
# ESE 566: HARDWARE-SOFTWARE CODESIGN OF EMBEDDED SYSTEMS

# PROJECT 2:
# REVIEW OF RESEARCH PAPERS AND IMPLEMENTATION

GUIDED BY:
Dr. ALEX DOBOLI

PREPARED BY:

AISHWARYA GANDHI (111424452)
KEWAL RAUL (111688087)
SHASHANK RAO (111471609)

# 1. REPORTS SUMMARY

Summary of paper 1: Hardware/Software Co-Design

Hardware/software co-design means meeting system-level objectives by exploiting the synergism of hardware & software through their concurrent design. This paper introduces various aspects of co-design by highlighting the co-design issues & their relationship to classical system implementation tasks. The paper develops a perspective on modern digital system design that relies on computer-aided design tools and methods. Majority of digital systems are programmable & thus consisting of hardware and software components. Traditionally a hardware circuit used to be configured at manufacturing time, but the introduction of FPGA technology has brought reconfigurability to hardware circuit. Functions of a hardware circuit can be chosen by the executing program whereas the program can be modified at run-time.

The paper associates the co-design problems with the classes of digital systems arise from & characterizes them based on general criteria such as domain of application, degree of programmability & implementation features. There are two important issues related to programming – who has access to programming the system & the technological levels at which programming is performed. The application developer requires systems to be retargetable, to port a given application across multiple hardware platforms. Digital systems can be programmed at different level such as the application, instruction or hardware levels. Most digital systems use components with an instruction set architectures(ISA). Instruction-level programming is achieved by executing the instructions supported by the architecture on the hardware. Hardware-level programming means reconfiguring the hardware in a desired way. FPGAs allow to configure the interconnections among logic blocks to determine their personality. System-level field programmability can be achieved by storing programs in read/write memories & exploiting the programmable interconnections.

The paper discusses general co-design problems & approaches. Embedded systems are elements of larger systems. Embedded control systems usually regulate mechanical components via actuators and receive input data provided by sensors. Control functions can be implemented both in hardware and in software, so specific design disciplines must be used to ensure. The concept of ISA plays a fundamental role in digital design. The primary goal of co-design in ISP development is to optimize utilization of the underlying hardware. Modern general-purpose processors exploit deep pipelines, concurrency & memory hierarchies. Hardware/software trade-off is possible in pipeline control & cache management mechanisms. Reconfigurable systems exploit FPGA technology, so that they can be personalized after manufacturing to fit a specific application. The major hardware/software co-design problems consist of identifying the critical segments of the software programs & compiling them efficiently to run on the programmable hardware.

The design of hardware/software systems involves modelling, validation & implementation. A hardware/software partition problem is finding those parts of the model best implemented in hardware and those best implemented in software. Partitioning can be decided by the designer, with a successive refinement and annotation of the initial model or determined by a CAD tool. Scheduling solutions based on heuristics techniques are implemented schedule partitioned hardware/software tasks. Thus the paper does an extensive study of the research & approaches in hardware/software co-design of digital systems.

Summary of paper 2: Hardware-software Codesign of Multimedia Embedded Systems: the PeaCE Approach

A codesign environment is a software tool that facilitates capabilities to solve various design problems including system specification, design space exploration, hardware/software co-verification & system synthesis. The paper presents a codesign environment PeaCE that mainly targets multimedia applications with real-time constraints. Increasing complexity & the extreme time-to-market pressure increases the challenges of designing multimedia embedded systems. The growth rate of conventional design method is far below of the system complexity of today's embedded systems & thus HW/SW codesigns has emerged as a new design methodology.

The paper defines PeaCE as a full-fledged codesign environment that provides seamless codesign flow from functional simulation to system synthesis. The HW/SW codesign starts with initial system specifications using models of computation that express concurrency naturally instead of sequential C code description. The flow of proposed codesign methodology begins with a data flow model for computation and an FSM model for control module of the system. In the next step PeaCE maps the system behaviour optimally to the hardware architecture by performing HW/SW partitioning & component selection.

An embedded system that supports multiple applications by dynamically reconfiguring the system functionality is called a multi-mode multimedia terminal (MMMT). Tasks in MMMT system have diverse activation conditions and port semantics need to be clearly specified in the task-level specification, thus the proposed system has devised a novel task-specification model. The layer software structure supports easy reconfiguration. Each task is created as a separate thread by the OS wrapper. After component selection & mapping the code is synthesized for each component. HW/SW co-simulation is performed at the instruction level to obtain the memory trace information from all the processing components. The paper efficiently provides experimental results of real examples to the viability of their proposed technique

Summary of paper 3: An Overview of Formal Hardware Specification language

The paper introduces the field of verification by surveying formal specification languages. Verification has been one of the most difficult aspect of hardware designing. It is highly important that a design is not sent into production unless thoroughly verified. A single bug can lead to staggering loss for a company. The paper proposes that an ideal language should have precise semantics, easier to learn, should be compatible with automatic verification techniques & also should easily integrate with current industrial design practice. The paper categorizes the languages into hardware languages, programming languages & Visual languages.

