4-1997

Evolutionary Codesign

Karthikeyan Ethirajan

Recommended Citation

Ethirajan, Karthikeyan, "Evolutionary Codesign" (1997). *Master's Theses*. 4855.
https://scholarworks.wmich.edu/masters_theses/4855

EVOLUTIONARY CODESIGN

by

Karthikeyan Ethirajan

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Electrical and Computer Engineering

Western Michigan University
Kalamazoo, Michigan
April 1997

ACKNOWLEDGMENTS

There are many people who helped me to complete this greatest venture of my academic life, a thesis. First, I extend my deepest gratitude and sincere appreciation to my major advisor, Dr. Sharon Hu for her continuous guidance throughout this project. She offered her undiminished support even after having moved to a different University. Second, I wish to express my thanks to the members of the thesis committee for their many reviews and valuable suggestions. I am thankful to Dr. Garrison Greenwood for letting me audit his course on Evolutionary Computation. I also highly appreciate Praveen Jayamohan for having spent many hours teaching me the fundamentals of Software Engineering. I am also very grateful to all my family for being so patient with me and for their encouragement. Finally, I want to thank my friends who have helped me in several ways from giving me access to computers to buying refreshments for my thesis defense!

Karthikeyan Ethirajan

EVOLUTIONARY CODESIGN

Karthikeyan Ethirajan, M.S.E.E

Western Michigan University, 1997

We present our approach to hardware/software partitioning for embedded systems based upon Evolutionary Algorithms. We have implemented it in a CAD tool, *EvoC - Evolutionary Codesign* which does automatic hardware/software partitioning of real-time embedded systems at the system level. Our objective is to find good design configurations that are tuned towards user's preferences. We are able to explore a large, often intractable design space using Evolutionary Algorithms while evaluating solutions having multiple and sometimes conflicting attributes in the light of Multi-Attribute Utility Theory. *EvoC* provides a generic format for specifying a wide variety of design problems and the implementation assumes no target architecture. A multiple bus and shared memory communication scheme has been incorporated into *EvoC* which analyzes the behavior and produces connected systems. Two design examples are given to illustrate the capability of our tool. Two of the factors which caused the high execution time of *EvoC* were identified and the appropriate corrective measures taken are discussed.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER

I. INTRODUCTION 1

II. RELATED RESEARCH 5

III. OVERVIEW OF *EvoC* 8

- Formulation of the Partitioning Problem 9
- A Generalized Evolutionary Algorithm 12
- Imprecisely Specified Multi-Attribute Utility Theory 13
- Integrating ISMAUT With EA 14
- Handling Timing Constraints in RTES 17

IV. SOFTWARE DEVELOPMENT OF *EvoC* 19

- Fundamental Concepts Underlying the Design of *EvoC* Data Structure 19
- Description of *EvoC* Data Structure 22
- User Input Files for *EvoC* 27
- Partitioning 28
- Ranking and Selection 31

CHAPTER

Deterministic Selection of Parents 32

Mutation 35

V. MODELING OF COMMUNICATION 37

Multiple Bus and Shared Memory Communication Model................. 37

Incorporating MBSM Model Into *EvoC* 39

Mutation With MBSM Communication Model................................. 42

Rationale Behind Multiple Bus Selection 44

VI. DESIGN EXAMPLES AND EXECUTION TIME ANALYSIS 46

Example I 46

Example II & III... 51

Timing Analysis.. 58

VII. CONCLUSIONS AND FURTHER WORK 61

APPENDICES

A. Input File Format of *EvoC* 64

B. Data Structure of *EvoC* 67

BIBLIOGRAPHY 70

LIST OF TABLES

1. Set of FPs With (*a, d, p*) in Microseconds and Their Components List 47

2. Component Libraries and Their Characteristics 48

3. Typical Solutions Found by *EvoC* When Lower Cost is Preferred 50

4. Typical Solutions Found by *EvoC* When Higher Feasibility Factor is Preferred 51

5. Set of FPs With (*a, d, p*) in Microseconds and Their Components List 52

6. Set of CPs 53

7. Component Libraries and Their Characteristics 54

8. Exhaustive List of Feasible Solutions for Example III 55

9. Typical Solutions Found by *EvoC* When Lower Cost is Preferred 56

10. Typical Solutions Found by *EvoC* When Higher Power is Preferred 56

11. *EvoC* Run Time for the Examples 59

12. Improved Execution Time of *EvoC* 60

LIST OF FIGURES

1. System Level Hardware/Software Partitioning Approach 8

2. The *EvoC* System 16

3. Mapping for System Implementation 19

4. Hierarchy of *EvoC* Data Structure 20

5. Data Flow Graph Based on FPs and CPs 22

6. *EvoC* Data Structure 24

7. Mapping Between `Func_Imp[]` & `Individual[]` Arrays 25

8. Partitioning During Generation of Initial Population 29

9. An Example System Configuration 38

10. Types of Component Modifications Due to Mutation 43

CHAPTER I

INTRODUCTION

Recent advances in the Computer-Aided Design (CAD) field have raised great interest in the hardware/software codesign of embedded systems [23]. Many aspects of hardware-software codesign are being actively studied, such as co-specification, co-synthesis, hardware-software partitioning. The objective of hardware/software codesign is to produce computer systems that have a balance of hardware and software components which work together to satisfy system specification. This balance between hardware and software implementations is referred to as the *partitioning problem*. Developing efficient means of performing hardware/software partitioning is key to the automatic design of complex computer systems.

We are particularly interested in hardware-software partitioning for real-time embedded systems (RTES). Such systems can be found in many applications, such as powertrain control of automobiles, navigation and landing control of aircraft, and networks and communications. The design of RTES, which are generally composed of both hardware and software components, is quite challenging. In addition to the usual design criteria for embedded systems, such as reliability, maintainability and cost effectiveness, RTES must provide timely services. That is, the functional behavior of thesie systems must not only logically correct but also temporally correct.

Hardware/software partitioning at the system level allows exploration of hardware architectures as well as hardware-software partitions. Our experience indicates that these combined decisions are the primary factors that define the cost and performance of an embedded system [4]. At the system level, hardware is modeled as resources with no detailed functionality and software is modeled as tasks utilizing the resources. At this high level of abstraction, we are able to evaluate various partitioning alternatives up front, which, in turn, guides lower level design efforts.

It is our belief that system level partitioning of a RTES should address some of the following questions:

1. Which processes should be implemented in dedicated hardware circuits and which should be in software ?
2. What processors and ASICs should be used, and which software tasks should be executed by which processor ?
3. Can an identified system configuration meet all of the temporal requirements ?
4. How to form a system architecture so as to minimize the communication overheads ?
5. Which system configurations best reflect the designer's preferences with respect to various performance measures ?

An automated CAD tool for the hardware/software partitioning of a RTES must have an input format for capturing system specifications such as the desired system functionality and their communication needs. It should also have the scope for specifying system design constraints like the temporal constraints for RTES. This stage is followed by *exploration*, which is the analysis and evaluation of several design configurations (formed from the user input system specifications). The output of the tool should yield good design configurations, which satisfies all the system constraints and completely specifies the system. This is then submitted to the user for his perusal.

The major contributions of this thesis are the following:

1. Development of a input format and a compatible data structure that facilitates implementation of Evolutionary Algorithm in a new version of *EvoC*.

2. Incorporating a multiple bus and shared memory communications model into *EvoC*.

3. Improving the execution speed of *EvoC*.

The remainder of the thesis is organized as follows. Chapter II gives a brief literature survey on the different approaches of the concurrent design methodologies of hardware and software. In Chapter III, some background on the techniques used in *EvoC* are given. Chapter IV elaborates the software developmental aspects of *EvoC*. A description of the communications modeling strategy is presented in Chapter V. Chapter VI contains design examples that were tested on *EvoC* along with some timing analysis. Chapter VII summarizes the salient features of our tool and discusses the scope for future expansion of *EvoC*.

CHAPTER II

RELATED RESEARCH

Many CAD tools that have been developed (COSYMA [11], CODES [3], COSMOS [15] and Genie [24]) so far demonstrate an increased enthusiasm in the field of *hardware-software codesign* [23]. We briefly discuss some approaches taken in the field of hardware/software codesign and thereby explain the need of our tool, *EvoC*.

Most hardware/software codesign techniques perform partitioning at a level of abstraction where the detailed functionality or behavior of a system is specified. These tools [5,6,10,21] contain as their output, a software derivative and a hardware derivative. The software partition is often expressed in a high-level programming language like C or C++. The hardware partition is given in a hardware description language (HDL) like VHDL, Hardware C, *etc.* A fundamental limitation of such "low" level modeling is the complexity associated with exploration of even a relatively small design space.

We feel that a hardware/software partitioning at a higher level than the behavioral level is needed. This allows exploration of tradeoffs in hardware architectures as well as hardware/software partitions. Of course, because detailed implementations are not considered at the higher level, the accuracy of results is of concern. However, the same problem also exists for lower-level partitioning [13]. Furthermore, by applying a *hierarchical refinement design methodology* [25], the higher-level partitioning results are considered as an initial design, which will be refined during lower-level design (*e.g.,* using lower-level co-synthesis tools such as COSYMA or VULCAN II [6,10]).

Some researchers restrict the candidate system implementations. For example, in COSYMA, Ernst, Henkel and Benner assume a target architecture of one general-purpose host processor and one customized coprocessor [6]. Gupta and DeMicheli [10] also use a single, pre-defined processor while allowing the inclusion of a few custom ICs. Usage of some partitioning algorithms [5,6,10] allows examination of some basic block of operations in the system specification to see if the original implementation can be altered so as to meet certain constraints (*e.g.,* timing and bus utilization). These approaches severely limits the possibility of exploring a large design space and hence provides only a local optimization.

Manual or interactive partitioning schemes have also been developed [3, 20]. This again restricts the number of alternate design configurations that can be explored in a reasonable amount of time as compared to an automatic hardware/software partitioning approach.

Kumar, Aylor, Johnson and Wulf [17, 18] have worked at the system level. But the actual partitioning is done at a lower level. The metric that is used to evaluate the design is a linear combination (weighted sum) of system attributes such as cost and execution time. The disadvantages of using a weighted sum approach is discussed in the next chapter.

To minimize customized hardware in microcontrollers, hardware designers are currently developing libraries of standardized peripheral components. This approach allows fast design turnaround time [6]. This supports the idea of maintaining separate component libraries for hardware and software when developing a CAD tool for doing hardware/software partitioning.

Our observation suggests that a low-level, full fledged codesign approach cannot concentrate on the partitioning process alone and explore multiple design configurations, in order to attain a balance between conflicting objectives (*e.g.,* cost and speed). A high-level design approach, on the other hand, is more suitable for the hardware/software partitioning process than any other aspects in the codesign field.

CHAPTER III

OVERVIEW OF *EvoC*

The general framework that we use for system-level hardware/software partitioning is depicted in Figure 1.

