ESE Project2 Report

System Level HW/SW Partitioning Based on Simple Simulated Annealing

May 20, 2019

Contents

1 Introduction

Hardware/software partitioning is very critical step in embedded system, which can help to obtain the best performance in the limited cost or can decrease the cost in the limited performance.In this project, we chose the simple simulated annealing algorithm to implement, the related paper is *System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search* . The reason we chose it for it's well discussed in class and its detail description above the other four papers stated the algorithm of hw/sw partitioning. The main goal of partitioning is to maximize performance in terms of execution speed.

In this report , we mainly focus on 3 parts:Summaries of Papers in section 2,Implementation of Simulated Annealing in section3, and Experiment and Result Analysis in section 4.

2 Summaries of Papers

2.1 An Overview of Formal Hardware Specification Languages

Choosing an appropriate specification language for a particular verification problem is very important. The ideal specification language, which stated in the paper of An Overview of Formal Hardware Specification Languages, should possess four characteristics: 1) Precise semantics; 2) Short learning curve; 3) Connections with automatic verification techniques; 4) Integration with existing design practice. All the specification languages can be divided into two different parts: 1) Textual Language, which constitute most current specification languages; 2) Visual Language, which comprise a small number of available specification languages, but are typically easy to learn.

2.1.1 Textual languages

They were divided into hardware description languages and conventional programming languages, which tend to describe designs in low-level details, and those arising from programming languages, which tend to specify behavior at a higher abstraction level. In hardware description languages section, basically there are two approaches: 1) hardware monitors and 2) Objective VHDL. The first is used for monitor modules as aids for simulation-based verification, and the second one is used for raising the abstraction level at which activities occur, enhancing design reuse. Next, we consider SpecC and Java for hardware specification. SpecC encourages design reuse by providing a single, executable language in which to document many design activities, and Java, a high-level, object-oriented programming language, enables hardware/software co-design by providing a single simulation environment for the entire system.

2.1.2 Visual languages

They were divided into those stemming from software engineering, hardware engineering and system engineering.

Typically, there are three languages make up specification languages developed to aid software engineering: 1) Harels Statecharts; 2) Message Sequence Charts; 3) Specification and Description Language. Statecharts extend state machine notation by adding mechanisms for expressing hierarchy, concurrency and communications. They are easily learned and integrated into industrial practice, however, assigning semantics to Statecharts is difficult. Message Sequence Charts (MSC) strength lies in their ability to describe communication between cooperating process. They are very easy to understand and are already used in some informal protocol specifications, but the issue serves to make the already difficult task of analyzing MSCs even more problematic. Specification and Description Language (SDL) is similar to Statecharts, while the disadvantage of this is that protocol specification descriptions tend to be large, and therefore difficult to understand and maintain.

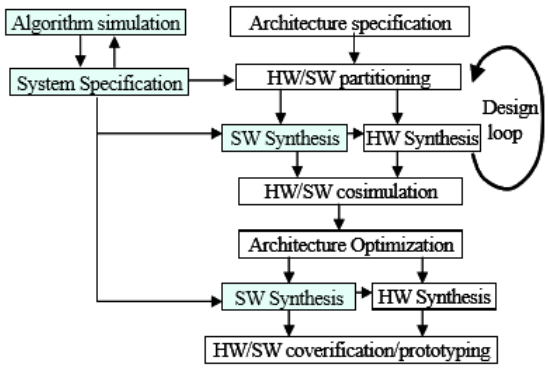
State machine diagrams, waveforms and circuit schematics are typical visual specification languages used in hardware design. Heterogeneous Hardware Logic presents a logic composed of hardware description diagrams, includes diagrammatic notations for circuit diagrams, algorithmic state machines, and timing diagrams. This language has been wildly learned and fully formalized, so no re-training is required and the models produced are already use in industrial practice, but it is better suited to human intensive verification techniques than to automatic ones.

Live Sequence Charts (LSCs) language is proposed recently, which its developers use to specify an automatic rail-car system. It extends Message Sequence Charts in several ways. However, its weakness is lack of a timing model. This reduces the languages facility for describing hardware protocols.