Hardware language category focuses on hardware monitors & Objective VHDL. Monitors possess capabilities useful for protocol design & verification. Monitor specification are written similar to a hardware description language, so no new language is needed to learn as well as it can get fully integrated into the design cycle. The drawback of monitors specification is it being written in HDL makes it difficult to parse for humans. Object VHDL raises the abstraction level at which design activities occur, enhancing design reuse. Object VHDL provides type classes for data abstraction & entity classes for structural abstraction

The paper includes SpecC & java in their discussion of programming languages for hardware specification. SpecC is a formal, executable modelling language with an accompanying tool suite to facilitate hardware-software codesign. SpecC shares a similar syntax to C programming language. SpecC tools provide static analysis & estimation to ensure that design metrics remain within specified bounds. Java is presented as hardware/software codesign platform in the hardware specification arena. Advantage of using Java for both hardware description & software programs allows entire system to be simulated at once. Java is easy to learn but is of limited use for compliance verification. The paper concludes SpecC to be the most promising of the textual languages considered for formal compliance verification.

The paper further discusses three languages which aid software engineering: Statecharts, Message Sequence Charts, Specification & Description Language. Statecharts extend state machine notation by adding mechanism for expressing hierarchy, concurrency & communication. Designers usually use state machines for modelling their design thus, statecharts are easily learned & integrated into industry. Message Sequence Charts (MSC) represent hardware communication intuitively which makes it easy to understand & are already been used in some informal protocol specification. Specification & Description Language are similar in notations and usage to the statecharts. It has a formal definition & hence is typically used for mission-critical systems, such as aerospace and complex real-time systems. They conclude that each of the above software originated languages fall short to their expectations.

In the Hardware Engineering perspective, Heterogeneous Hardware Logic is considered. This logic included diagrammatic notations for circuit diagrams, algorithmic state machines & timing

diagrams. It does not require any special training & is already used in industrial practice. Its only drawback is that it's better suited to human intensive verification techniques compared to automatic ones. Lastly Live Sequence charts (LSCs) language is discussed which is an extension to the Message Sequence Chart. They provide an equivalent to if-then-else mechanism. The main weakness of the language is its lack of timing model.

Summary of paper 4: System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search

The paper presents two heuristic algorithms for automatic hardware/software partitioning of system level specification. Most of the embedded systems have high performance requirements which can only be met by implementing certain components directly on hardware. This is usually achieved using ASICs or FPGAs. The Hardware/Software co-design term denotes the cooperative development process of both hardware and software components of a system. The paper presents an approach which tries to maximize performance under given hardware and software cost constraint. The proposed design environment accepts a system level, implementation independent specification as an input. The partitioning strategy is based on metric values derived from profiling, static analysis of the specification and cost estimation. The paper implements a simulated annealing based algorithm & a Tabu search algorithm for hardware/software partitioning.

Partitioning starts with an initial system specification described as a set of processes interacting through communicating channels. The partitioning algorithms outputs a models two sets of interacting processes marked as hardware candidates & software candidates. The partition algorithm takes into account majorly two types of statistics: computational load (CL) which is a quantitative measure of the total computation executed by a region and communication intensity (CI) is the total number of send operations executed on the respective channel.

The paper breaks the entire partitioning into 4 major steps. During the first step processes are individually examined to identify performance critical regions which mostly include loops and subprograms. Secondly a process graph is generated with processes represented as weighted nodes. The partitioning algorithm takes into account the weights associated to each node & edge. The node weights determine the degree to which it is suitable for hardware implementation, edge weights measure communication & mutual synchronization between processes. The partitioning of the graph is presented by comparative study & implementation of the simulated annealing & tabu search algorithm. Simulated annealing selects the neighbouring solution randomly and always accepts an improved solution while the tabu search approach accepts uphill movement and simulates convergence towards a global optimum by creating and exploiting data structures.

The paper does a comparative implementation of the both the algorithms by implementing on a couple of real world examples. Finally the paper concludes that TS performed superior to SA.

Summary of paper 5: Partitioning Decision Process for Embedded Hardware and Software Deployment

The paper presents a design process for partitioning application units into hardware and software based on Multiple Criteria Decision Analysis (MCDA). The paper suggests pushing partitioning decisions to the later phases of system design. The arguments provided by the authors to support this methodology is that if the application is partitioned at earlier phase of design, it would lead to different modules being interleaved and would require multiple modifications/iterations to obtain the optimum solution. Partitioning at an earlier stage is also not recommended since hardware design does not take into account the computational power required by the software, as well as does not allow software design to exploit the available hardware resources, thus requiring multiple iterations with different combinations to obtain the optimum solution.