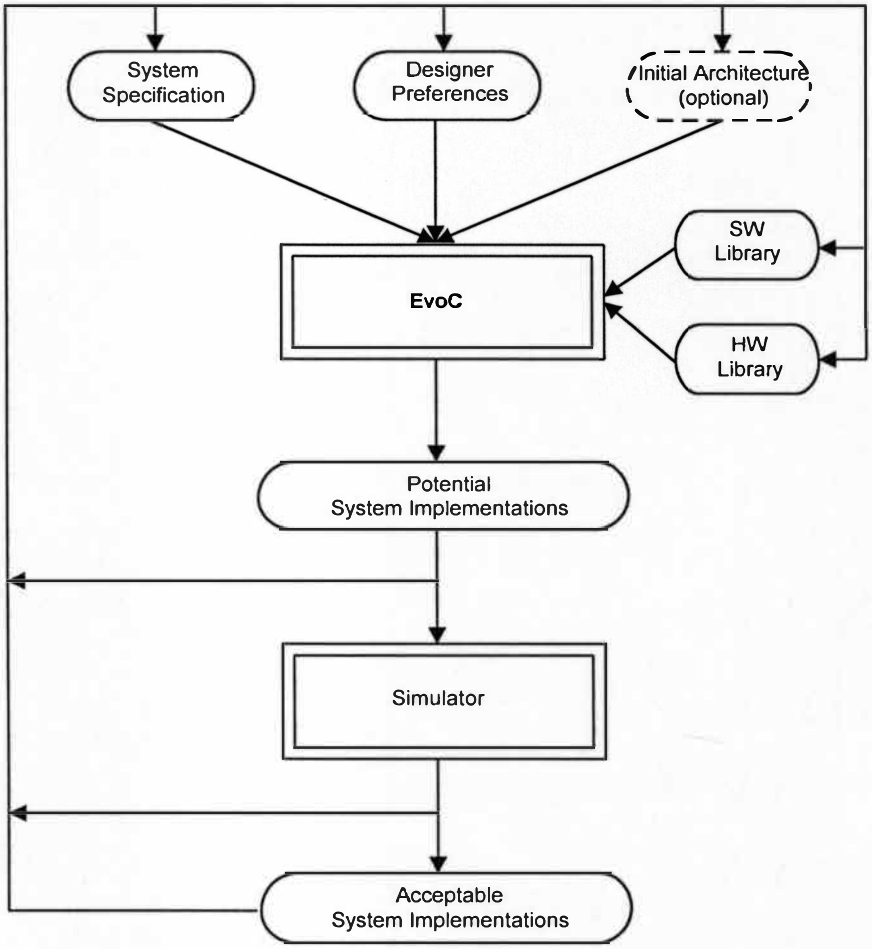
System Specification
Designer Preferences
Initial Architecture (optional)
SW Library
EvoC
HW Library
Potential System Implementations
Simulator
Acceptable System Implementations

Figure 1. System Level Hardware/Software Partitioning Approach.

Based on the given system specifications, component (hardware and software) libraries and designer preferences regarding various quality measurements (*e.g.*, cost, power consumption and expandability), system architectural configurations of increasing quality are identified by the optimizer *EvoC* (*Evolutionary Codesign*). Some of the configurations identified by *EvoC* may have to be simulated to verify their acceptability. Currently this is achieved by using a commercial simulation tool `SES/Workbench`[1].

The simulation results may indicate that a design configuration is not acceptable. This can be caused by any number of factors. *EvoC* does not have the capability of resolving conflicting preference information furnished by the designer or conflicting system specifications. Component libraries may have to be modified to ensure that adequate resources are available. In certain cases the system specifications may require reinvestigation. The scope of this report is, however, limited only to *EvoC*. The following sections describe the partitioning problem formally and the techniques that are used in *EvoC* in solving it.

Formulation of the Partitioning Problem

At the system level, hardware is modeled as resources (or components) with no detailed functionality and software is modeled as tasks utilizing the resources. To find an "optimal" implementation (a design configuration) for a given system

---

[1] A product of Scientific and Engineering Software, Inc., Austin, Texas.

specification, we need to quantify "optimality". Quite often, several attributes are used to gauge the quality of a system, e.g., cost, chip area and power. In RTES, timing related attributes, such as feasibility factor and critical excess MIPS are also important attributes [4].

The partitioning problem has been stated as an optimization problem by D'Ambrosio, Hu and Greenwood [14]. According to them, the system specifications are modeled as a collection of processes, f. Associated with each function, there are constraints and timing requirements, $\Re$. The different components available for implementing these processes are maintained in different libraries, Υ. Let a_k be the k-th attribute and w_k be the weight associated with a_k. Consequently, the partitioning problem can be formulated as an optimization problem as follows:

$$\begin{aligned} &\text{Maximize: } \textstyle\sum_k w_k a_k(\chi) && (1)\\ &\text{Subject to: } \{\cup_{x \in \chi} f(x)\} \supseteq f \\ &\qquad\qquad\quad \Re(\chi) \\ &\qquad\qquad\quad x \subseteq \Upsilon \end{aligned}$$

where $f(x)$ represents the processes implemented by module x, and $\Re(\chi)$ represents the set of constraints to be satisfied by an implementation χ.

Generally the design space is too large and forbids an exhaustive search for solving the above optimization problem. In the past, simulated annealing has proven to be capable of finding good solutions. However, the long execution time has been cited as a disadvantage of this technique [1]. Even randomized search has been shown to outperform simulated annealing if the global optimum is sought [2]! Hence, we resort to the *Evolutionary Algorithm* (EA) to solve the optimization problem. EAs are stochastic search techniques based upon population genetics. EAs have generated a great deal of recent research interest because of their ability to identify good solutions to NP-hard problems [7].

Another hitch in solving the above described optimization problem is that of combining multiple attributes as a single objective function. Typically, multiple attributes have been combined in an *ad hoc* manner to form a scalar objective function, usually through a linear combination (weighted sum) of the multiple attributes, or by turning objectives into constraints (with associated thresholds and penalty functions). The final solution is usually very sensitive to small changes in the penalty function coefficients and weighting factors. A technique based on *Imprecisely Specified Multi-Attribute Utility Theory* (ISMAUT) from the field of decision analysis may be used to handle tradeoffs among the attributes based on user's preferences.

The ISMAUT technique provides a method for combining multiple attributes, but does not address the difficulty of searching large problem spaces. The EA, on the other hand, are well suited to searching intractably large, poorly understood problem spaces, but have mostly been used to optimize a single objective. EA and ISMAUT are therefore complementary techniques for optimization and design. ISMAUT has no specific method for handling intractable search spaces while traditional EA assumes a single attribute. The direct combination of ISMAUT and EA is the next logical step for multi-objective EA optimization [12].

A Generalized Evolutionary Algorithm

EAs are heuristic techniques based upon the principle of adaptive selection in the natural world. A population consists of a set of *individuals,* where each individual is a solution to the problem. The particular genetic encoding for an individual is referred to as the *genotype.* Decoding this genetic material gives the set of observed characteristics of the individual which is referred to as the *phenotype.* *Genetic operators* (*e.g.,* mutation and recombination) produce offspring by slightly altering the genotype of the parents. Mutation operator takes a single parent to produce an offspring. In the evolutionary framework, the *fitness* of an individual is measured only indirectly by its growth rate in comparison to others, *i.e.,* its propensity to survive and reproduce in a particular environment. In *EvoC*, fitness is represented by the rank that is assigned to an individual.

Each *generation* (iteration) of the EA takes a population of individuals (potential solutions) and modifies the genetic material (genotype) to produce offspring (new solutions). Only the highest fit individuals (*Selection*) survive for the next generation. Tournament Selection is a type of selection in which candidates (individuals competing for selection) are compared against a set of randomly chosen individuals from the current population. EA have been successfully used to solve various types of optimization problems.

The EA terminates after a fixed number of generations (Γ) have been produced and evaluated or earlier, if an acceptable assignment has been found. The EA algorithm is implemented as follows:

1. Create an initial population of μ design alternatives by randomly assigning functions as either hardware or software implementations.
2. Conduct a tournament to select alternatives for reproduction. Each selected alternative generates one offspring by applying mutation operators. This creates a population with a total of 2μ alternatives.
3. Rank all alternatives according to their fitness.
4. Deterministically select the μ alternatives with the highest fitness.
5. Proceed to step 2 unless an acceptable solution has been found or Γ generations have been evaluated.

Imprecisely Specified Multi-Attribute Utility Theory

The ISMAUT technique is used in *EvoC* for evaluating the design configurations and is explained in detail in [9]. A preference relationship is used to assign fitness to each design configuration. Once a design configuration is identified, its attributes (*e.g.,* cost, power consumption or speed) can be quantified. To reflect the designer's preferences in the trade-off of different attributes, we make use of imprecisely specified value functions which are taken from the field of utility theory [16]. Attribute raw scores are mapped to values in the interval [0.0, 1.0] by a *value function*. An imprecisely specified multi-attribute value function corresponding to the design configuration χ has the following form:

$$V_{\chi} = \sum_{k} w_k v_k(a_k(\chi)) \tag{2}$$

where $w_k \in R_+$ is the weight and $v_k(a_k)$ is the attribute value function for attribute a_k. All the weights must satisfy $\Sigma_k w_k=1$; $w_k>0$. We denote a design configuration χ is preferred to a design configuration χ' by $\chi \succ \chi'$. Design configuration χ is said to have a higher fitness over design configuration χ if $\chi \succ \chi'$.

V_{χ} is imprecise in the sense that each w_k does not have a specific assignment, but is constrained by preferences among attributes. Such constraints can be formulated based upon preferences between distinct design configurations (provided by the designer or generated by an EA). Using the attribute value functions and the set of w_k constraints, other configurations created by running the EA may be evaluated.

Integrating ISMAUT With EA

The approach used in *EvoC* to conduct hardware/software partitioning can be summarized as follows (*c.f.,* Figure 2). Initially, we obtain a small set of solutions or implementations which can either be generated by our EA technique or can be given by the designer. The designer ranks the implementations. The ranking can simply be pair-wise comparisons of the implementations. No total order of the implementations are needed. Based on the ISMAUT, the ranking information supplied by the designer defines the designer's preferences, which are stored in a file ("PREF.in").

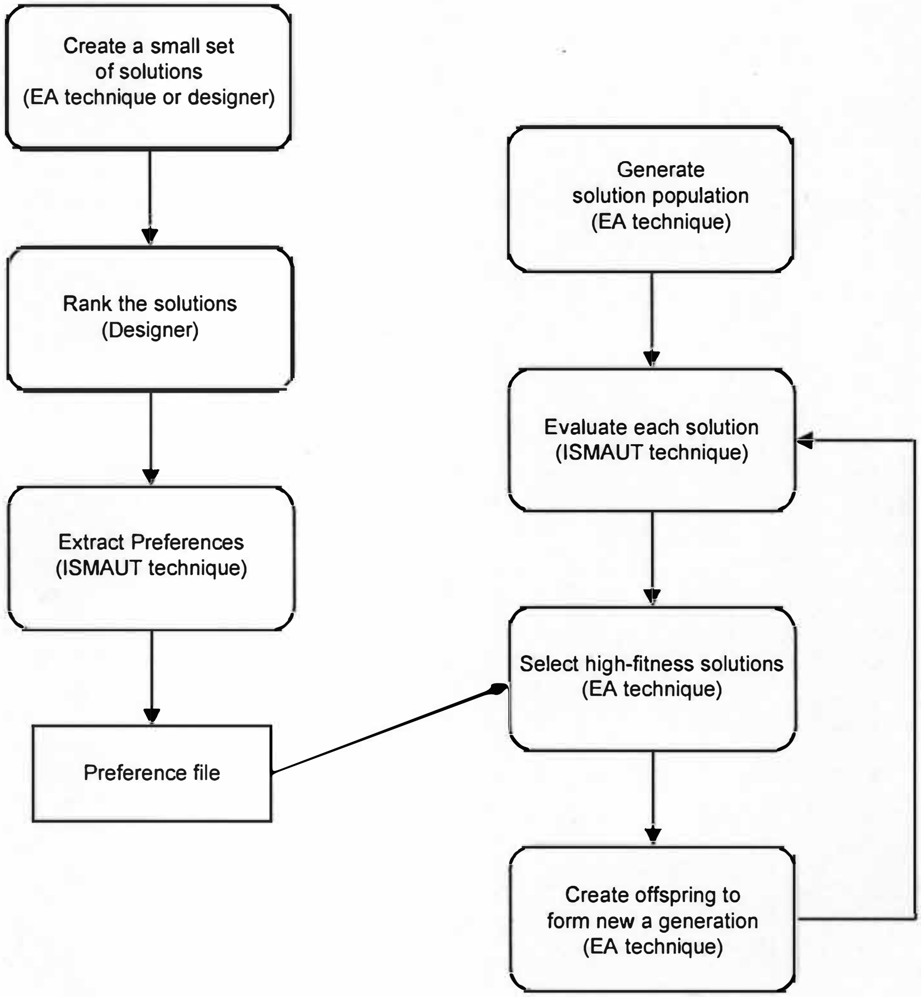

Figure 2. The *EvoC* System.