2.2 Hardware-software Codesign of Multimedia Embedded System: The PeaCE Approach

PeaCE, which is a full-fledged codesign environment, provides seamless co-design flow from functional simulation to system synthesis, mainly targeting for multimedia applications with real-time constraints. And also, PeaCE is developed as a re-configurable framework to which a third-party design tool can be easily integrated, as stated in this paper.

Hardware/software co-design process starts with system specification. Usually we can use C code descriptions in this step, but it is not adequate for initial system specification in the system level design. The next step is to map the system behavior to an optimal hardware architecture. In this step, we perform hardware/software partitioning and component selection at the same time. After partitioning decision is made, PeaCE generates the partitioned codes for each processing element and co-simulates the system to generate the memory traces from the processing elements. After the communication architecture is determined, we verify the final architecture before synthesis. This step performs time-accurate hardware/software co-simulation for co-verification. Finally, we generate the codes for the processing elements in a prototyping board. This whole step is showed in the following figure:



PeaCE specifies the system behavior with a heterogeneous composition of three models of computation: dataflow model, FSM model, and task-level specification model. The PeaCE environment facilitates all codesign steps from system specification to prototyping utilizing the features of the formal modes maximally during the whole design process.

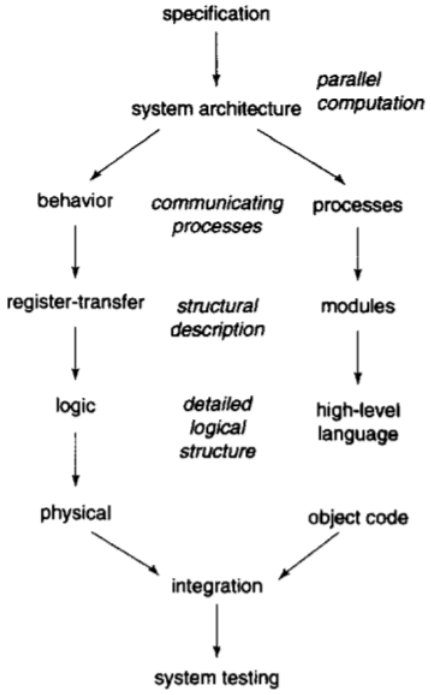
2.3 Hardware-Software Co-Design of Embedded Systems

According to the viewpoints of this thesis, embedded system design can be divided into four major tasks: 1) Partitioning the function to be implemented into smaller, interacting pieces; 2) Allocating those partitions

to microprocessors or other hardware units, where the function may be implemented directly in hardware or in software running on a microprocessor; 3) Scheduling the times at which functions are executed, which is important when several functional partitions share one hardware unit; 4) Mapping a generic functional description into an implementation on a particular set of components, either as software suitable for a given microprocessor or logic which can be implemented from the given hardware libraries. The design goals in each task depend on the application: performance, manufacturing cost, testability, etc.

Methods for the design of an embedded systems hardware and software can be divided into following sections: 1) Process Models and Program Specification; 2) Hardware Engine Design; 3) System and Hardware-Software Partitioning; 4) Distributed System Scheduling; 5) Process Partitioning; 6) Distributed Process Allocation.

The design process of an embedded system must vary considerably with the application: the design of a paper is very different from the design of an autopilot. In a top-down design process, design starts with the creation of specification, as shown in this figure.



In this section, a system is described in terms of two models: a system requirements model and a system architecture model. These two modes are jointly refined in a spiral development cycle. A requirements model consists of a data flow diagram, a control flow diagram, response time specifications, and a requirements dictionary. The architecture model includes an architecture flow diagram, an architecture interconnect diagram, and a dictionary. After the architectural decisions have been made, hardware and software design can proceed somewhat separately. Hardware design proceeds through several steps: a description of behavior, a register-transfer design, the logic design, the physical design, etc. Software design starts with a set of communicating processes since most embedded systems have temporal behavior which is best expressed as a concurrent system. Once the hardware and software components have been implemented, they must be separately tested, integrated, and tested again.

Once we start designing an embedded system, we must take care of the system performance. Soft and hard performance goals are essential parts of the specification of most embedded systems. The goal of system performance analysis is to translate performance specifications, which are typically given on user-level functions, into constraints on the design of the hardware engine and the application software. System performance analysis includes several tasks: determining the implications of performance specifications, estimating the hardware costs of meeting performance constraints, identifying key development bottlenecks, and estimating development time.