In order to perform the partitioning, the paper describes a formal definition of the embedded system as a Component-based system represented by - a set of components (hardware and software) comprising the application, a set of bindings between the components and a platform to deploy these components. These components and their properties (obtained from component libraries possessing info regarding different interfaces and extra-functional properties for different components possible to be used) are included in a partitioning decision table. There are a number of activities for enabling a systematic partitioning process - (i) modeling of application as a set of components, (ii) identification of overall application and project constraints required to arrive at the decision criteria, (iii) identification of project-related and application-related properties, (iv) filtering out the variants that do not satisfy constraints and assigning priorities on different properties of components using weights, (v) performing partitioning using MCDA methods, (vi) applying suitable MCDA methods for ranking solutions in case of multiple solutions.

The paper also presents an industrial case study in order to validate the research work. The proposed methodology is applied to develop and deploy a wind turbine control application prototype. The paper provides an overview of how the system is designed and created using the techniques discussed in the paper. The paper concludes with related works and future work related to the methodology described in the paper.

The paper, while giving an extensive overview of the partitioning techniques, does not go in-depth into any of the methods which could be used for implementing the methodology. The paper also considers that the partitioning decisions will be taken according to MCDA techniques, without actually performing a systematic review of these techniques and tools used to implement these techniques. The methodology also requires the component libraries to be available with the designer in order apply MCDA techniques; this may not be possible in all cases.

Summary of paper 6: Online Hardware/Software Partitioning in Networked Embedded Systems

This paper investigates an iterative discrete algorithm which can be applied to distributed embedded systems with time-varying demands. It makes use of reconfigurable hardware/software nodes to balance the load of the partition at run-time, i.e., when there are changing computational demands, the system will dynamically assign tasks to the reconfigurable nodes and move tasks between nodes if required. It is able to do so by using an Evolutionary Algorithm combined with a discrete version of Diffusion Algorithm.

The paper considers embedded systems consisting of networked hardware/software reconfigurable nodes having different properties such as interconnect (links between nodes), embedded (features including power, latency and area) as well as the nodes themselves. The paper then goes on to explain the concepts in further depth and also gives two formal definitions, one for the temporal partitions and the other for workload characterization. Temporal partition at a point of time is defined as the assignment of each task (process) to a resource (node), as well as the indication whether the task is implemented in hardware or software. Workload characterization for each task has been defined using weights; these weights are calculated as fraction of required area to maximal available area in case of a hardware load and as fraction of execution time to period for a software load. The paper then goes on to explain the basic diffusion algorithm, which states that amount of loads on a node are are calculated iteratively using weights and previous iteration loads. The paper then explains the online hardware/software partitioning envisioned by the authors, which applies the evolutionary algorithm and the discrete diffusion algorithm. The paper gives multiple theorems and their proofs to compare the behavior of the discrete diffusion algorithm with its continuous counterpart. The paper then concludes with a few experimental results (with regards to network congestion and relative deviation) calculated for different types of networks like meshes with 3x3 or 4x4 nodes and ring and chordal ring with 8 nodes, as well as future work planned by the authors.

While the paper provides an excellent validation of its proposition, it fails to explain in-depth details regarding most of its implementation. There is no explanation offered as to how the reconfigurable nodes will be created/implemented as well as how tasks are to be linked to these nodes. However, the paper provides a good starting point for implementing a feasible solution based on the techniques thought of by the authors.

Summary of paper 7: Configuration-Level Hardware/Software Partitioning for Real-Time Embedded Systems

This paper provides a methodology for hardware/software partitioning at the configuration level, where hardware are modeled as resources with no detailed functionalities and software are modeled as tasks utilizing these hardware resources. Analysis at this level of abstraction allow cost and performance tradeoffs to be performed at an earlier phase of the design process and also

provides a large design space to be explored by the designer. According to the paper, the goal of the configuration-level design is to determine the overall system architecture, particularly for real-time embedded systems.

The paper proposes approaching the partitioning problem as follows - a system is specified by a set of time-critical functions and constraints associated with each function, such as timing. These function evaluations can be carried out dedicated hardware circuits or by executing software tasks on processors. It makes use of an existing library of different processors and hardware components to be used for such a purpose. Within these libraries, each processor or hardware component can be characterized by attributes such as cost and power, out of which some have to be selected for optimized design. In this paper, the authors make use of Global Optimal Part Selection (GOPS) tool to accomplish this global optimization. GOPS finds the Pareto-optimal set of part-set solutions; GOPS enumerates the Pareto-optimal set by performing a multi-attribute branch-and-bound search of possible functions.

For a real-time system, each time-critical function is specified using activation times, deadline constraints to complete the task, and the periodic time intervals at which the evaluation of the functions are to be repeated. A metric known as feasibility factor is used to address the problem in configuration-level design, which indicates the possibility of a system being feasible. The paper then provides two lemmas and a theorem in order to help the reader better understand as well as calculate the feasibility factor. The paper also presents an application example (using nine time-critical functions to be implemented in a single processor system) to help understand the partitioning methodology. The example applies all aspects of the techniques proposed in the paper to obtain the optimized design solution. The paper ends with a brief summary and discussion with regards to future work.