It is important to emphasize that the initial ranking of the selected design configurations is done merely to obtain the constraint subspace W', which is then used in evaluating other design configurations considered during the EA implementation. A detailed description of using preferences to represent fitness in EA can be found in a forthcoming publication [9].

The EA works in a conventional manner of using genetic operators to generate new potential solutions. ISMAUT compares a pair of individuals (say, χ and χ') at a time. If neither of the individuals *dominates* (*i.e.,* solutions which at least as good as any other solutions with respect to every attribute value) the other, then the preference check is done by solving the following linear program.

$$\begin{aligned} &\text{Minimize (w.r.t. } w_k\text{):} \ \sum_k w_k\left[v_k(a'_k) - v_k(a_k)\right] \\ &\text{Subject to: } \ w_k \in W' \end{aligned} \tag{3}$$

where, W' is the constrained weight space. If the result is greater than zero, then χ' is preferred to χ. Based on the evaluation done by ISMAUT, *EvoC* assigns ranks to individuals. The ranks are used as fitness measures for determining if individuals survive.

In a multi-attribute optimization problem a set of solutions are *non-dominated* in the sense that there exists no other solution that is superior in all attributes. In *attribute space*, the set of all non-dominated solutions lie on a surface known as the *Pareto optimal frontier.* The goal of *EvoC* is then to find and maintain a representative sampling of solutions on the Pareto front that match the designer's preferences.

Handling Timing Constraints in RTES

When a set of time-critical processes are assigned to a processor as a set of software tasks, the partitioning process must determine if such an assignment can satisfy all timing constraints, *i.e.,* meet the deadline requirements of the processes. In addition to considering only guaranteed feasible designs (which can be quite costly and have a low processor utilization), we need to evaluate possibly feasible designs also. Hence a metric called *feasibility factor* [4] is used to indicate the possibility of an assignment being feasible.

The feasibility factor for processor *P* is defined as,

$$\lambda_P \equiv \begin{cases} \dfrac{TR_P - TR_L}{TR_U - TR_L} & \text{if} \quad TR_P - TR_L < TR_U - TR_L \\ 1 & \text{otherwise} \end{cases} \tag{4}$$

where, TR_P is the throughput rate of processor *P* (given in MIPS), TR_L and TR_U are the lower and upper bounds of the minimum throughput requirement in order for processor *P* to feasibly schedule all the processes assigned to it. In then follows that the process set on processor *P* is feasible if $\lambda_P = 1$, and it is not feasible if $\lambda_P<0$. For $0 \leq \lambda_P \leq 1$, the larger the value of λ_P, the greater the chance for the process set to be feasible. Hence, λ_P indicates the possibility of processor *P* being able to meet all the timing requirements of the processes assigned to it. Given the throughput rate of a processor and the set of processes to be executed, the feasibility factor can easily be calculated [4].

Other timing-related attributes may also be included for evaluating the performance of RTES. An important property of an embedded system is its *expandability*. To limit costs, much of the hardware and software of an embedded system must be reusable through several design cycles and accommodate increasingly demanding functionality over the life of the design. Therefore, a designer may be willing to tradeoff cost for expandability (*i.e.,* to increase the task execution requirement) in a particular design. To model the expandability of a RTES, we introduce an attribute called *critical excess MIPS,* Δ_C. It is defined as $\Delta_C \equiv TR_P - TR_L$. Clearly, the value of Δ_C is an estimate of the amount of peak execution power that a processor has after meeting the timing constraints of the current process specifications. A larger Δ_C will allow the current process to be expanded and still be feasible. It may also allow the system to handle new time-critical processes.

CHAPTER IV

SOFTWARE DEVELOPMENT OF *EvoC*

EvoC (Evolutionary Co-design) is a CAD tool developed using C/C++ language on a UNIX operating system, for the automatic hardware/software partitioning of RTES, based on EA. This chapter discusses the conceptual base for forming the data structure in its present form and the implementation of this EA based partitioning approach.

Fundamental Concepts Underlying the Design of *EvoC* Data Structure

A set of standard library of components exists, and is referred to when forming a design configuration. This need not be completely replaced by a new set when a different problem is considered. The user-given system specifications are, however, problem specific. The system implementation, as shown in Figure 3, is a mapping from the system specifications (processes) to the component library.

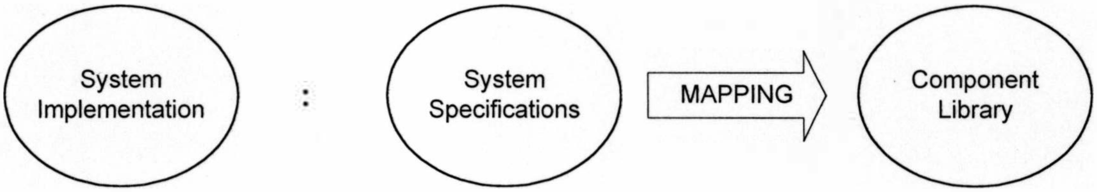

Figure 3. Mapping for System Implementation.

The objective of the *EvoC*'s *Data Structure* is to capture the problem parameters in the genotype of an individual for a wide variety of hardware/software partitioning problems in RTES and to facilitate implementation of EA.

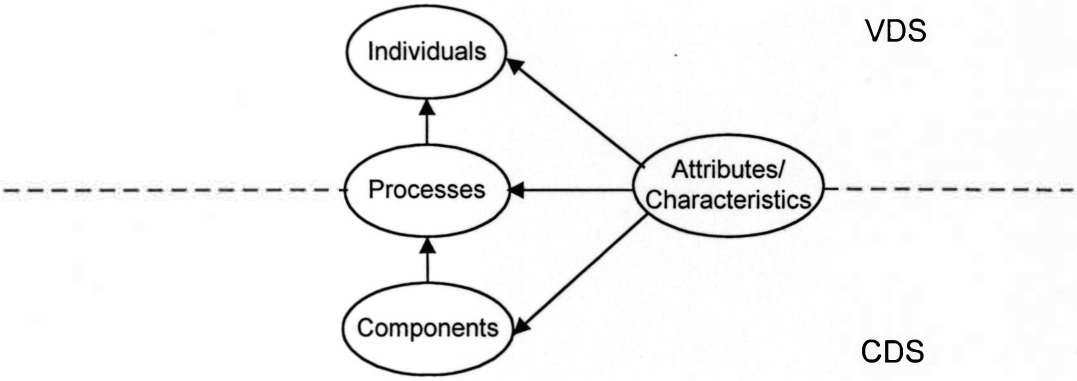

Figure 4. Hierarchy of *EvoC* Data Structure.

Figure 4 illustrates the hierarchy of the *EvoC* data structure. The actual data structure is given in Appendix B. Some of the terminologies which also forms the different levels of the hierarchy are defined as follows:

A configuration fully implementing the user's specifications of the system is called as *Individual* or *Alternative*.

Process refers to a function or a task that a system has to perform.

Components are physical entities, an IC chip for example, that are available as off-the-shelf items in the market or custom designed modules like ASICs or software routines capable of implementing system processes.

Component Cells form the fundamental building blocks of any component. For example, a microcontroller component may be composed of the following cells (with the size or number indicated within brackets) – cpu[1], volatile memory[2000], non-volatile memory[2000].

Attributes are used to determine the degree to which properties of a good alternative is met. Examples include cost, power consumption and critical excess MIPS.

Characteristics are properties associated with components or processes. For example, the real-time characteristics of user-defined system processes are activation, period and deadline.

The *Constant Data Structure* (CDS) is composed of system process specifications, components and cells with their associated characteristics as given by the user (of *EvoC*) and they remain unmodified for a given design problem. CDS serves as a reference library when new alternatives are created. The *Variant Data Structure* (VDS), on the other hand, is the system implementation itself that keeps changing every generation and can be considered as a subset of CDS. It also contains the system attributes.

A processes can be a *Functional Process* (FP) or *Communication Process* (CP). All functions or tasks that a system has to perform are categorized as FPs and the point-to-point communication link between two FPs (implemented on different resources) for exchange of data between the FPs are modeled as CPs. Note from

Figure 5 that if FP_2 must send data back to FP_1, another distinct CP is required. Also, the quantity in brackets next to the CP indicates the number of bytes in the data transfer.

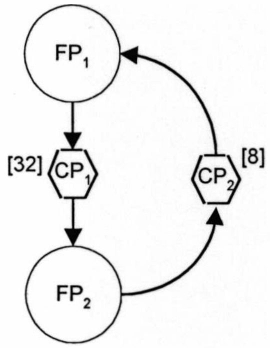

Figure 5. Data Flow Graph Based on FPs and Cps.

FPs are further classified as *User-Defined Processes* (UDPs) and *Additional Processes* (APs). UDPs are FPs defined by the user. APs are FPs "created" by *EvoC* in order to satisfy the system's requirements for completeness (refer to the section Partitioning) and are appended to the list of UDPs in the class `Func_Imp[]`.

CPs can be of two types, *Regular CPs* (RCPs) and *Mono-link CPs* (MCPs). The basic difference between them is whether the CP has both input and output UDP (defined as a RCP) or just one of them (defined as a MCP).

Description of *EvoC* Data Structure

The classes in *EvoC*'s data structure are formed to support the different levels in the hierarchy of the data structure. Each level typically has two classes associated with it, one for specification (definition) and the other for implementation (instantiation). For example, the functional processes are defined in the class `Func_Spec[]` and the actual implementation details corresponding to all individuals are contained in the class `Func_Imp[]`. A similar structure holds for the class `Attr_Type[]` and `Attr_Val[]` that specify the attributes and characteristics, and for the class `Comm_Spec[]` and `Comm_Imp[]` that specify the communications.

It can be noted from Figure 6 that all specifications form part of CDS and the implementation details are in VDS. Some other classes that define components, component cells and individuals do not have this two-tier description as specification and implementation classes.

All levels in the hierarchy have associated attributes or characteristics. The characteristics are present in the specification classes to avoid multiple copies being present for every individual in the implementation class. The implementation classes contains the implementation details of all the individuals from the current generation.

This is shown in Figure 7. The top half of the implementation arrays, `Func_Imp[]` & `Individual[]`, contain parents and the bottom half contain the off-spring.

Certain features unique to some of the classes are worth mentioning here. The class `Func_Spec[]` lists the set of all components capable of implementing that FP.