Select appropriate hardware architecture for an embedded system is significantly essential. Any CPU may be used in an embedded computer system. An application-specific processor (ASIP) is a CPU optimized for a particular application. Nowadays, considering of time constraints, money cost, size and number of devices, root-system and subsystems, etc, many embedded systems are implemented as distributed systems, with code running in multiple processes on several processors, with inter-processor communication links between the CPUs. Meanwhile, we have to consider of CPU performance to meet all the constraints. For example, interrupt latency the time required for the CPU to execute the first instruction of an interrupt handler after an interrupt is raised - can significantly affect performance. Moreover, we have to take care of the performances of tasks on shared CPUs. To estimate the performance of a concurrent system, we must be able to estimate the performance of a single process. Software performance estimation estimates bounds on the running time of a single-threaded code fragment when run on a specified processor. While CPU modeling concentrates on a stream of instructions small enough to fit in the processor pipeline and cache, software performance estimation analyzes larger sections of code. Software performance estimation can be divided into two steps: 1) identifying legal paths through the code; 2) determining the execution time of each path. There are lots of approaches doing this. For example, execution graphs, which are flow charts which use fork and join operators to specify concurrent activity; bounds declarations to capture user execution information which could not be directly derived from the program and more accurately estimate maximum execution time; fast timing analysis technique for use in hardware-software partitioning, etc.

A systems function must be partitioned when it is implemented on either multiple physical units or onto heterogeneous units. System partitioning requires some performance information to be able to compute the systems critical performance path, but partitioning should use a simple timing model which can be quickly evaluated while analyzing large partitioning problems. Hardware-software partitioning algorithms try to meet performance goals by implementing some operations in special-purpose hardware. Most partitioning algorithms divide the behavior specification into a set of software processes running on one CPU and one co-processor: ones which start with all operations in hardware and move some to software; and ones which start with everything in software and move some operations to hardware. In this part, the author cited several algorithms, such as: 1) migrating operations from hardware partition to the software partition; 2) identifying critical operations in an instruction stream and moves those operations to hardware.

2.4 Embedded Software in Real-Time Signal Processing Systems

In Embedded Software in Real-Time Signal Processing Systems, increasing the amount of software in an embedded system, several important advantages can be obtained. First, it becomes possible to include late specification changes in the design cycle. Second, it becomes easier to differentiate an existing design, by adding new features to it. Finally, the use of software facilitates the reuse of previously designed functions, independently from the selected implementation platform. There are two main aspects in these developments: 1) Architectural retargetability; 2) Code quality. These are also the issues that todays embedded processors are facing. Thats because: 1) Instruction-level parallelism and the occurrence of heterogeneous register structures, the different compilation phases become strongly interdependent. In order to generate high quality code, each code generation phase should take the impact on other phases into account. This is called phase coupling; 2) In order to generate high quality code, more specialized compiler algorithms may be required, that explicitly take into account aspects like heterogeneous register structures. An important

point is that larger compilation times can be tolerated in the case of embedded processors, compared to general-purpose microprocessors.

On the processor architecture level, a processor architecture based on the following parameters: 1) arithmetic specialization, 2) data type, 3) code type, 4) instruction format, 5) memory structure, 6) register structure, and 7) control-flow capabilities. The use of efficient and powerful models to represent all required characteristics of a processor is a key aspect in making the software compilation process retargetable. For each of the compilation phases, many compiler attempts have been made to use a single processor model for retargetable compilation, supported with a user-friendly processor specification language. Basically, it can be divided into 1) Netlist-Based Language, 2) High-Level Language. There are some processor models for compilation: 1) Template Pattern Bases: A first approach, used by traditional compilers for general-purpose CISC and RISC processors, is to represent the target processor by means of a template pattern base, that essentially enumerates the different partial instructions available in the instruction set; 2) Graph Models: An alternative approach, mainly used in compilers for embedded processors, is to represent the processor by means of a graph model. Such a model has the advantage that it can more readily represent structural information, which makes it possible to describe several of the peculiarities of ASIP architectures.