The paper presents a well-rounded and in-depth explanation of the methodology and techniques proposed by the authors, including necessary mathematical proofs, succinct explanations as well as an application example which helps the reader in better understanding the proposed methodology. There are also clear explanations regarding the working of the design tool like GOPS and libraries which are used to implement hardware/software partitioning.

Summary of paper 8: Dynamic Hardware/Software Partitioning: A First Approach

The paper presents a first approach to dynamic hardware/software partitioning of system architecture and initial on-chip tools for a simplified configurable logic fabric. Dynamic partitioning has advantages over traditional partitioning approaches, in that it is more transparent, meaning that the designer can achieve the benefits of partitioning while writing regular software and using standard software tools and flows. According to the paper, an ideal dynamic partitioning technique would monitor a microprocessor's executing binary program, detect critical code regions, decompile these regions, synthesize them to hardware, place and route the hardware onto on-chip

reconfigurable logic, and update the binary to communicate with the logic. Since dynamic partitioning deals with small regions of code, implementing synthesis and place-and-route tools on-chip will not use too much time and memory, and are indeed feasible.

The paper then presents the overall architecture for dynamic hardware/software partitioning. The architecture consists of standard embedded microprocessor and memory for normal application software execution. It also consists of a dynamic partitioning module as well as a configurable logic module. The dynamic partitioning module finds the most frequently executed software regions and re-implements these regions as hardware on the configurable logic modules. It is able to do so by using a separate partitioning co-processor and memory to run a program that will decompile and synthesize selected binary regions for hardware implementation. The Configurable Logic Fabric (CLF) module is made very simple since it will mainly be used to handle smaller and simpler inner loops of software regions. In the paper, the CLF module is implemented as a matrix of simple 3-input 2-output look-up tables (LUTs) surrounded by switch matrixes (SMs) for routing.

The paper then proceeds to explain an overview of the working of the tool. The first component explained is the loop profiler, which detects regions of software which can be implemented as hardware. The next component is the decompilation unit which converts software loops into a high-level representation which is more suitable for synthesis. The component first converts each assembly instruction into equivalent register transfers; followed by building of control flow graphs for the software region and a data flow graph by parsing semantic strings for each register transfer. The next component brought into work in the tool is the DMA configuration module which maps memory accesses of decompiled loop onto the DMA architecture. Following this, Register-Transfer (RT) synthesis converts each output bit into a boolean expression by traversing the data flow graphs of the software region. Logic synthesis creates a directed acyclic graph (DAG) of the boolean logic network created in the previous step. After this, technology mapping, place and route modules convert the boolean expression into a netlist. Technology mapping traverses the DAG backwards starting with the output nodes and combines nodes to create LUT nodes corresponding with 3-input single-output LUTs. The LUT nodes are then placed onto the configurable logic fabric followed by routing between inputs, outputs and LUTs using a simple greedy algorithm. Following this Bitfile creation module combines the placed and routed hardware description with the DMA configuration information into a single bitfile that can be used to initialize the configurable logic. Binary modification module handles updating the software binary to utilize the hardware for loops. Original software instructions for the loop are replaced with a jump to hardware initialization code.

The paper then presents experiments performed by the authors validating their propositions. Average speedup due to the dynamic partitioning was 2.6 which was close to the ideal speedup of 2.8. The paper presented the ideas in a clear and explained topics in a detailed and in-depth way, which helps to reinforce the authors' proposals.

Summary of paper 9: Embedded Software in Real-Time Signal Processing Systems: Design Technologies

This paper mainly concentrates on the lack of software compilation techniques for the present embedded processors. Compilation techniques are well defined for the traditional general purpose processors which had a regular architecture. But, the advanced processors like the fixed-point DSPs and the ASIPs which have several architectural features are not utilized by their software compilers to their fullest. This way, a bottleneck is created in the embedded system design process by the software design phase. But, shifting design from hardware to software increases the flexibility of design and thus it is important to advance in software compilation techniques along with hardware advancement.

Two main aspects which are taken into consideration are architectural re-targetability which means the compilations tools should work for different processors like GPP, DSP, ASIP & Parameterizable processors as they are constantly changing. Second is the code quality which means the compiled code should utilize all the features in these new processors effectively. It discusses a classification scheme to characterize a given compiler depending upon the architectures it can handle and characterize a given processor based on parameters such as data type, code type, instruction format, memory & register structure, control flow, etc. This scheme can then be used to characterize a compiler and find out how much this compiler can be used for further designs. Also it can be used to know the problems in the compiler development for particular DSP and ASIP.

Further, the paper discusses the issues in software compilation and how the code generated by available compilers for advanced DSPs can't be used for industrial purpose instead the design team had to manually perform the assembly coding. To resolve these issues, more phases and phase coupling is added in code generation compared to code selection, register allocation & scheduling phases of traditional compilation process. Also, specialised compiler algorithms are developed that have larger compilation time but it is tolerated for these embedded processors. A survey is shown of existing compilation techniques based on which the complex techniques can be developed with focus on re-targetable software compilation.