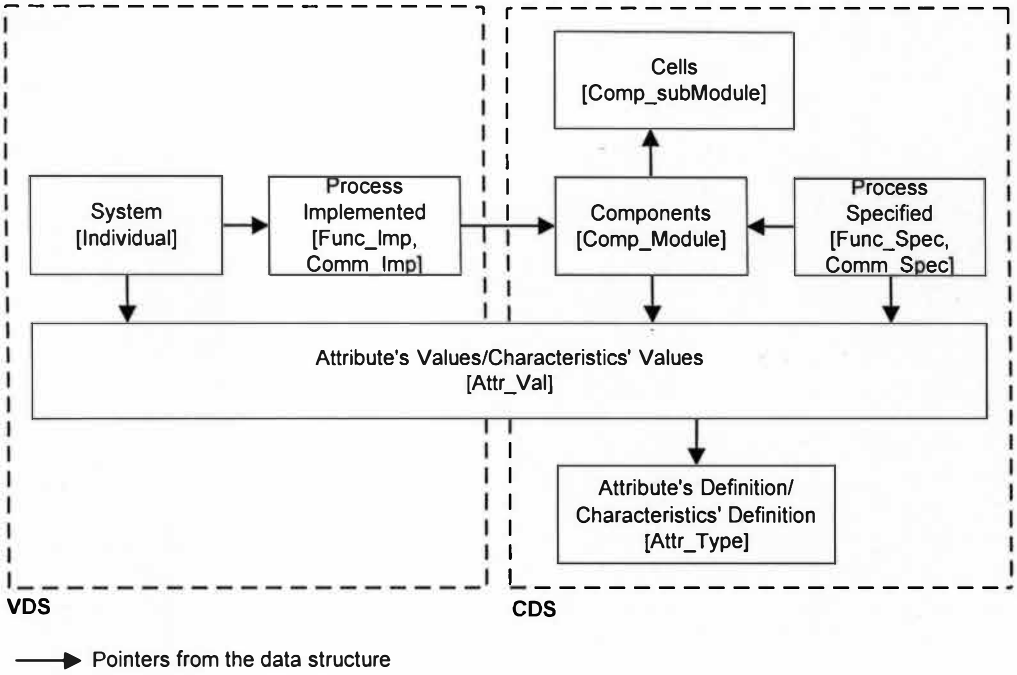

Figure 6. *EvoC* Data Structure.

This can be conveniently utilized during the initial partitioning or mutation, for choosing an implementation for the FP.

A CP might require many components (such as bus, memory, glue logic, *etc.*) to implement and the class `Comm_Imp[]` is flexible enough to handle it. The data structure for the communication aspects are not tailored to the capabilities of the present communication model alone. In fact, it is capable of supporting multiple communication models in the same version of *EvoC* with none or very little modifications to it.

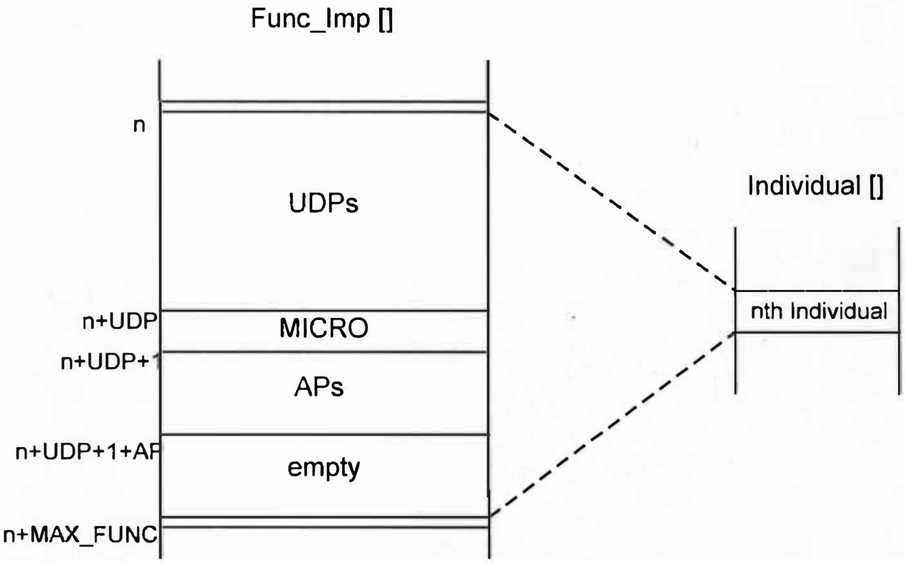

Figure 7. Mapping Between `Func_Imp[]` & `Individual[]` Arrays.

The class `Comp_Module[]` contains all component libraries as its different objects. In addition, it also specifies the different cells a component may require or supply. The following component libraries are found to cover the vast expanse of available components:

1. *Software library* (SW_LIB) contains software routines for the processes.
2. *ASIC library* (ASIC_LIB) contains custom made IC chips for implementing different processes.
3. *Programmableware library* (PW_LIB) contains programmable components like PLAs, FPGAs, etc.
4. *Microprocessor library* (MICRO_LIB) contains different microprocessors and microcontrollers

5. *Miscellaneous library* (MISC_LIB) contains components such as memory chips, timing channel ICs, *etc.*, which are used as auxiliary components rather than components that implements FPs in the system.

An implementation *type* of a FP may be software, hardware or programmbaleware.

Pseudo-Components

A special class of components called *pseudo-components* is discussed here. The user can exploit this technique in order to consider programmable devices in the system configuration. Certain components may not be selected to implement FPs on a one-to-one basis. For example, more than one FP may be implemented on a single FPGA chip.

EvoC assumes that any component specified under the component's list of a FP can only be used by this FP. But, as we have pointed out, an FPGA chip may be able to implement several FPs. To handle such shared allocations pseudo-components are introduced. It allows the user to specify the FP's requirement of a component in terms of number of cells. For example, a pseudo-component (representing a programmable device) for a FP indicates the number of gates (a cell) required by the FP if implemented on the programmable device. In addition, a pseudo-component represents an implementation type for that FP.

Pseudo-components are effectively utilized during the different phases of partitioning for assessing the number of physical components (*e.g.,* FPGA) needed to implement a partitioned set of FPs. This is discussed in length with examples in the section Partitioning.

User Input Files for *EvoC*

The set of input files constructed for a design problem and their contents are discussed here. Appendix A gives the detailed input file formats. The input file formats of *EvoC* forms a generic platform for collecting system information from the user.

The CDS is filled by reading in from the following input files:

1. *attr.dat* contains attribute's definitions for the individuals and characteristics definition for processes and components, and is used to form the class `Attr_Type[]`.
2. *func.dat* contains all UDPs to be implemented and is read into the class `Func_Spec[]`.
3. *comm.dat* specifies communication as in the form of CPs and fills in the class `Comm_Spec[]`.
4. *PREF.in* indicates the user preferences as phenotypical characteristics.

The following files contains components and are directly read into the corresponding component libraries specified as objects of the class `Comp_Module[]`.

1. *sw.dat* contains all software modules for different processes.
2. *asic.dat* consists of custom made ICs or ASICs.
3. *pw.dat* contains pseudo-components for processes that can be implemented on a programmable hardware such as timing channel, FPGA, PLA, *etc.*
4. *micro.dat* contains various microprocessors and microcontrollers.
5. *misc.dat* contains miscellaneous components that supports the functioning of components from other libraries. Unlike the components described above, these components can be considered as *accessory components* for system implementation, which does not implement any UDP directly but is needed for system operation. Typically this file contains memory chips, buses and also the physical entities of pseudo-components.

Partitioning

Partitioning is the allocation of hardware or software components to FPs and is carried out during the generation of initial population. Figure 8 illustrates the sequence through which the partitioning is done.

Partitioning is purely stochastic in nature and is done in two phases. In the *first phase of partitioning,* a component is chosen to implement each FP along with a microprocessor. This is a random selection of a component from the component's list of each FP specified in the class `Func_Spec[]`. The selected components may include actual components that implement FPs on a one-to-one basis and pseudo-components. In the *second phase of partitioning*, system accessory components and

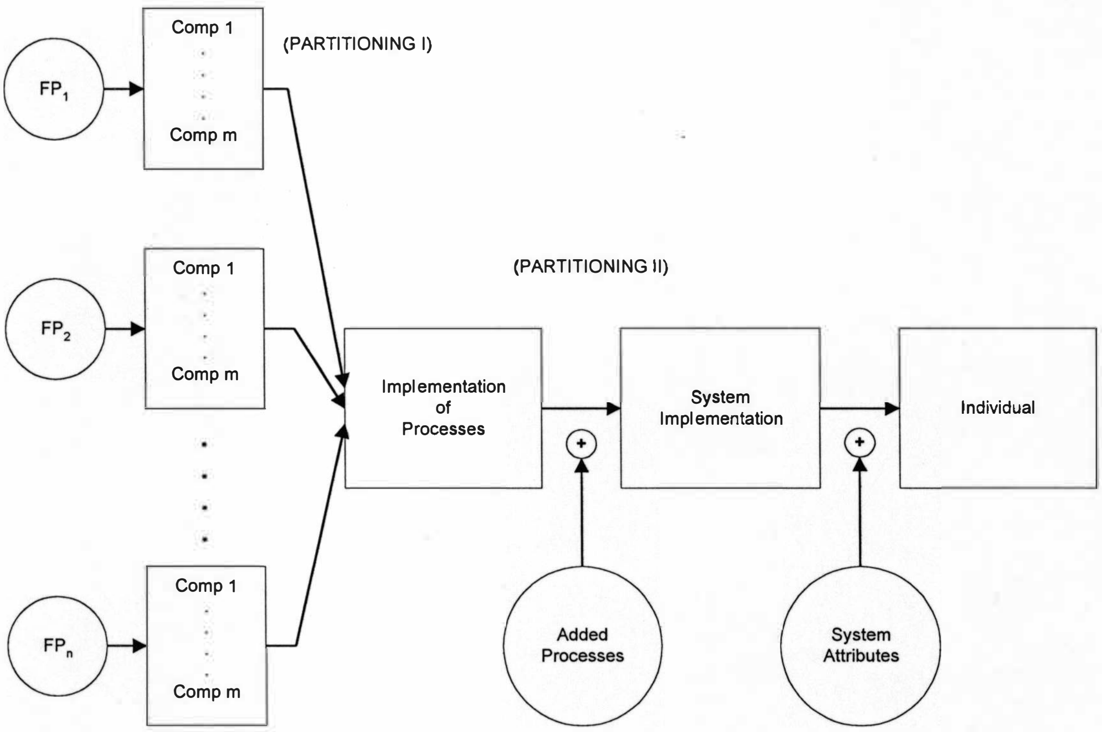

Figure 8. Partitioning During Generation of Initial Population.

the physical entities of pseudo-components are selected. System accessory components include different kinds of memory chips selected to support the operation of software components in the system. Also, a set of APs are attached to the list of FPs in the class `Func_Imp[]` to account for the inclusion of accessory components and physical entities of pseudo-components into the system. APs are created by the software to be able to represent the components selected during the second phase of partitioning in the class `Individual[]` and is transparent to the user.

This phased partitioning scheme is based on the observation that the accessory components do not need to participate in the random selection process. Instead, *EvoC* gets a complete count of the number of different cells required from all the primary components selected during the first phase of partitioning. Based on this information sufficient number of additional components (physical entity of pseudo-component or accessory component) can be selected to completely specify the system, in the second phase of partitioning.

For example, the total instruction memory requirements from all the SW components (selected during the first phase) is unavailable during the first phase of partitioning. Hence, it is during the second phase of partitioning that the memory chips are selected based on the total memory requirement.

As another example, a set of FPs may be implemented on timing channels and a peripheral chip may contain many timing channels. In such cases, a pseudo-component can be introduced for those FPs, which will specify the number of timing channels required to implement them. First phase of partitioning selects such pseudo-components. In the second phase of partitioning the total number of timing channels required is calculated from the selected pseudo-components and sufficient number of peripheral chips are chosen.

Phased partitioning scheme often leads to "sharing" of components among FPs. For example, a physical entity of a pseudo-component may be selected to implement multiple FPs or several RAM chips may be used to support software execution on the microprocessor selected. This results in better optimization. ASICs may also exhibit sharing if it is capable of implementing multiple FPs. In short, pseudo-components provides the user with yet another dimension of flexibility to distribute the assignment of the processes among an optimized set of components. The next step in partitioning is the selection of buses and bus memory components to implement CPs. This is discussed in Chapter V.

Ranking and Selection

After initial population is generated, *EvoC* needs to select highly fit individuals for survival in the subsequent generations. Selection of off-spring for survival is deterministic by a ranking procedure. Ranking is based on the preference relationships between individuals determined by ISMAUT. The *Heapsort Algorithm* is implemented to rank and sort μ individuals to be passed on to the next generation, which are the fittest among a total population of 2μ individuals. It sorts by comparison and requires only *O(nlogn)* comparisons to sort n individuals. It calls the ISMAUT routines which in turn calls the linear programming (LP) software lp_solve (ver. 2.0) for solving the LP problem set up by ISMAUT.