Going to the code selection phase of code generation, including those approaches that incorporate local register allocation in the code selection process. There are several techniques for optimizing code selection: 1) Dynamic Programming, 2) LR Parsing, 3) Graph Matching, 4) Bundling, and 5) Rule-Driven Code Selection. For global register allocation, there are also two techniques: 1) Graph Coloring, and 2) Data routing. Meanwhile, there is a problem related to register allocation is the allocation of data memory locations for data values (scalar) in the intermediate representation. This is important when memory spills have been introduced in the register allocation phase, or for passing argument values in the case of function calls. Often these values will be stored in a stack frame in data memory. Thus, there was an important issue is the addressing of values in a stack frame in memory. As a solution, we can support a pointer modification in this phase.

Due to the lack of instruction level parallelism, no additional scheduling phase is required. However, the scheduling task is essential for architectures that exhibit pipeline hazards or instruction-level parallelism. Even when the parallelism is restricted, scheduling is a crucial task for targets such as DSP processors and ASIPs, because of the requirement of high code quality, which implies that the scarce architectural resources should be used as efficiently as possible, including the possibilities for data pipelining. This is especially true for deeply nested blocks in the algorithmic specification. 1) Local Versus Global Scheduling. When the architecture has only a restricted amount of instruction-level parallelism, a local scheduling approach may already produce efficient results. However, in the case of more parallel architectures, there may be a mismatch between the architectural parallelism offered by the processor and the algorithmic parallelism within individual basic blocks. To use the processors resources effectively, a global scheduling approach is required, whereby partial instructions can be moved across basic block boundaries; 2) Global Scheduling Techniques for Conditional Branches: 1. Trace scheduling; 2. Percolation based scheduling; 3) Techniques for Software Pipelining: 1. Modulo scheduling; 2. Loop folding.

2.5 Dynamic Hardware/Software Partitioning: A First Approach

In the paper of Dynamic Hardware/Software Partitioning: A First Approach, the dynamic partitioning approach could solve the partitioning problems, including: 1) The designer must use an appropriate profiler to detect regions that contribute to a large percentage of program execution; 2) The designer must use a compiler with partitioning capabilities to partition the software source; 3) The designer must apply a synthesis tool to convert the partitioning compilers hardware description output to an FPGA configuration. The dynamic hardware/software partitioning has the following advantages: 1) transparent; 2) reusable; 3) supports legacy programs. Dynamic hardware/software partitioning is feasible if the partitioning module

can fit in a small enough area. Power of the dynamic partitioning module is small compared to overall chip power. However, the drawback of dynamic partitioning could be less optimized performance and energy of the partitioned designs.

The following figure shows the dynamic hardware/software partitioning system architecture: a) overall architecture, b) dynamic partitioning module architecture, c) configurable logic architecture.

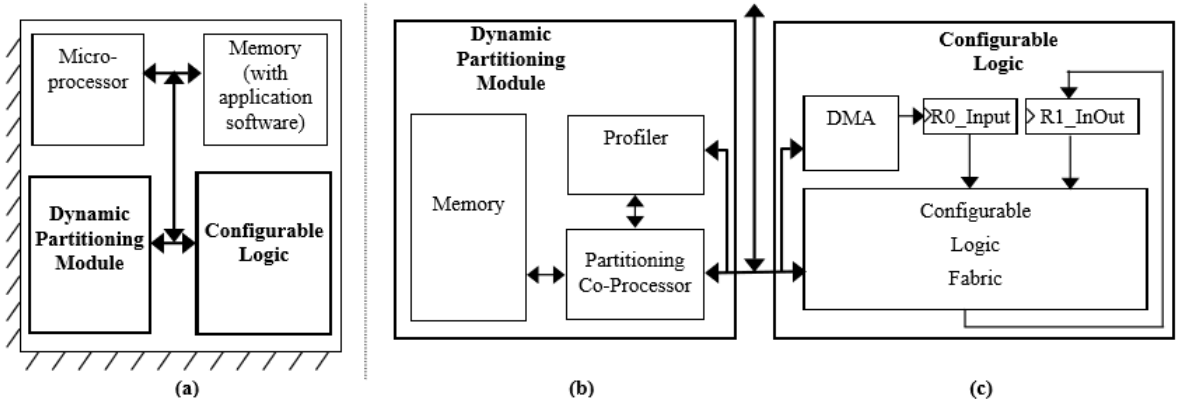
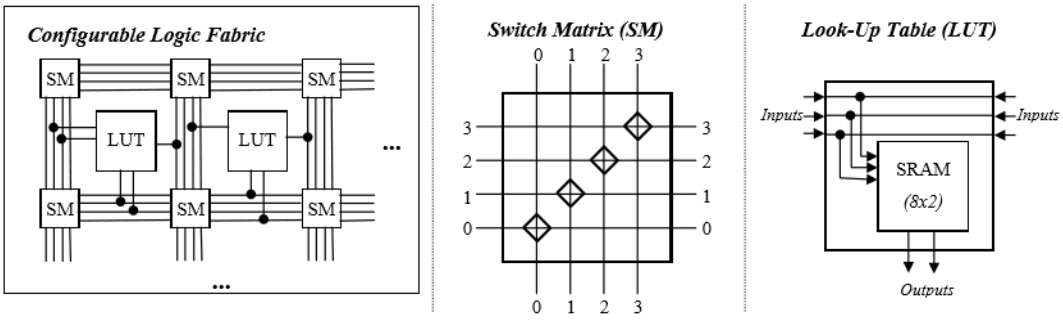