Summary of paper 10: Hardware-Software Co-Design of Embedded Systems

This paper states that even though there is a tremendous improvements made in designing of Hardware and Software separately, it is difficult to achieve the same performance, cost, & reliability requirements for a hardware software co-design. It discusses about the difficulties faced in this design at different levels of abstraction in the co-design. It describes a model of an embedded system which involves selection of a hardware engine upon which the software architecture is built simultaneously and not one after the other as it is not possible to choose a hardware engine without knowing the software processes run on it. It describes how a spiral development model works based on the system specifications in which refinements are made to the architecture after

the initial design until the requirements are satisfied. It does a performance analysis of the system as a whole as well as the CPU and Software performance individually.

It discusses about the mapping of hardware and software with an example of process data flow graph for software which states the control flow between the processes and a processor flow graph for hardware which states the linkage of CPU nodes and the mapping of the two. It discusses different techniques of hardware software partitioning. APARTY system partitioning tool uses clustering in different stages. Depending on the objective, it clusters nodes stage by stage until the required chip area is achieved. It discusses different forms of partitioning algorithms like migration of operations from hardware partitioning to software partitioning to satisfy the CPU, rate and bus utilization constraints. In another one, only those parts which do not satisfy the requirement in software are moved to hardware and speedup is estimated. If speedup is not increased due to reasons like dependency in pipeline, it is shifted back to software. It also describes a method of separating ASIC in catalog and custom ASIC where straightforward functions are put in the catalog ASIC and custom ASIC has application specific.

Further, it discusses how poor process partitioning affects the implementation cost of a software architecture if one process is delayed it causes another processor to be ideal which in fact affects the performance. Also, it describes the two ways in which the system performance is affected in a distributed process allocation. Firstly, by reducing the cost of intercommunication of processes by allocating the processes which need to talk to one another frequently to a single hardware engine. This allocation of processes is faster and cheaper. Secondly, by changing how tasks sharing a single hardware engine are scheduled.

# 3. SOFTWARE IMPLEMENTATION OF PAPER

We have selected to implement the following paper - "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search". We have provided a functional overview in the previous sections. This section will deal with the technical aspects related to implementing the paper. We have used C++ language to implement and test the Simulated Annealing partitioning technique.

We began with creating a data structure which we could use to implement the algorithm. The most basic data structure of the algorithm is the ProcessFunction class. The following lines of code show the definition of the ProcessFunction class variables:

```
class ProcessFunction{
public:
        int funcId;
        list<pair<int, int> > funcBlockOfStatements;
        int numOfLoops;
        map<int, pair<int, pair<int, int> > > perLoopBlockStatements;
        map<int, double> CL;
        map<int, double> RCL;
        int func_Nr_opi;
        int func_Nr_kind_opi;
        int func_L_path;
}
```

The funcId variable holds a unique identification number for every ProcessFunction object. The different individual number of operations (number of activations) as well as the kind of operations (weight of operations) are stored as a pair within a list. This means that if the list contains N pairs, then there are N different blocks of statements within the function, each with its own attribute (number and kind of operations). The variable numOfLoops stores the number of different loops present within the function. The hashmap perLoopBlockStatements is used to hold the individual attributes of the block of statements which is present within each different loop. The key of the hashmap is a loop id (which will be unique across the whole program; assigned to the key by the program) and the value is a pair - the first element of the pair will hold the number of iterations within the loop; the second element is a pair again, which holds the number of operations and kind of operations for the block statements within the loop. The creation of the data structure in such a way ensures that implementing the first step of partitioning (extraction and creation of new processes) can be done easily, since blocks of code or loops which are computationally intensive can be easily removed from the list or map and added to a new ProcessFunction object which is created.

The CL and RCL maps respectively hold the computational load and relative computational load statistics for the different blocks of statements and loops (key is loop or blocks id) within the function. func_Nr_opi, func_Nr_kind & func_L_path hold values for total number of operations,

number of different operations and length of the critical paths within the different subprograms of the ProcessFunction (like blocks of statements, loops, etc.).

The next data structure created by us is the Process class. The following psuedocode gives an idea regarding the Process class:

```
class Process{
public:
        int pid;
        vector<ProcessFunction*> processFuncs;
        list<pair<pair<int, int>, pair<int, int> > > communicationMapping;
        double proc_CL;
        double proc_RCL;
        int proc_Nr_opi;
        int proc_Nr_kind_opi;
        int proc_L_path;
}
```

As can be seen in the pseudocode, Process class objects will be assigned a pid, which will be unique across processes in the entire program. The processFuncs variable is an arraylist of ProcessFunction type, i.e., it will hold reference pointers to all functions present within a particular process. The communicationMapping list is used to hold the information regarding function calls, both within as well as external to a process. The list can be considered to be holding tuples as - { [Process Id of process containing *calling* function, Function Id of *calling* function], [Process Id of process containing *called* function, Function Id of *called* function] }. Thus, communication between process as well as within the same process can be easily managed through the use of this list. The variables proc_CL, proc_RCL, proc_Nr_opi, proc_Nr_kind_opi, proc_L_path again store similar details as the corresponding ProcessFunction class variables, only this time for the entire process.

