies do not provide a way to compare or evaluate the performance of the solutions in alternate FCCMs with different performance requirements.

DREAM builds upon research that supports mapping algorithms into defined macros or profiles [5] [7]. DREAM also takes advantage of the ability to potentially reprogram a hardware device an infinite number of times to realize various circuit designs. The flexibility exhibited by programmable logic devices creates an opportunity for profiles of applications to be paged in and out of a device as needed. The data files processed in this framework demonstrate how easily these profiles can be inserted and deleted on a device.

The implementation of the DREAM framework evaluated three placement algorithms in an FPGA fabric of size $m = 100$. A set of data request files with a total of 512 insertions and deletions were processed for each algorithm. The resulting data has shown that once incorporated into the framework, a placement algorithm can place modules of varying size and shape to assess the quality of placement as well as the average usage of the device. The exhaustive search and first-fit algorithms have high device utilization as compared to random, but require significantly more execution time. If high utilization and low execution time are preferred, first-fit would be the appropriate choice.

In conclusion, DREAM has proven to be a useful tool in evaluating the quality of placement of algorithms for two-dimensional hardware devices. The algorithms presented in this thesis were either created or modified for application to an FPGA. The quality is based on a cost function created to quantify the resulting effect of each RHM placement.

The DREAM framework can be easily modified to incorporate other cost functions and placement algorithms. Future work includes adding another user mode to DREAM that simulates an environment where one or more FPGAs exist, each connected to a host processor, and each device is allocated to a single user. The framework should also accommodate larger module libraries and modules that are constrained to pin-to-pin device connections.

Bibliography

[1] D. Eatmon and C. S. Gloster. Evaluating Placement Algorithms for Run-time Reconfigurable Systems. *Military and Aerospace Applications of Programmable Devices and Technologies International Conference*, 1999.

[2] P. M. Athanas and H. F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *Computer*, 26(3), Mar 1993.

[3] J. Oberg A. Jantsch, P. Ellervee and A. Hemani. A Case Study on Hardware/Software Partitioning. *IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[4] W. D. Bishop and W. M. Louks. A Heterogeneous Environment for Hardware/Software Cosimulation. *Proceedings of the IEEE Annual Simulation Symposium*, 1997.

[5] M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. *Proceedings of the SPIE*, 1995.

[6] K. Bazargan and M. Sarrafzadeh. Fast Online Placement for Reconfigurable Computing Systems. *IEEE Symposium on Field Programmable Custom Computing Machines*, 1999.

[7] J. M. Emmert and D. Bhatia. A Methodology for Fast FPGA Floorplanning. *ACM/SIGDA International Symposium on FPGAs*, 1999.

[8] J. O. Berkey and P. Y. Wang. Two-Dimensional Finite Bin-Packing Algorithms. *Journal of the Operational Research Society*, 38(5), 1987.

[9] S. Brown and J. Rose. FPGA and CPLD Architecture: A Tutorial. *IEEE Design and Test of Computers*, 12(2), 1996.

[10] R. C. Seals and G. F. Whapshott. *Programmable Logic PLDs and FPGAs*. McGraw Hill, New York, NY, 1997.

[11] S. Hauck. The Roles of FPGA's in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4), Apr 1998.

[12] N. W. Bergmann and J. C. Mudge. Comparing the Performance of FPGA-Based Custom Computers with General-Purpose Computers for DSP Applicaitons. *IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[13] D. Roncin P. Bertin and J. Vullemin. Introduction to Programmable Active Memories. Technical report, June 1989.

[14] M. Gokhale and R. Minnich. FPGA Computing in Data Parallel C. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.

[15] The XC4000 Data Book. Technical report, Xilinx, Inc., San Jose, California, 1992.

[16] D. Roncin P. Bertin and J. Vullemin. Programmable Active Memories: a Performance Assessment. Technical report, 1993.

[17] P. Casselman. Virtual Computing. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.

[18] Y. Sank and J. Rose. Trading Quality for Compile Time: Ultra-Fast Placement for FPGAs. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1999.

[19] Levi Simoni and David. New Worst-Case Results for the Bin-Packing Problem. *Naval Research Logistics*, 41(4), June 1994.

[20] Xilinx Foundation Series Software. Technical report, Xilinx, Inc., 1997.

Appendix A

DREAM Command Line Options

The DREAM framework responds to several instructions that can be implemented fro m the command line. These command line options allow the user to determine the mode in which the framework operates, the size of the FPGA fabric modeled and manual or automatic placement of the candidate modules.

PARAMETER	COMMAND	ADDITIONAL REQUIRED VALUE
Help Option	-help	
User Mode	-musd	
	-mumd *size*	Individual device size (natural numbers)
Process Mode	-man	Placement Algorithm
	-auto	Placement Algorithm
Fabric Size	-size *size*	Size of fabric (whole numbers)
Tentative Placments	-tent k	Number of tentative placements (natural numbers)

Table A.1: DREAM Command Line Options

Table A.1 lists each of the parameters that can be modified on the ccommand line as well as any additional values that are needed. With the command line options, the user can choose in which mode the DREAM framework will be executed. The user can choose from MUSD, the default mode or MUMD. As explained in the above sections, each mode simulates device usage in different scenarios.

The DREAM framework displays a list of valid commands when the user enters

“-help” on the command line. This list is also displayed when an invalid command is entered. This help options gives the proper commands to be used as well as any additional required value for a command option.