Figure (a) shows the overall architecture for dynamic hardware/software partitioning. The architecture consists of a standard embedded microprocessor and memory for normal application software execution. The architecture also includes on-chip configurable logic. The dynamic partitioning module dynamically detects the most frequently-executed software regions and reimplementes those regions as hardware on the configurable logic. Figure (b) shows the architecture of the dynamic partitioning module. The module includes a partitioning co-processor and memory to run a program that decompiles and synthesizes selected binary regions for hardware implementation. Figure (c) shows the architecture for the configurable logic. The configurable logic uses a direct memory access (DMA) controller to access memory. For the configurable logic fabric architecture, it consists of a matrix of simple 3-input 2-output look-up tables (LUTs) surrounded by switch matrices (SMs) for routing. The architecture of CLF, SM, LUT are in the following figure:



The tool flow for dynamic hardware/software partitioning: First, the loop profiler detects regions of software that should be implemented as hardware. The profiler uses a small cache of only a few dozen entries, divides loops into small, frequent loops with a small amount of associativity, to save area and power. Second, decompilation converts the software loops into a high-level representation more suitable for synthesis. Third, the DMA configuration tool maps the memory accesses of the decompiled loop onto our DMA architecture. During DMA configuration, we can remove loop counters and exit conditions for the decompiled loop. In addition, since memory accesses are limited to sequential locations, we can remove all address calculations from the decompiled loop. Fourth, RT and Logic synthesis. In this step, register-transfer

(RT) synthesis converts each output bit into a Boolean expression by traversing the dataflow graphs of the software region. Logic synthesis followed by technology mapping, placing, and rout convert the Boolean equation into a netlist. After logic synthesis, technology mapping traverses the DAG backwards starting with the output nodes and combines nodes to create LT nodes corresponding with 3-input single-output LUTs. Finally, we perform routing between inputs, outputs, and LUTs using a simple greedy algorithm, thus we can route the wires connecting LUTs together. Bitfile creation combines the placed and routed hardware description with the DMA configuration information into a single bitfile that we use to initialize the configurable logic. Meanwhile, binary modification handles updating the software binary to utilize the hardware for the loops, in order to make whole performance directed in the expected way.

2.6 Configuration-Level Hardware/Software Partitioning for Real-Time Embedded Systems

In hardware/software partitioning for real-time embedded systems, the level of abstraction that we have adopted for modeling hardware and software components is called configuration level, which is similar to the system level, as stated in this paper. Performing hardware/software partitioning at the configuration level allows us to efficiently analyze the timing behavior of a large number of implementations with respect to the system timing specifications.