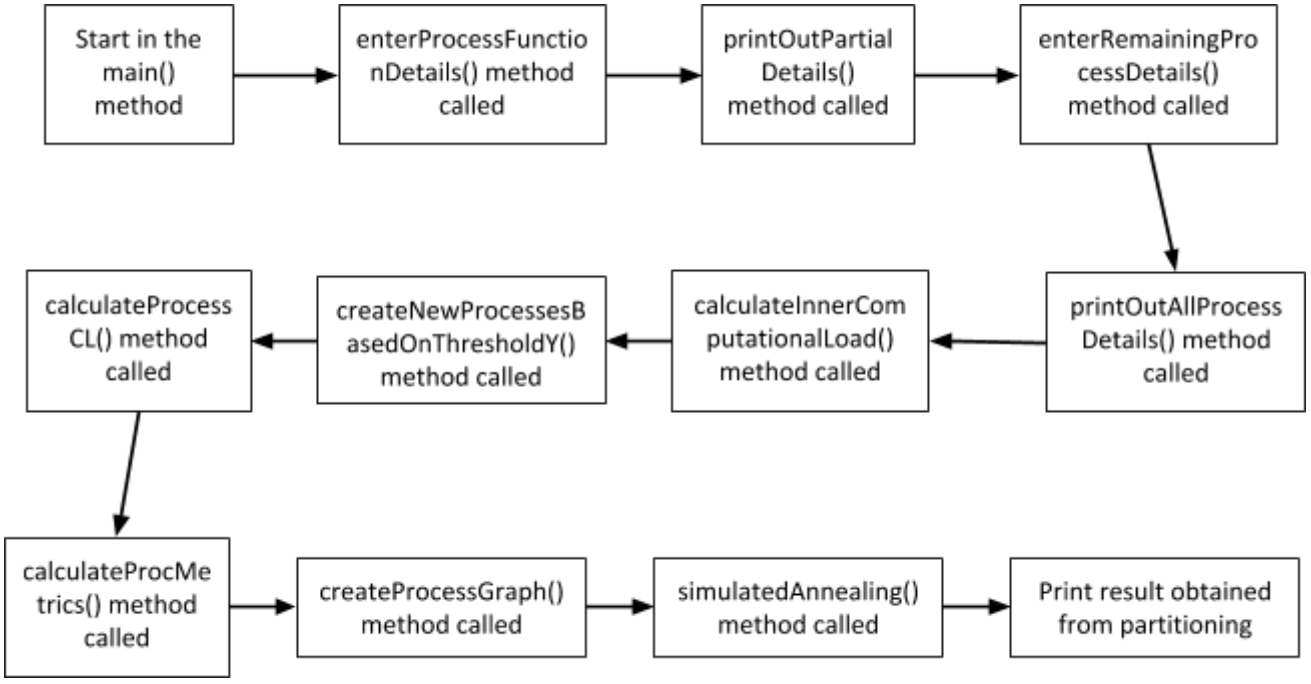
The final data structure which was used for the implementation is that of the ProcessGraphNode class. The following is a pseudocode of the class:

```
class ProcessGraphNode{
public:
        Process* node;
        vector<pair<ProcessGraphNode*, pair<double, double> > > edgesAndEdgeWeights;
        bool hwNode;
        double nodeWeight1, nodeWeight2;
}
```

This data structure is used to create different nodes required for the process graph, which is used to partitioning. As can be seen, the node variable is a pointer to a Process type object. Each node of the graph will hold the process, which is done through the node variable. The edgeAndEdgeWeights arraylist holds a tuple of another node in the graph, as well as another pair, which is used to hold the weights W1E and W2E associated with that particular edge. Put simply, each tuple in the arraylist contains - { [Pointer to another node in Process Graph, [weight W1E associated with the

edge, weight W2E associated with the edge] ] }. The boolean variable hwNode is used to indicate whether the node in the process graph is currently on the hardware partition or on the software partition. If the variable is true, then it means that the node is in the hardware partition, and if it is false, then this node is in the software partition. nodeWeight1 and nodeWeight2 store weights W1N and W2N corresponding to the particular node object.