ISMAUT evaluates individuals having multiple and sometimes conflicting attributes, as was explained in Chapter III. The types of system attributes that are handled in the present form of *EvoC* are feasibility factor, critical excess MIPS and several additive attributes (*e.g.,* cost and power). Feasibility factor and critical excess

MIPS are special attributes to handle timing constraints in RTES. In addition to being an attribute, feasibility factor is also considered as a system constraint. By this, we mean that all individuals selected should have a feasibility factor greater than zero. A more detailed background on feasibility factor and critical excess MIPS can be found in [4].

Deterministic Selection of Parents

EvoC implements *tournament selection* to select individuals (parents) for reproduction. This technique along with *niching* enables us to have a diversity among the population in the design space. In tournament selection, a set of individuals (comparison set) is randomly chosen from the current population and in a *binary tournament* two randomly selected individuals are compared against this comparison set. If one candidate is preferred by the comparison set, and the other is not, the later is selected for reproduction. If neither or both are preferred by the comparison set, we use Equivalence Class Sharing to choose a winner [12]. This ensures genetic diversity along the population fronts and allows EA to develop a reasonable representation of the Pareto optimal front. By adjusting the size of the comparison set (t_{dom}) we can exert some control over the amount of *selection pressure*. The following pseudo-code, given by Horn and Nafpliotis [12], is implemented in *EvoC*.

PSEUDO-CODE 1

```
select ():
begin
```

```
        shuffle (Population[POP_SIZE]);
        candidate_1=Population[1];
        candidate_2=Population[2];
        candidate_1_preferred=true;
        candidate_2_preferred=true;
        for pop_index = 3 to tournament_size+3
            begin
                if (Rank[candidate_1] > Rank[Population[pop_index]])
                    candidate_1_preferred=false;
                if (Rank[candidate_2] > Rank[Population[pop_index]])
                    candidate_2_preferred=false;
            end
        if (candidate_1_preferred and ¬ candidate_2_preferred)
            return candidate_1;
        else if (candidate_2_preferred and ¬ candidate_1_preferred)
            return candidate_2;
/* do sharing */
        else if (niche_count[candidate_1] > niche_count[candidate_2])
            return candidate_2;
        else
            return candidate_1;
end
```

Niching is employed in the implementation of *Equivalence Class Sharing.* The goal of equivalence class sharing is to facilitate the exploration of the design space. This is achieved by picking parents from regions (in the design space) that are not densely populated with other individuals in the current population. A *niche count* (m_i) is calculated for those individuals tied in a tournament selection. The niche count is an estimate of how crowded the neighborhood (niche) of an individual is and is calculated over all individuals in the current population.

$$d[i,j] = \left[\sum_{k} \left| a_k(\chi_i) - a_k(\chi_j) \right|^{\rho} \right]^{1/\rho} \qquad (5)$$

where, d[i,j] is the distance between individuals χ_i and χ_j, $a_k(\chi_i)$ is the normalized value of the *k-th* attribute of individual χ_i and ρ (a value of 0.5 is used in *EvoC*) is the degree of the *Holder metric.*

Once the distances are computed the niche count can be found using the following pseudo-code.

PSEUDO-CODE 2

```
begin
    for candidate_i = 1 to POP_SIZE
        for candidate_j = 1 to POP_SIZE
            if (( d[i,j] < $\Omega$) and (i $\neq$ j))
                niche_count_i=niche_count_i+1;
end
```

where, Ω is chosen to be a very small number, which defines the boundary for the niches in the attribute space. A smaller value will be chosen for this constant if the population size is large.

Equivalence class sharing assumes that most of the individuals in an equivalence class may be labeled as "equally" fit. Individuals within close proximity in the design space tends to have a higher niche count as they are all in the same niche. Hence sharing (refer to pseudo-code 1) would select for reproduction the individual (parent) with the smallest niche count.

Mutation

EvoC implements mutation operation to create off-springs (new system implementations). Mutation operators are chosen based on what the user wants to investigate since different operators have different effect on the genotype of an individual. We have used three mutation operators (M_1, M_2, M_3) — M_1 for perturbing the microprocessor chosen for an individual, M_2 for selecting between hardware and software implementations of FP and M_3 for re-assignment of a hardware implementation of FP. Each mutation operator is applied with a probability denoted by ρ_{M_i}. In the current implementation ρ_{M1}, ρ_{M_2} and ρ_{M_3} are 0.15, 0.50 and 0.35, respectively.

A mutation operation is always followed by second phase of partitioning for the off-spring. This does not produce a drastic difference between the genotypes of the parent and the off-spring because the components that gets replaced (in the second phase of partitioning) fall under the category of system support components or accessory components. Their effect on the values of the system attributes is relatively small compared to that of the components implementing FPs. Hence this does not perturb the position of the individual in the attribute space greatly. Mutation will be revisited when the communication aspects are discussed in the next chapter.

This chapter explained how the data structure has been designed to capture the system information from the user. The list of input files that have to be given by the user and their contents are given. Finally, implementation details of the different steps in an EA, such as partitioning scheme, selection procedure and mutation operators were presented.

CHAPTER V

MODELING OF COMMUNICATION

We used a *Multiple Bus and Shared Memory* (MBSM) model for the communication scheme in the design of embedded systems at the system level. The components required for this model are buses and memory chips.

Multiple Bus and Shared Memory Communication Model

The MBSM communication model has been added to *EvoC* in version 3.0. In this model, communication is modeled as CPs which are further divided into two types, RCPs and MCPs [refer to Chapter IV]. A RCP specifies the communication link between two FPs. A MCP is a communication link between a FP and shared memory. This is the case when the system has to communicate with its environment, *e.g.,* some external signals are read into some internal buffers (memory) in the system for later use. Figure 9 shows an example system configuration with its communications links.

An *inter-process communication* need not be synchronized since processes may run at different rates. Hence our MBSM model addresses the issue of *asynchronous communication.* However, the cost and delay for inter-process communication cannot be ignored. The communication cost is due to addition of

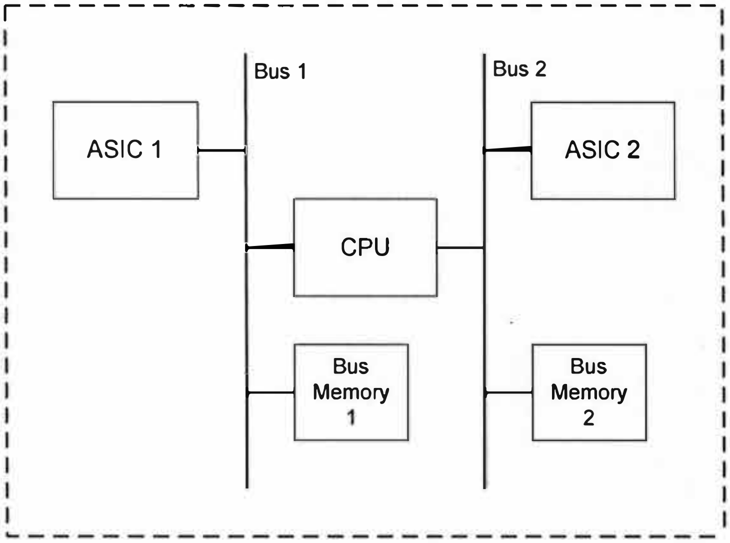

Figure 9. An Example System Configuration.

hardware and the delay is primarily the result of bus transfer time (i.e., arbitration time and propagation delay) and bus memory access time (explained below). They, of course, depend on the amount of data that needs to be transferred.

Some systems attempt to reduce bus transfer times by adding redundant buses; a processor selects an available bus rather than waiting to arbitrate for a single bus. *EvoC* has been modified to model such systems so that the impact of communications can be ascertained. This becomes imperative when attempting to design systems with real-time constraints.

We assume that the inter-process communication overhead between processes implemented on a common resource to be zero. On the other hand, the communication between processes implemented on different resources is *via* the bus memory, which is local to every bus in the system. This means that there exists buses connecting every subset of processes that needs to communicate with each other. The MBSM model may result in multiple bus selections and every component may be connected to one or more buses based on the communication requirement of the processes implemented on that resource (or component).

Incorporating MBSM Model Into *EvoC*

The following characteristics are introduced to support the MBSM model:

1. BW denotes the bandwidth of a bus which is defined as the amount of data it can transfer in unit time.

2. *NB* is the number of bytes per transfer for a CP.

3. *NC* is a characteristic of a SW component that indicates the communication overhead in terms of the number of processor instruction counts if the FP implemented by the SW component needs bus communication.

4. τ_I and τ_S are the initial and the subsequent memory access times associated with a bus memory.

Selection of buses is done in a manner which minimizes the total number of buses in the system and meets the throughput requirements of all CPs. Selection of a new bus is avoided if the CP under consideration could be assigned to an already selected bus (subject to bus throughput). This leads to better bus utilization. Let TP_i denote the throughput requirement associated with the *i*-th CP. TP_i can be calculated as follows:

$$TP_i = \begin{cases} \frac{NB_i}{P_{inp}} + \frac{NB_i}{P_{out}} & for \quad RCPs \\ \frac{NB_i}{P_{inp}} (or) \frac{NB_i}{P_{out}} & for \quad MCPs \end{cases} \tag{6}$$

where, P_{inp} and P_{out} are the periods of the input FP and the output FP, respectively. Each CP assigned to a bus imposes a throughput requirement on that bus. Assigning CP_i to a bus B is considered infeasible if $\sum_{k:CP_k \in B} TP_k > BW_B$. In such cases a new assignment is sought.

When computing the bus memory requirements, the *NB* from all CPs attached to the bus are added together, irrespective of whether they are from a RCP or a MCP. However, their difference is emphasized when computing throughput requirement TP_i since the rate of bus usage is twice for a RCP as compared to a MCP.

Communications overhead needs to be added to the processor executing the input or output FPs. This is modeled as an increase in the instruction count of the corresponding input and/or output FPs. Thus the instruction count of a software component consists of two quantities: the instruction count of computations (specified in the software component library) and the instruction count resulting from associated CPs which force bus communication (calculated as the characteristic *NC*). The instruction count for CP_i is calculated as,

$$NC_i = \left(\frac{NB_i}{BW} + \tau_I + (NB_i - 1) * \tau_S \right) * M \tag{7}$$

where M is the MIPS rate of the microprocessor executing the corresponding FP. Equation (7) is a simple conversion of bus transfer latency added with the memory access time to the instruction count of the processor executing the FP. This additional workload is used to assess the feasibility of software assignment to the processor. After successful selection of buses, memory requirements are aggregated and sufficient memory is allocated. The communication cost arising due to the additional hardware (bus and bus memory chips) selected to implement MBSM model is added to the total system cost.

Representation of MBSM Communication Model in the *EvoC* Data Structure

The class `Comm_Spec[]` captures the system's communication needs as CPs by explicitly specifying its input and output connections (FPs) and the *NB* associated with it. The class `Comm_Imp[]` gives the implementation details of each CP defined in the class `Comm_Spec[]`. The class `Bus_Comp[]` helps to view the system as a set of buses with attached components. It categorizes "buswise" the different components and CPs of an individual.

Representation of Bus in the Input Files

Bus is a component in MISC_LIB. All HW components have a bus-list which lists the buses that can be attached to this hardware. The SW components do not need a bus-list, but rather the microprocessors (implementing SW components) have them. In case of pseudo-components, the bus-list contains the buses which can be attached to their corresponding physical components. Hence, the actual physical component (present in MISC_LIB) need not have a bus-list.