At the configuration level, hardware is modeled as resources and software as tasks utilizing the resources. In particular, software is partitioned into tasks and each task is represented by the amount of memory and estimated number of instructions required to execute it. The corresponding processor model specifies the available memory and time needed to execute a single instruction. Since an application-specific hardware component implements specific system functions, it is modeled as a resource that is used exclusively by the corresponding functions. We approach the hardware/software partitioning 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 either by executing software tasks on processors or by dedicated hardware circuits. There is a library of different processors and hardware components that can be used for such a purpose. Using available data or previous design knowledge, each processor and hardware component is characterized by several attributes, such as cost and power. Then, the partitioning problem becomes that of selecting hardware components and processors and assigning software tasks to processors such that the resultant system is optimized in terms of cost/performance, while satisfying all the given constraints.

GOPS is a configuration-design tool for synthesizing electronic systems. Given a set of functions to implement, a library of parts that implement the functions, a set of constraints that must be satisfied, and a set of attributes for evaluating solutions, GOPS finds the Pareto-optimal set of part-set solutions. Each part set in the Pareto-optimal set implements all functions an satisfies all constraints. For a part set S to be included in the Pareto-optimal set, no other part set may exist with all attribute values better than or equal to that of S. Hence, the optimal design is guaranteed to be in the Pareto-optimal set.

2.7 Online Hardware/Software Partitioning in Networked Embedded Systems

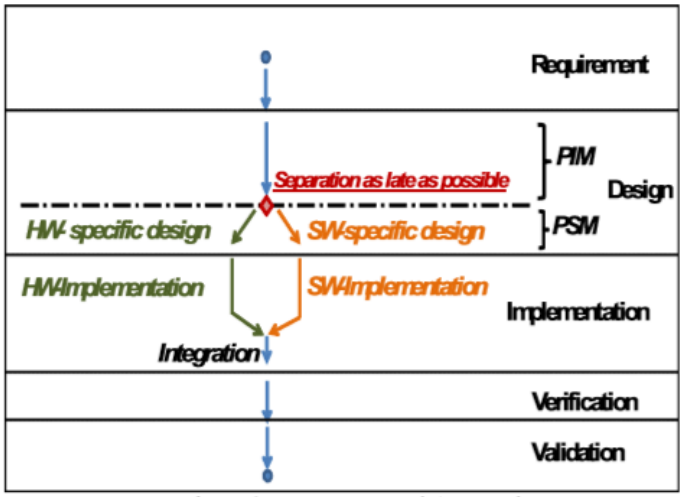
Diffusion algorithm, which for online partitioning and dynamic load balancing, is a strategy based on a class of algorithms that are successfully applied in this paper. Also, we consider embedded systems consisting of networked hardware/software re-configurable nodes with the following properties: 1) Interconnect: A network consists of computational nodes connected by bidirectional point to point links; 2) Embedded: requires the optimization of competing objectives like power, latency or area; 3) Hardware and Software re-configurable: Tasks can be executed either in software or in hardware on a node.

In online hardware/software partitioning, comparing to the original diffusion algorithm, and in order to cope with faults and dynamic load over longer time intervals, an iterative optimization that finds optimal

(re)partitions of tasks is applied, the idea is to increase the likelihood that future node and link failures may be compensated fast due to load reserves in the neighborhood of each node that may fail. Thus, we consider a special diffusion algorithm called Discrete Diffusion Algorithm. With this discrete diffusion algorithm, we have to overcome two problems: 1) it is advisable not to split one process and distribute it to multiple nodes. This increases the data traffic in the network; 2) it could occur that negative loads are assigned to computational nodes. Compared to the continuous diffusion algorithm, the discrete diffusion algorithm is always better concerning the congestion. Moreover, the discrete diffusion algorithm has theoretically deduced the upper bounds for the deviation of the optimal load distribution and the discrete load distribution.