With a basic understanding of the data structures used, we now explain the control flow of our implementation. The program begins execution at the main class. The following flowchart gives the sequence with which functions are called, in order to implement the Simulated Annealing partitioning technique:




As seen in the flowchart, the program begins its execution at the main method. The very first step is to create an arraylist of Process type, which is used as the set of interacting processes. This arraylist is populated using user input, i.e., the user keyes in the initial number of process, the initial functions within the processes, and the block of statements and loop attributes - all in the enterProcessFunctionDetails() method. According to the paper, this initial attaining of the input data is done through VHDL, or some other lower level languages, but we are assuming that our user is knowledgeable with regards to the input data and we will start our program implementation at this level. Once the initial details are taken in by the user, the data entered is output along with the unique process Ids, function Ids and loop Ids at the user terminal. This is done in the printOutPartialDetails() method. Once the user has an overview of the data structure filled so far, he/she is next asked to input any knowledge he/she might have with regards to function calls between the different functions; this is done in the enterRemainingProcessDetails() method. The user inputs the newly created function Ids which he/she is able to see on the terminal now; this

user input is used to fill the communicationMapping variable of the Process object. Once the user inputs the function call details into the terminal, the entire data structure filled so far is displayed on the screen and the program begins execution of the steps for partitioning.

The next method called is calculateInnerComputationalLoad(). This method will calculate and fill in the CL and RCL map variables for the ProcessFunction objects which have been created for each Process object. Once these have been calculated and updated, createNewProcessesBasedOnThresholdY() method is called. This method will go through the relative computational load of each attribute of the function (blocks of statements and loops), and if any of the attributes pass the threshold (Threshold Y in the paper), then a new process will be created and these attributes will be moved over to the new process. This means that if there is a loop within a particular function in a process that has RCL value greater than the threshold set, then a new process will be created, and this loop alone will be moved from its original function and process to a new function within new process. This move will also cause an entry into the communicationMapping list used, since now there is a new call to a function in another process. Similarly, all subprograms within each function of all processes are checked if they exceed the threshold and new processes are created if any of them do. This completes the first step of the partitioning as given in the paper.

After this calculateProcessCL() method is called, which will calculate and populate computational load and relative computational load for the entire process. Following this, calculateProcMetrics() method is called, whereby, func_Nr_opi, func_Nr_kind & func_L_path are first calculated and stored within each ProcessFunction object of a Process object, followed by calculation and population of proc_Nr_opi, proc_Nr_kind_opi, proc_L_path class variables for all Process objects created. This will enable us to calculate weights required for creation of process graph.

We next create the process graph. An arraylist of ProcessGraphNode class objects is first created. Then, createProcessGraph() method is called, which will use the earlier created arraylist of interacting processes as well as width of data channel to create each node of the graph. This method will internally call multiple other functions in order to calculate weights as well as set edges for the nodes. Initially, the entire set of interacting processes are divided in half, i.e, the first half of the arraylist are put in the hardware partition and second half are put in the software partition. There is no initial cost calculation during this step. Once the graph is created, simulatedAnnealing() method is called with the created graph and an initial temperature. This algorithm has been implemented as in the paper, and it iteratively calculates the cost function in order to determine the partitioning which will give the least cost for its implementation. This final partitioning is displayed to the user. The entire code can be found in the Appendix section of the report.

There are also multiple user defined weight multipliers such as MCL, MU, MP, MSO, Q1, Q2, Q3, etc. as well as thresholds and initial temperature values which have not been explicitly defined in the paper; hence we have taken arbitrary values in order to perform the task at hand. Other than

this, we have also not considered any form of constraints, other than the ones used for calculating weights of a node of a process graph. We have also not experimented with Tabu search mentioned in the paper.

# 4. APPLICATION OF ABOVE IMPLEMENTED ALGORITHM

An application which we decided to use to test the algorithm was one that we read in another paper, where a wind turbine model was explained briefly. An overview of the entire model was provided in the paper, which allowed us to estimate as well as assume to some extent, the input data required for our model. The core functionalities of the model are - Main Controller (directs overall control), Pitch Regulator (calculates pitch angle), Park and Brake Controller (responsible for park or brake of turbine), a diagnostic system (for measuring readings), Input Signal Filter, as well as the main plant used to provide a feedback loop to the main input. We approximated all the functional aspects to take ~6 processes for its full implementation.

Since we did not have access to hardware components, we had to approximate nearly all the process and function attributes which would be used. We decided on function attributes ourselves based on how much importance we gave to each component of the system. We decided on the following inputs:

- Process 1 (For Main Controller): 5 functions and 5 function calls
  - Function 1:
    - 2 Blocks of statements - [(8, 3) (6, 8)]
    - 3 loops
      - Loop 1 - 10 iterations, [(8, 3)]
      - Loop 2 - 50 iterations, [(10, 7)]
      - Loop 3 - 82 iterations, [(9, 23)]
  - Function 2:
    - 1 Blocks of statements - [(2, 77)]
    - 2 loops
      - Loop 1 - 47 iterations, [(83, 2)]
      - Loop 2 - 73 iterations, [(8, 87)]
  - Function 3:
    - 4 Blocks of statements - [(83, 66) (7, 8) (2, 84) (36, 88)]
    - 0 loops
  - Function 4:
    - 3 Blocks of statements - [(54, 73) (33, 74) (23, 95)]
    - 1 loops
      - Loop 1 - 837 iterations, [(8, 1)]
  - Function 5:

        - 0 Blocks of statements
        - 5 loops
            - Loop 1 - 43 iterations, [(9, 8)]
            - Loop 2 - 654 iterations, [(9, 33)]
            - Loop 3 - 862 iterations, [(95, 7)]
            - Loop 4 - 3834 iterations, [(32, 94)]
            - Loop 5 - 42 iterations, [(92, 55)]
    - Function call 1: Function 3 of Process 1 to Function 2 of Process 4
    - Function call 2: Function 2 of Process 1 to Function 1 of Process 2
    - Function call 3: Function 4 of Process 1 to Function 3 of Process 1
    - Function call 4: Function 2 of Process 1 to Function 2 of Process 5
    - Function call 5: Function 1 of Process 1 to Function 1 of Process 6

- Process 2 (Pitch and Brake controller): 2 Functions and 1 function calls
    - Function 1:
        - 3 Blocks of statements - [(7, 8) (90, 77) (8, 999)]
        - 1 loops
            - Loop 1 - 983 iterations, [(8, 3)]
    - Function 2:
        - 4 Blocks of statements - [(9, 73) (8, 37) (93, 9) (23, 33)]
        - 2 loops
            - Loop 1 - 23 iterations, [(9854, 7)]
            - Loop 2 - 65 iterations, [(6743, 5)]
    - Function call 1: Function 2 of Process 2 to Function 1 of Process 5

- Process 3 (Input Signal Filter): 1 Functions and 2 function calls
    - Function 1:
        - 1 Blocks of statements - [(10000, 900)]
        - 0 loops
    - Function call 1: Function 1 of Process 3 to Function 2 of Process 5
    - Function call 2: Function 1 of Process 3 to Function 4 of Process 1

- Process 4 (Input Signal Filter): 2 Functions and 0 function calls
    - Function 1:
        - 3 Blocks of statements - [(32, 43) (12, 9) (55, 8)]
        - 1 loops
            - Loop 1 - 73 iterations, [(635, 88)]

- Function 2:
  - 4 Blocks of statements - [(1, 1) (2, 3) (73, 7) (21, 7)]
  - 0 loops

- Process 5 (Park and Brake Controller): 3 Functions and 1 function calls
  - Function 1:
    - 1 Blocks of statements - [(500, 200)]
    - 0 Loops
  - Function 2:
    - 1 Blocks of statements - [(8, 1)]
    - 2 loops
      - Loop 1 - 71 iterations, [(92, 7)]
      - Loop 2 - 93 iterations, [(3, 5)]
  - Function 3:
    - 1 Blocks of statements - [(8, 700)]
    - 1 loops
      - Loop 1 - 850 iterations, [(9000, 9)]
  - Function call 1: Function 2 of Process 5 to Function 1 of Process 5

- Process 6 (diagnostic system): 1 Functions and 0 function calls
  - Function 1:
    - 3 Blocks of statements - [(100, 200) (200, 250) (300, 100)]
    - 3 loops
      - Loop 1 - 600 iterations, [(746, 8)]
      - Loop 2 - 300 iterations, [(845, 888)]
      - Loop 3 - 920 iterations, [(23, 7)]

These were values input into the program to obtain the final partitioned result. The program was run with the following constants and thresholds:

- MCL = 0.5
- MU = 10.0
- MP = 5.0
- MSO = 15.0
- Q1 = 5
- Q2 = 7
- Q3 = 10
- threshold_Y = 0.35
- widthOfDataChannel = 100;

- initialTemperature = 400 (changed during testing to perform test cases)
- alpha = 0.95 (changed during testing to perform test cases)

The following table presents how the execution time varied with response to changes initialTemperatures and alpha using the partitioning algorithm:

| **initialTemperatures** | **alpha** | **Execution Time with Simulated Annealing Partitioning** |
|---|---|---|
| 400 | 0.95 | ~15 min |
| 200 | 0.95 | ~10 min |
| 100 | 0.80 | ~5 min |
| 400 | 0.80 | ~8 min |
| 600 | 0.90 | ~25 min |
| 800 | 0.95 | >40 min |
| 400 | 0.85 | ~11 min |

As can be seen from the table, increasing initialTemperatures or alpha both lead to an increase in execution time. This is mainly because increasing initialTemperatures increases the iterations to be performed, while increasing alpha decreases the rate at which the initialTemperatures reaches zero.

The same application when executed without using the partitioning algorithm took nearly 30 minutes to complete. The reason behind this was that the data structures were still being created which took nearly the same as with using the partitioning algorithm. However, in this case all permutations and combinations would have to be tested without the use of parameters such as alpha or initialTemperatures. The complexities of the data structures created also did not help during the iterative process of checking all possible combinations. This made the entire experiment (execution without the algorithm) an extremely slow one.

# 5. CONCLUSION

The project helped us to explore different options available for hardware-software partitioning and also use our own ideas to come up with solutions on how to implement as well as apply one of them to use. It greatly broadened our scope and understanding of how hardware-software partitioning is generally done in practical cases.

# 6. REFERENCES:


1. All the research papers required to be reviewed for the project
2. Google Search
3. Wikipedia