Mutation With MBSM Communication Model

A mutation on a fully connected system may result in the following types of modification on system components (as shown in Figure 10): (a) *Reallocation* is reassignment of processes among the existing resource itself; (b) *Addition* is adding a new component to the system; (c) *Removal* is removing an existing component from the system and reassigning the process implemented on it on the remaining resources; (d) *Substitution* is assigning all processes implemented on an existing resource completely onto a new resource, and removing the unused resource from the system; and (e) *Processor Replacement* is choosing a new processor in place of the old one and reassigning all SW processes on the new processor and finding a different HW implementation for all HW processes implemented on the old microprocessor (as custom circuits).

Though mutation results only in a relatively small change with respect to system components (chosen during the first phase of partitioning), its effect can be large with respect to the overall system structure. For instance, some of the bus connections of the "old" component (the parent) may have to be severed if the "new" component (the off-spring) is not compatible with those buses. This leaves a hole in the communication set-up, which needs to be filled by some other means, such as introducing new buses or reusing the old compatible ones. This might very well lead to cascaded refinement of the entire communication system for the off-spring considered. Hence considering communications for mutation and at the same time trying to create an off-spring that has a system structure that is "similar" to its parent is certainly not a trivial task.

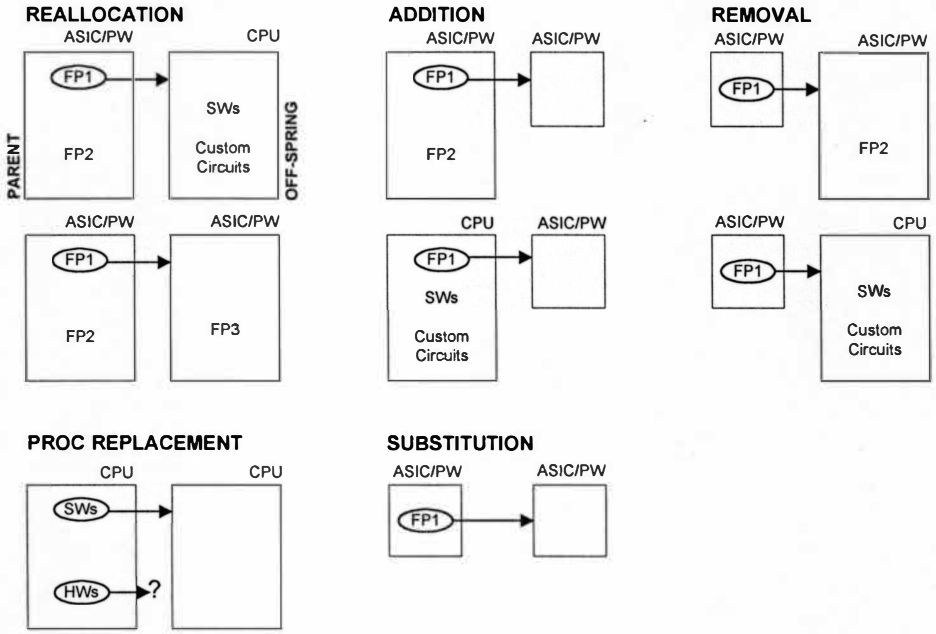

Figure 10. Types of Component Modifications Due to Mutation.

We have developed a method for adjusting the communication setup for the off-spring which addresses the above stated problem as much as possible.

Mutation operators are applied only to the set of FPs (before the assignment of CPs) in the individual. Then the bus compatibility of the newly selected component with the rest of the components which it needs to communicate with in the individual is examined. The following three cases can handle all of the mutation effects that were previously identified. This procedure is repeated for each CP that has the mutated FP as its sending or receiving FP in an individual.

Case 1: A CP corresponding to a FP that is mutated, is implemented on the same bus that implements the CP in the parent, if the newly selected component can be attached to that bus.

Case 2: If a CP cannot be implemented as explained in step 1 due to bus incompatibilities, then it is implemented on one of the existing buses in the individual that can be attached to the newly selected component.

Case 3: If a CP cannot be implemented on any of the existing buses, then a new bus is chosen to implement that CP from the bus-list of the new component.

Rationale Behind Multiple Bus Selection

A number of issues are worthwhile to be mentioned regarding the MBSM model. First, we notice that the choices of buses and memory chips are often constrained and their effects on the resulting architecture are relatively easy to predict. Random selection of buses often leads to invalid system configurations. Hence, though our *EvoC* tool selects hardware components and software allocation based on EA, we use a greedy algorithm for bus and memory selection.

We allow component compatibility to be a selection constraint. This is necessary when more than one family of processors are considered in the exploration process. Yet, one may argue that this is redundant for ASIC components since they can be made to be compatible with any buses. (Note that our *EvoC* tool does allow the specification of such "generic" components.) However, the capability of specifying ASICs with different bus interfaces and costs gives a mechanism for considering component reuse. For instance, if an ASIC with certain bus interface is already available, its non-recurring engineering cost will be much lower than a to-be-designed ASIC. Hence, we can model the two components of similar functionality with two different costs and let *EvoC* evaluate their overall merits.

CHAPTER VI

DESIGN EXAMPLES AND EXECUTION TIME ANALYSIS

EvoC has been tested on three design examples, two (Example I & II) without considering communications and the other (Example III) with communication aspects. The computing environment includes Sun Sparc IPC workstation, Sun OS and GNU C++ compiler (ver. 2.4). In this chapter, Example I & III are discussed in detail and the timing features with respect to all the sample runs are given.

Example I

This example is presented by Hu *et al* [4,14]. It involves the design of a RTES implementing a subset of an engine control module. Table 1 summarizes the set of FPs along with its characteristics and component list. The component libraries are listed in Table 2. It contains a collection of components including processors, ASICs, peripheral devices and memory chips. Table 2 gives an approximate value for the instruction counts for the SW components based on a generic RISC architecture. The number of cells implemented or required are indicated next to the cell itself, inside the brackets. . The memory (RAM & ROM) size is given in bytes and timing channels ("tc") as a number.

Table 1

Set of FPs With (*a, d, p*) in Microseconds and Their Components List

Functional Process	Activation (a)	Deadline (d)	Period (p)	Components List
FP1	0.00	46.00	104.17	SM1, ASIC1, MC1, MC4, MC5, MC6, FP1-PS
FP2	9895.83	10000.00	10000.00	SM2, ASIC1, MC1, MC4, MC5, MC6, FP2-PS
FP3	0.00	83.00	208.33	SM3, ASIC1, MC1, MC4, MC5, MC6
FP4	83.00	138.00	208.33	SM4, ASIC1, MC1, MC4, MC5, MC6
FP5	0.00	416.67	10000.00	SM5
FP6	0.00	208.33	416.67	SM6
FP7	833.33	1333.33	2500.00	SM7
FP8	1666.67	2500.00	2500.00	SM8
FP9	0.00	312.50	416.67	SM9

Population size (μ) chosen for this example is 45. Tables 3 and 4 summarizes a representative sampling of the typical solutions obtained from several runs (number of generations, Γ, is 20), when lower cost and higher feasibility factor value is preferred respectively.

The following observations can be made from Table 3. The microprocessor selected (MP1) is the least expensive of all available choices in MICRO-LIB. One or both of the FPs, FP1 and FP2 is implemented on a peripheral device (PIO1-IO) as timing channels in order to off-load additional SW components from being executed on the processor selected. Individual 2 from Table 3 should have had a larger Δ_C

Table 2

Component Libraries and Their Characteristics

Library Name	Component	Implemented Cells	Required Cells	Cost ($)	MIPS	Instr.
SW-LIB	SM1		ram[100], rom[100]			64
	SM2		ram[100], rom[100]			32
	SM3		ram[200], rom[300]			30
	SM4		ram[200], rom[300]			30
	SM5		ram[100], rom[100]			30
	SM6		ram[200], rom[200]			20
	SM7		ram[500], rom[400]			480
	SM8		ram[400], rom[300]			100
	SM9		ram[100], rom[100]			40
ASIC-LIB	ASIC1			2.50		
MICRO-LIB	MC1	ram[2000], rom[2000]		3.50	1.30	
	MC2	ram[2000], rom[2000], tc[32]		3.25	1.50	
	MC3	ram[4000], tc[16]		5.25	2.50	
	MC4	ram[4000]		6.25	2.50	
	MC5	ram[2000], tc[14]		3.75	1.70	
	MC6	ram[2000], tc[14]		3.25	1.35	
	MC7	ram[2000], tc[16]		2.50	1.70	
	MP1	ram[2000], rom[2000]		2.00	1.43	
	MP2			13.00	13.50	
PW-LIB	FP1-PS		tc[1]			
	FP2-PS		tc[1]			

Table 2—Continued

Library Name	Component	Implemented Cells	Required Cells	Cost ($)	MIPS	Instr.
MISC-LIB	RAM1	ram[2000]		2.00		
	ROM1	rom[2000]		1.00		
	PIO1-IO	tc[16]		1.00		

Table 3

Typical Solutions Found by *EvoC* When Lower Cost is Preferred

No.	Individual	Cost ($)	Feasibility Factor	Critical Excess MIPS
1	FP1 & FP2 on PIO1-IO, MP1	3.0	0.013	0.011
2	FP1 on PIO1-IO, MP1	3.0	0.009	0.011

Note: Remaining FPs are implemented in SW

when compared to individual 1. But their difference is so small that the Δ_C values appears as equal with the precision in Table 3. The above solutions obtained by *EvoC* can be simulated using the `SES/Workbench` to ascertain if the individuals are indeed feasible.

The results from Table 4 shows a variety of microprocessor selections among the individuals. All the solutions are definitely feasible since the feasibility factor is 1.0. Also, FP1 through FP4 are implemented in HW, as ASICs, timing channels or custom circuits on the microcontroller chips, in different combinations. This has the effect of increasing the feasibility factor and the critical excess MIPS values for the individual.

The solutions given by *EvoC* is ordered according to the user's preferences, with the most preferred solutions appearing first followed by lesser preferred ones. Some of the solutions from Tables 3 and 4 were part of the Pareto optimal solution set identified from the branch-and-bound technique by D'Ambrosio and Hu in [4].

Table 4

Typical Solutions Found by *EvoC* When Higher Feasibility is Preferred

No.	Individual	Cost ($)	Feasibility Factor	Critical Excess MIPS
1	FP1, FP3 & FP4 on ASIC1, MP2, RAM1, ROM1	18.50	1.0	12.383
2	FP1, FP2 on PIO1-IO, MP2, RAM1, ROM1	17.00	1.0	12.081
3	FP1, FP3 & FP4 as custom circuits on MC4, MC4, ROM1	7.25	1.0	1.382
4	FP1, FP2, FP3 & FP4 on ASIC1, MC7, ROM1	6.00	1.0	0.583

Note: Remaining FPs are implemented in SW

Example II & III

The Example II is extracted from a multi-media application. The system receives, processes and transmits streams of data representing text, voice or images. Since the external links have restricted speeds, real-time constraints are imposed on some of the FPs. Other timing constraints are derived from performance requirements. Example III is merely an extension of Example II with the inclusion of communications specification.

Table 5 defines the FPs along with their respective *a, d, p* values and their components list. Some of the FPs involve simple input/output operations while others manipulate large image files (*e.g.,* JPEG). The CPs required to transfer data among the FPs are indicated in Table 6. Finally, Table 7 describes the component libraries. Static RAM (SRAM) is external to the processor and BUF represents an internal memory to a processor. SRAM can be attached only to external buses (Ext-8bit-B & Ext-16bit-B) and BUF only to internal bus (Int-32bit-B). The rest of the devices can be attached to any of the buses defined in Table 7.