2.8 Partitioning Decision Process for Embedded Hardware and Software Deployment

In the paper of Partitioning Decision Process for Embedded Hardware and Software Deployment, there is a new partitioning method that comprises a complete development process from the requirement management, architectural design, component, modeling, to the decision for their implementation either as software or hardware components. Basically, there are several questions for embedded systems built on heterogeneous platforms: 1) properly enable technology-independent design in the earlier stage of the design phase and perform the partitioning decision process in a later stage; 2) enable a systematic and effective process that supports the design engineers before partitioning; 3) provide an effective and accurate partitioning decision process providing optimal and sustainable results which taken into account requirements and project constraints. In order to solving these problems, there is an approach which is inspired by Model-Driven Architecture with Platform-Independent Model (PIM) and Platform-Specific Model (PSM) stages and supported by Model-based and Component-based approaches, as shown in the following figure:



In addition, the approach enables high-level component reuse. In order to achieve this, a component library is built, which includes existing components and their variants. In the part of activities and partitioning decision table building, there are 6 main activities: 1) Modeling of the application as a set of components; 2) Identification of overall application and project constraints to derive decision criteria; 3) Identification of project and application-related properties; 4) Filtering and property prioritization; 5) Component variants selection; and 6) Solution ranking.

2.9 Hardware/Software Co-Design

Conceptions of the design of hardware/software systems. The design of hardware/software systems involves modeling, validation, and implementation. For implementation section, there are two techniques affecting

the system implementation characteristics: Hardware/Software Partitioning and Scheduling. These two techniques address where and when the system functions are implemented respectively.

2.9.1 Hardware/Software Partitioning

The partition of a system into hardware and software is of critical importance because it has a first order impact on the cost/performance characteristics of the final design. When considering general purpose computing systems, a partition represents a logical division of system functionality, where the underlying hardware is designed to support the software implementation of the complete system functionality. This division is captured by the instruction set. Thus instruction selection strongly affects the system hardware/software organization. 1) Architectural Assumptions, 2) Partitioning Objectives, 3) Partitioning Strategies.

2.9.2 Scheduling

Scheduling can be loosely defined as assigning an execution start time to each task in a set, where tasks are linked by some relations. Scheduling under timing constraints is common for hardware circuits, and for software applications in embedded control systems. Tasks execution requires the use of resources, which can be limited in number, thus causing the serialization of some task execution. Most scheduling problems are computationally intractable, and thus their solutions are often based on heuristic techniques. Basically, the scheduling algorithms can be divided into: 1) Operation Scheduling in Hardware; 2) Instruction Scheduling in Compilers; 3) Process Scheduling in Different Operation Systems.

In the first part, the operation scheduling in hardware has the most approaches in hardware scheduling. Basically, most practical implementations of hardware schedulers rely on list scheduling, which is a heuristic approach that yields good schedules in linear time. Another heuristic for scheduling is force-directed scheduling, which addresses the latency-constrained scheduling problem.

Second, in instruction scheduling, scheduling can be viewed as the process of organizing instructions into streams. It is related to the choice of instructions, each performing a fragment of the computation, and to register allocation. For example, when considering compilation for general-purpose microprocessors, instruction selection and register allocation are often achieved by dynamic programming algorithms, which also generate the order of the instructions.

Finally, in process scheduling in different operation systems, process scheduling is the problem of determining when processes execute and includes handling synchronization and mutual exclusion problems. Algorithms for process scheduling are important constituents of operating systems and run-time schedulers. Process scheduling in real-time operating system is characterized by different goals and algorithms. For example, Rate monotonic analysis is one of the most celebrated algorithms for scheduling periodic processes on a single processor. Processes are statically scheduled with priorities that are higher for processes with higher invocation rate, hence the name. Nevertheless, RM analysis has been the basis for more elaborate scheduling algorithms.

2.9.3 Conclusion

In summary, process scheduling plays an important role in the design of mixed hardware/software systems, because it handles the synchronization of the tasks executing in both the hardware and software components.

2.10 System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search

The partitioning strategy in this paper is based on metric values derived from profiling (simulation), static analysis of the specification, and cost estimations.

When we consider doing partitioning, basically, we have to consider: 1) To identify basic regions which are responsible for most of the execution time in order to be assigned to the hardware partition; 2) To minimize communication between the hardware and software domains; 3) To increase parallelism within the resulted system. Therefore, the hardware/software partitioning is performed in four steps: 1) Extraction of blocks of statements, loops, and subprograms; 2) Process graph generation; 3) Partitioning of the process graph; 4) Process merging. The paper provides us with two partitioning algorithms: 1) Simulated Annealing; 2) Tabu Search.