The way in which the data request file is processed can be determined before execution of the DREAM framework begins. The user can also decide whether to manually process the insertions/deletions in the file or to have the entire file processed automatically by using the command options “man” or “auto”. To allow the user to process each line of the request file individually, the “man” command is used. This is the default mode for the DREAM framework. If the request file is to be processed entirely before displaying the final placement, the “auto” command is used. However, at any time during manual processing, the “Place All” button on the feature panel can be pressed to achieve the same results as using the “auto” option.

When exercising the process mode option, the user also must select an algorithm for the framework to use. Unlike the graphical interface where the placement algorithm is chosen before placing any modules, the command line option must choose the algorithm without any type of panel available. The options for the placement algorithm are “exhaust” for exhaustive search, “first” for first -fit, and “rand” for random placement. The random placement algorithm will search a default number of locations for a module. The user can change the default vaule with the “search” parameter on the command line. If no option is chosen on the command line, the framework defaults to the first-fit algorithm. In the following command line, the random algorithm is selected to process the entire data request file. For this run, 75 tentative locations will be attempted for each module.

```
> java Dream -auto rand -tent 75
```

The size of the entire FPGA fabric can be determined by the user of the DREAM framework. By entering “s” followed by the size of the fabric, as a positive integer, the user can alter the size of the fabric from the default fabric size of 100.

The command line options allow the DREAM framework user to modify the execution parameters to fit the evaluation of each specific application. If the options are not exercised for any of the parameters, the default settings are used. The command line options can be entered in any order after the requisite Java commands are are used to begin execution of the software. For example, if the size of the entire FPGA fabric is to be changed to 30, the command line would read:

```
> java Dream -size 30
```

Figure A.1 shows a partially populated FPGA fabric that was created using the commmand line above.

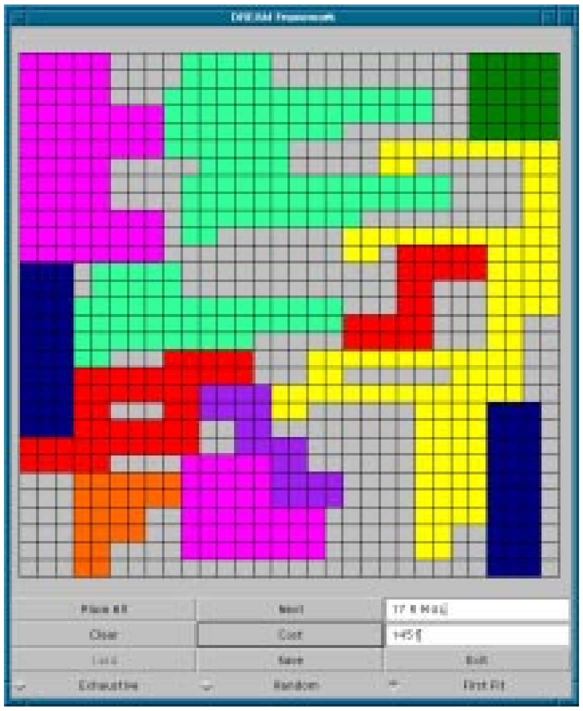


Figure A.1: The DREAM Framework, m = 30

Similarly, if the software is to be executed in MUMD mode with 20 CLBs in each device, the changes to the command line would be as follows:

```
> java Dream -mumd 20
```

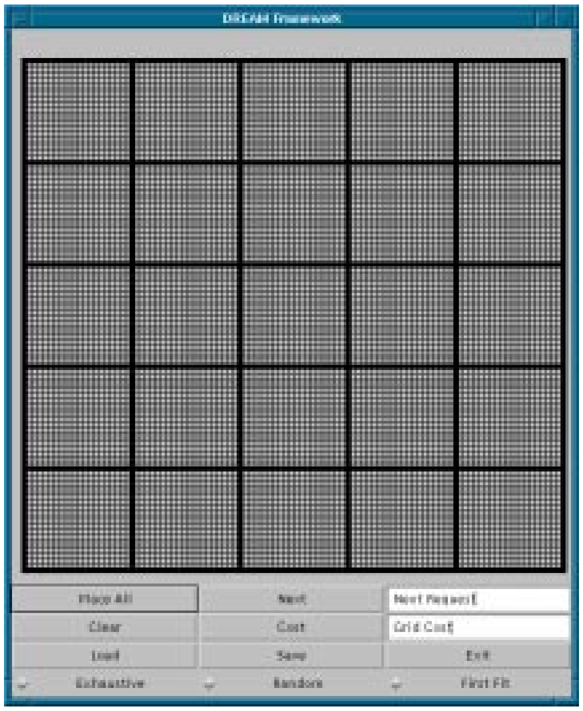


Figure A.2: Sample FPGA Fabric in MUMD mode with Device Size = 20

The default size (m = 100) will have 5 devices, each 20 x 20, on each row. The DREAM interface corresponding to the above command line is shown in Figure A.2.

To combine two or more commands, the user can add them to the end of the command line. To automatically run the software with the random placement algorithm and a fabric size of 75 CLBs, the user would type:

```
> java Dream -auto rand -s 75
```

The command line options can be combined and entered in any order, but must be on the same execution line.

Appendix B

Evaluating Placement Algorithms for Run-Time Reconfigurable Systems