Table 5

Set of FPs With (*a, d, p*) in Microseconds and Their Components List

Functional Process	Activation (a)	Deadline (d)	Period (p)	Components List
FP1	0	$2.2 x10^3$	$5.5 x10^2$	SM1, ASIC2
FP2	0	$1.1 x10^3$	$1.1 x10^3$	SM2, ASIC2
FP3	0	$5.5 x10^2$	$5.5 x10^2$	SM3
FP4	0	$5.0 x10^2$	$1.1 x10^3$	SM4
FP5	0	$2.5 x10^2$	$1.1 x10^3$	SM5, ASIC2
FP6	0	$2.5 x10^2$	$1.1 x10^3$	SM6
FP7	0	$1.0 x10^7$	$6.0 x10^8$	SM7
FP8	$1.0x10^7$	$3.0 x10^7$	$3.0 x10^7$	SM8
FP9	0	$6.0 x10^8$	$3.0 x10^7$	SM9
FP10	0	$2.5 x10^2$	$1.0 x10^3$	SM10
FP11	0	$1.0 x10^3$	$2.5 x10^2$	ASIC1

Since this system specification results in only a small number of configurations, we can identify all possibly feasible solutions (feasibility factor > 0.0) and they are listed in Table 8. All the solutions have a feasibility factor of 1.0. All the FPs other than FP1, FP2 and FP5 are implemented on the single component available for them in their component list. Note that the bus selected dictates the bus memory selected because of the restriction in the specification.

Table 6

Set of CPs

Communication Process	Sending FP (input)	Receiving FP (output)	NB (bytes)
CP1	FP1	-	64
CP2	-	FP2	64
CP3	-	FP3	64
CP4	FP2	-	64
CP5	FP4	-	64
CP6	FP5	FP6	64
CP7	-	FP7	128
CP8	-	FP8	1800
CP9	FP8	-	153600
CP10	-	FP9	360
CP11	-	FP9	2500
CP12	FP10	-	3
CP13	-	FP11	360

The Pareto optimal set of solutions from Table 8 were determined by a pair-wise comparison of all the solutions. They include all individuals with AAS & Ext-8bit-B and AAS & Ext-16bit-B selections; individuals with AAA & Int-32bit-B along with MP3, MP4 or MP5; individual with AAS & Int-8bit-B along with MP2. In fact, 40% of the entire set of feasible solutions are Pareto optimal. The lowest cost solution is the individual with AAS & Ext-8bit-B along with MP4. The lowest power solution is the individual with AAS & Ext-8bit-B along with MP3. The highest critical excess MIPS solution is the individual with AAS & Int-32bit-B along with MP2.

Table 7

Component Libraries and Their Characteristics

Library Name	Component	Cost ($)	Power (mW)	MIPS	BW	τ_I	τ_S	Instr.
SW_LIB	SM1							34000
	SM2							398000
	SM3							1224
	SM4							5431
	SM5							100
	SM6							304
	SM7							640000
	SM8							1488000
	SM9							21004800
	SM10							17
ASIC_LIB	ASIC1	5.5	300					
	ASIC2	5.5	300					
MICRO_LIB	MP1	10	500	10.56				
	MP2	44	4500	87.5				
	MP3	14	160	28.0				
	MP4	9	424	25.13				
	MP5	10	220	28.0				
MISC_LIB	SRAM	7	220			150	60	
	BUF	5	4			30	30	
	Ext-8bit-B	4	120		16.7			
	Ext-16bit-B	5	220		33.3			
	Int-32bit-B	8	400		66.7			

Note: BW in Mbytes/Sec; τ_I & τ_S in nanoseconds

Table 9 shows the 3 solutions obtained by *EvoC*, after running for 10 generations (Γ), when a lower cost is preferred. A population size (μ) of 10 was chosen for this example. The solutions differ only in their microprocessor selection. It also contains the best cost solution.

Table 8

Exhaustive List of Feasible Solutions for Example III

Individual	with MP3			with MP4			with MP5			with MP2		
	Δ_C (MIPS)	Cost ($)	Power (mW)	Δ_C (MIPS)	Cost ($)	Power (mW)	Δ_C (MIPS)	Cost ($)	Power (mW)	Δ_C (MIPS)	Cost ($)	Power (mW)
AAA, Bus8	14.11	34	1100	11.36	29	1364	14.11	30	1160	71.04	64	5440
AAA, Bus16	14.40	37	1200	11.63	32	1464	14.40	33	1260	71.97	67	5540
AAA, Bus32	14.87	38	1220	12.05	33	1484	14.87	34	1280	73.42	68	5560
AAS, Bus8	14.32	34	1100	11.53	29	1364	14.32	30	1160	72.09	64	5440
AAS, Bus16	14.52	37	1200	11.71	32	1464	14.52	33	1260	72.72	67	5540
AAS, Bus32	14.83	38	1220	11.99	33	1484	14.83	34	1280	73.70	68	5560
ASS, Bus8	-	-	-	-	-	-	-	-	-	46.34	64	5440
ASS, Bus16	-	-	-	-	-	-	-	-	-	49.70	67	5540
ASS, Bus32	-	-	-	-	-	-	-	-	-	47.26	68	5560
ASA, Bus8	-	-	-	-	-	-	-	-	-	46.29	64	5440
ASA, Bus16	-	-	-	-	-	-	-	-	-	46.66	67	5540
ASA, Bus32	-	-	-	-	-	-	-	-	-	47.24	68	5560

Note: `A' or `S', in the Individual, stands for ASIC or SW implementations of FP1, FP2 and FP5 respectively;
Bus8 is Ext-8bit-B; Bus16 is Ext-16bit-B; Bus32 is Int-32bit-B

Table 9

Typical Solutions Found by *EvoC* When Lower Cost is Preferred

No.	Individual	Cost	Power	Feasibility Factor	Critical Excess MIPS
1	AAS, MP4, Ext-8bit-B, SRAM	29	1364	1.0	11.535
2	AAS, MP5, Ext-8bit-B, SRAM	30	1160	1.0	14.322
3	AAS, MP3, Ext-8bit-B, SRAM	34	1100	1.0	14.322

Table 10 gives the solutions found by *EvoC*, after running for 10 generations, when lower power solutions yielding higher critical excess MIPS is preferred. The best power and MIPS solutions are found in the above table. All the individuals identified by *Evoc*, in Tables 9 & 10, are part of the Pareto optimal set of solutions.

Table 10

Typical Solutions Found by *EvoC* When Lower Power is Preferred

No.	Individual	Cost	Power	Feasibility Factor	Critical Excess MIPS
1	AAS, MP3, Ext-8bit-B, SRAM	34	1100	1.0	14.322
2	AAS, MP5, Ext-8bit-B, SRAM	30	1160	1.0	14.322
3	AAS, MP2, Ext-8bit-B, SRAM	64	5384	1.0	72.093
4	AAS, MP2, Int-32bit-B, BUF	68	5504	1.0	73.697

The following observations can be drawn from this example. Though FP1 can be implemented in either software or hardware, the SW component is never chosen as it demands a very high processing power (MIPS rate), which none of the microprocessors in the MICRO-LIB has. But, FP2 can be implemented in SW if MP2 is chosen for an individual. In such individuals, since the time-critical processes implemented in SW, especially FP2, consumes most of the processor's processing power, the value of critical excess MIPS becomes very low compared to when FP2 is implemented on ASIC2.

When comparing AAA and AAS configurations with external bus selections, it is interesting to note that moving FP5 from HW to SW results in an increased value of the critical excess MIPS, which is somewhat contra-intuitive. This fact is due to the additional communication overhead (CP6) incurred in all AAA configurations. To illustrate this, consider two configurations, individual A (having AAA, MP3, Ext-8bit-B, SRAM) and individual B (having AAS, MP3, Ext-8bit-B, SRAM). *NC* for CP6 can be calculated, from Equation (7) as follows:

$$NC = \left(\frac{64}{16.7} + 0.15 + (1 - 64) * 0.06 \right) * 28 = 217$$

This gives rise to 217 additional instructions for FP6 (implemented in SW) in individual A, compared to individual B which implements FP5 in SW that has only 100 instructions. If communications were neglected for Example III then all the final solutions would be of type AAA .

The difference in the critical excess MIPS values between AAA and AAS individuals decreases with the selection of a bus having a higher bandwidth. This is because, as the bandwidth of the bus increases the overall communication overhead in the system decreases.

It could be noted from Tables 9 & 10 that the bus Ext-8bit-B is chosen predominantly over the other types, because of its lower cost and power while still being able to yield feasible solutions.

Timing Analysis

Table 11 gives the comparison of the run time of the three examples based on the average of several runs for each on the latest version of the *EvoC*. Example I is a more complex design problem compared to the other examples. The running time of *EvoC* depends on many factors, including the complexity of system specification (such as number of FPs and CPs), number of available components, population size and number of generations.

Two of the factors that contribute to the relative long execution time of *EvoC* were identified. The lp_solve (ver. 2.0) software was invoked through system calls in the earlier versions of *EvoC*. In Unix shell, system calls are handled as child processes and the main program as the parent process. The system call interface amounted to unacceptably large execution times. On an average (based on the examples used) parent process consumed only about 10-15% of the total program execution time.

Table 11

EvoC Run Time for the Examples

Design Problems	Average Execution Time per Generation in Seconds	Population Size
Example I	30	45
Example II	1.7	10
Example III	4.6	10

In the earlier ranking scheme given by Goldberg [8] (during Selection) used in *EvoC*, the set of all preferred individuals in the current population receives the highest rank and are removed from further contention. The remaining individuals then compete for the next highest rank. The above procedure is repeated until all individuals are ranked. This procedure involves many comparisons in every iteration to determine winners (preferred solutions) among a set of individuals.

We implemented heapsort algorithm for ranking [refer to Chapter IV] and procedural call interface to the lp_solve software to reduce the execution time of *EvoC*. The former speeds up the program execution by reducing the number of calls made to the lp_solve software during the evaluation process in EA and the later by eliminating the system calls. Table 12 summarizes the speed-up values gained through the above methods for Example I and II. The speed-up value is calculated from the total execution time of a run.

Table 12

Improved Execution Time of *EvoC*

Design Problem	Preferred Attribute	Speed-up with Heapsort Algorithm	Speed-up with Proc. Call Interface
Example I	Feasibility	3.7	34.0
	Cost	1.0	5.0
Example II	Power	1.1	6.7
	Cost	0.8	5.2

There is one case in Table 12 that has a speed-up of 0.8. Speed-up due to heapsort algorithm is very much problem dependent and the number of individuals per population also influences its value. The speed-up value less than 1.0 is mainly due to the overhead involved with the heapsort algorithm itself which becomes significant with a very small population size. This is proved by an increase in the execution time of the parent process when implementing heapsort algorithm.

Finally, software implementation of EA algorithm demands a good random number generator. We use a *linear congruential pseudo-random number generator* that generates uniformly distributed random numbers.

CHAPTER VII

CONCLUSIONS AND FURTHER WORK

We presented the CAD tool, *EvoC* used for the automated hardware/software partitioning of RTES at the system level. The highlights of our approach can be summarized as follows:

1. A high level of abstraction, system level is used. It helps us to efficiently explore many design alternatives and analyze the tradeoffs. 2. There are no existing CAD tool which combines EA and ISMAUT to solve a RTES hardware/software partitioning problem. The combination of EA and ISMAUT gives *EvoC* the capability of exploring a large design space corresponding to a multiple objective optimization problem.

3. *EvoC* targets its solution set to be a subset of the Pareto optimal set [14] and will vary with user's preferences.

4. The modeling of communication does not merely result in adding components to the system, but rather expresses the behavior of the system as an integrated set of components – the compatibilities and tradeoffs between a connected set of components.

5. *EvoC* has an input format that forms a generic platform for specifying a variety of RTES.

6. *EvoC* assumes no target architecture (for the monoprocessor case) and is flexible enough to produce design alternatives that are drastically different in their system architecture.

Though the current version of the tool can handle all the aforementioned features, further improvements can be made to enhance this tool. Some of them are given here.

A characteristic *latency* for ASICs may be introduced. The latency of an ASIC represents the delay in the hardware circuitry. It can be used during selection of components in the first phase of partitioning to form a constraint as follows:

$$\text{Constraint\_}\textit{pass}\text{ condition:}\qquad \text{latency} + (NC_i \;/\; M) < d_i \qquad (8)$$

where, *NC* is the communications overhead and *M* is the processor MIPS rate. An allocation of a FP to an ASIC is acceptable only if the deadline (d_i) of the FP is greater than the sum of the latency of ASIC and the communications overhead (expressed in time units) incurred by the FP. The above constraint will serve as a check for HW components on their "load", due to the implementation of one or more UDPs, just like the system feasibility check for SW components. Added to this is the effect of communications overhead, that is associated with the FPs implemented on the ASIC.

While accounting for communication overhead in the system attributes, bus arbitration time cannot be ignored especially if buses are heavily utilized. *EvoC* does not presently include any bus arbitration overhead although a worst-case average bus bandwidth requirement serves as a constraint used during bus selection.

In the present form of the tool, multi-processor implementations are not considered. Hence, the next natural step for extending the features of *EvoC* would be to include multi-processor cases. The data structure has been designed to be flexible enough to handle such cases. For example, there is a provision in the data structure for specifying multiple SW components (that can be implemented on different processors) for each user-defined processes.

Appendix A

Input File Format of *EvoC*

Once the input files are formed, *EvoC* can be executed using the command "evoc". The user will then be prompted to enter the number of generations. The output file "results" will be created on successful completion of a run. It contains all the final design configurations, explicitly showing the hardware/software allocations of all user-defined system processes and the component interconnections *via* buses. It also gives the total system cost, critical excess MIPS available, feasibility factor and other additive attributes specified by the user.

EvoC input file format is given in the following page. The words in lower case are keywords and the words in upper case are variable names.

```
// attributes/characteristics in attr.dat input file

ATTRIBUTENAME   (instructions, cost, mips, power, area, activation,
                 period, deadline, feasibility, xbytes, bandwidth,..)
attribute-oper-type
ATTRTYPE   (additive, feasibility-factor, mips-factor,...)
attribute-goal-type
ATTRGOAL   (minimize, maximize)
attribute-value-type
ATTRVAL         (real, integer,long, ...)
attribute-util-type
ATTRUTIL   (linear, ...)    // lower case are keywords
end
```

```
// components in sw.dat, asic.dat, pw.dat, micro.dat & misc.dat input
   files

COMPNAME
attributes
ATTRBUTENAME    VALUE
...
ATTRBUTENAME    VALUE
implemented-parts
PARTNAME1 (ram, rom, bus, ...)  NUM
...
PARTNAMEn                NUM
required-parts
PARTNAME1                NUM
...
PARTNAMEn                NUM
end
```

```
// functions or processes in func.dat input file

FUNCTIONNAME
attributes
ATTRBUTENAME    VALUE
...
ATTRBUTENAME    VALUE
components
COMPNAME1       LIBNAME (sw, asic, pw, micro, misc)
...
COMPNAMEn       LIBNAME
end
```

```
// communication in comm.dat input file

COMMNNAME
attributes
ATTRBUTENAME    VALUE
...
ATTRBUTENAME    VALUE
input
FUNCTIONNAME
output
FUNCTIONNAME
end
```

Appendix B

Data Structure of *EvoC*

```
class Attr_Type
{
public:
     Attr_Type ();
     char         name[MAX_LINE_SIZE];   // generic name e.g. cost
     int          oper_type;
     int          goal_type;
     int          val_type;
     int          util_type;
}attr_type[MAX_ATTR];


class Attr_Value
{
public:
     Attr_Value ();
     char         name[MAX_LINE_SIZE];   // specific name e.g. ProcCost
     long         index;                 // unique ID
     Attr_Type*type;
     float             value;
}attr_value[20*MAX_AVAL*MU_LAMBDA+1];


class Comp_subModule                     // all Cells
{
public:
     Comp_subModule ();
     char              name[MAX_LINE_SIZE];
} comp_submodule[MAX_PARTS];


class Comp_Module                        // all COMPONENTs
{
public:
     Comp_Module ();
     char                   name[MAX_LINE_SIZE];
     Attr_Value             *attr_list[MAX_ATTR];
     Comp_subModule         *imp_parts[MAX_PARTS];
     long                   num_imp_parts[MAX_PARTS];
     Comp_subModule         *req_parts[MAX_PARTS];
     long                   num_req_parts[MAX_PARTS];
} sw_lib[MAX_SW], asic_lib[MAX_ASIC], pw_lib[MAX_PW],
  micro_lib[MAX_MICRO], misc_lib[MAX_MISC];


class Func_Spec
{
public:
     Func_Spec ();
     char              name[MAX_LINE_SIZE];
     Attr_Value        *attr_list[MAX_ATTR];
     Comp_Module       *allsw_imp_func[MAX_IMP_FUNC],
                       *allasic_imp_func[MAX_IMP_FUNC],
                       *allpw_imp_func[MAX_IMP_FUNC],
                       *allmicro_imp_func[MAX_IMP_FUNC];
} func_spec[MAX_FUNC];
```

```
class Func_Imp
{
public:
     Func_Imp ();
     char           name[MAX_LINE_SIZE];
     int            type;     // SW=1, ASIC=2, PW=3, MICRO=4, MISC=5
     Comp_Module    *pos_imp_func;
} func_imp[MAX_FUNC*2*MU_LAMBDA+1], func_impl[MAX_FUNC*MU_LAMBDA+1];


class Comm_Spec
{
public:
     Comm_Spec ();
     char           name[MAX_LINE_SIZE];
     Attr_Value     *attr_list[MAX_ATTR];
     Func_Spec      *input;
     Func_Spec      *output;
} comm_spec[MAX_COMM];


class Comm_Imp
{
public:
     Comm_Imp ();
     char           name[MAX_LINE_SIZE];
     Comp_Module    *pos_imp_comm[MAX_COMP_COMM];
} comm_imp[MAX_COMM*2*MU_LAMBDA+1], comm_impl[MAX_COMM*MU_LAMBDA+1];


class Bus_Comp
{
public:
    Bus_Comp ();
    Comp_Module    *BusPtr;
    Comp_Module    *BusCompPtr[MAX_COMP_BUS];
    Comp_Module    *BusMem[MAX_COMP_BUS];
    Comm_Imp       *BusCpPtr[MAX_COMM];
} bus_comp[MAX_BUS*2*MU_LAMBDA+1], bus_compl[MAX_BUS*MU_LAMBDA+1];


class Individual
{
public:
     Individual ();
     Attr_Value     *attr_list[MAX_ATTR];
     Func_Imp       *func_config;
     Comm_Imp       *comm_config;
} alternative[2*MU_LAMBDA+1], temp_alt [MU_LAMBDA+1];
```

BIBLIOGRAPHY

[1] S. Bollinger and S. Midkiff, "Heuristic techniques for processor and link assignments in multicomputers", *IEEE Trans. on Computers*, Vol 40, No. 3, Mar 1991.

[2] B. Braschi, A. Ferreira and J. Zerovnik, "On the behavior of parallel simulated annealing", in *Parallel Comp. 89*, D. Evans, G. Joubert and F. Peters (edt.), Elsevier Science Publishers B.V. (North-Holland), '90.

[3] K. Buchenrieder and C. Veith, "CODES: A practical concurrent design environment", *International Workshop on Hardware-Software Codesign*, October 1992.

[4] J.G. D'Ambrosio and X. Hu, "Configuration-level hardware/software partition for real-time embedded systems", *Proceedings of the Third International Workshop on Hardware-Software Co-Design*, September 1994, pp. 34-41.

[5] P. Eles, Z. Peng and A. Doboli, "VHDL system-level specification and partitioning in a hardware/software co-synthesis environment", *Proceedings of the Third International Workshop on Hardware-Software Codesign,* Sept. 1994, pp.49-55.

[6] R. Ernst, J. Henkel and T. Benner, "Hardware-software cosynthesis for microcontrollers", *IEEE Design & Test of Comp.*, Vol 10, no. 4, December 1993, pp. 64-75.

[7] D. Fogel, *Evolutionary Computation,* IEEE Press, 1995.

[8] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley Pub. Co., 1989.

[9] G. Greenwood, X. Hu and J. D'Ambrosio, "Fitness Functions for Multipleobjective Optimzation Problems: Combining Preferences with Pareto Rankings", *Foundations of Genetic Algorithms,* R. Belew and M.Vose (Eds.), Morgan-Kaufmann, San Mateo, CA (accepted, in press)

[10] G.K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems", *IEEE Design & Test of Comp.*, Vol 10, no. 3, September 1993, pp. 29-40.

[11] J. Henkel, T. Benner, R. Ernst, W. Ye, N. Serafimov and G. Glawe, "COSYMA: A software-oriented approach to hardware/software codesign", *J. of Comp. & Software Engineering*, 2(3), '94, pp. 293-314

[12] J. Horn and N. Nafpliotis, "Multiobjective optimization using the niched pareto genetic algorithm", *IlliGAL Report No. 93005*, July 1993, U. of Illinois at Urbana-Champaign.

[13] X. Hu, J.G. D'Ambrosio, B.T. Murray and D. Tang, "Codesign of Architecture for Automotive Powertrain Modules", IEEE *Micro,* August 1994, pp. 17-25.

[14] J.G. D'Ambrosio, X. Hu and G. Greenwood, "An evolutionary approach to configuration-level hardware/software partitioning", *PPSN IV*, July 1996, pp. 900-909.

[15] T.B. Ismail, M. Abid and A. Jerraya, "COSMOS: A codesign approach for communicating systems", *Proc. of the Third Int'l Workshop on Hardware-Software Codesign*, September 1994, pp. 17-24.

[16] R.L. Keeney and H. Raiffa, *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*, John Wiley & Sons, NY, 1976.

[17] S. Kumar, J.H. Aylor, B.W. Johnson and W.A. Wulf, "Exploring hardware/software abstractions & alternatives for codesign", *The Second International Workshop on Hardware-Software Codesign*, Cambridge, Massachusetts, October 1993.

[18] S. Kumar, J.H. Aylor, B.W. Johnson and W.A.Wulf, "Object-oriented techniques in hardware design", *Computer*, Vol 27(6), 1994, pp. 64-70

[19] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment", *J. of the Association for Computing Machinery*, Vol. 20(1), 1973, pp. 46-61.

[20] M.B. Srivastava and R.W. Brodersen, "Rapid-prototyping of hardware and software in a unified framework", *Proc. of Int'l Conf. on CAD*, November 1991, pp. 152-155.

[21] D.E. Thomas, J.K. Adams and H.Schmitt, "A model and methodology for hardware-software codesign", *IEEE Design & Test of Comp.,* Vol 10, no. 3, Sept. 1993, pp. 6-15.

[22] T.Y. Yen and W.W. Wolf, "Communication synthesis for distributed embedded systems", *Proc. of IEEE Int'l Conf. on CAD*, Vol 28, Nov. 1995, pp. 288-94.

[23] W.W. Wolf, "Hardware-software codesign of embedded systems", *Proc. of the IEEE,* 82(7), July 1994, pp. 967-989.

[25] S.Yalamanchili, L.T. Winkel, D. Perschbacher and B. Shenoy, "Genie: An environment for partitioning and mapping in embedded multiprocessors", *IEEE Symposium on Parallel and Distributed Processing,* Vol 18, Dec. 1993, pp. 522-9.

[25] S. Yerramareddy and C.Y. Lu, "Hierarchical and interactive decision refinement methodology for engineering design", *Research in Engineering Design,* Vol. 4, 1993, pp. 227-239.