2.10.1 Simulated Annealing

Simulated annealing selects the neighboring solution randomly and always accepts an improved solution. In general, the simulated annealing algorithms work as follows: At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and then decides to move to it or to stay with the current solution based on either one of the probabilities between which it chooses on the basis of the fact that the new solution is better or worse than the current one. During the search, the temperature is progressively decreased from an initial positive value to zero and affects the two probabilities: at each step, the probability of moving to a better new solution is either kept to 1 or is changed towards a positive value; instead, the probability of moving to a worse new solution is progressively changed towards zero. In partitioning area, the paper implemented two strategies for solution generation: 1) Simple move: a node is randomly selected for being moved to the other partition. The configuration resulted after this move becomes the candidate solution. Random node selection is repeated if transfer of the selected node violates some design constraints; 2) Improved move: accelerates convergence by moving together with randomly selected node also some of its direct neighbors. A direct neighbor is moved together with the selected node if this movement improves the cost function and does not violate any constraint.

Table 3. Partitioning time with SA.

number of nodes	CPU time (s)		speedup
	SM	IM	
20	0.28	0.23	22%
40	1.57	1.27	24%
100	7.88	2.33	238%
400	4036	769	425%

As a conclusion of figure shown above, we can clearly see that Improved move will perform much better than simple move when number of nodes going larger.

2.10.2 Tabu Search

In contrast to simulated annealing, Tabu Search controls uphill moves not purely randomly but in an intelligent way. The Tabu Search approach accepts uphill moves and stimulates convergence toward a global optimum by creating and exploiting data structures to take advantage of the search history of the next move. In Tabu search, short term memory and long term memory are the two key elements, which can be applied to perform diversification that meant to improve exploration of the solution space by broadening the spectrum of visited solutions. Tabu search enhances the performance of local search by relaxing its basic rule. First, at each step worsening moves can be accepted if no improving move is available. In addition, prohibitions are introduced to discourage the search from coming back to previously-visited solutions. The implementation of Tabu search uses memory structures that describe the visited solutions or user-provided sets of rules. If a

potential solution has been previously visited within a certain short-term period or if it has violated a rule, it is marked as forbidden so that the algorithm does not consider that possibility repeatedly.

Table 5. Partitioning times with SA and TS.

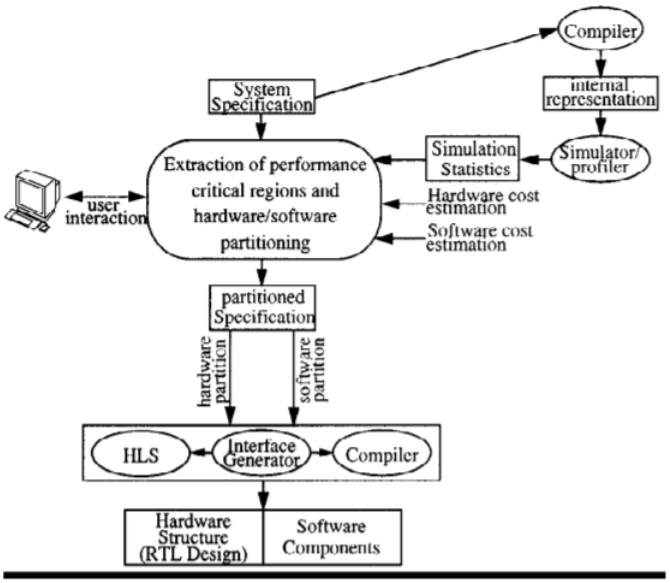
number of nodes	CPU time (s)		t_{TS}/t_{SA}
	SA[a] (t_{SA})	TS (t_{TS})	
20	0.23	0.008	0.034
40	1.27	0.04	0.031
100	2.33	0.19	0.081
400	769	30.5	0.039

a. SA algorithm with improved moves.

From the experiment result shown above, we can know that, comparing to one classical iterative-improvement approach named Kernighan-Lin (KL) algorithm, partitioning times with KL are slightly better than those with SA for small and medium graphs. For the 400 nodes graphs SA outperforms the KL-based algorithm. TS is on average 10 times faster than KL for 40 and 100 nodes graphs, and 30 times faster for graphs with 400 nodes. From the result, we know that performances obtained with Tabu search, which until now has been ignored in the context of system level partitioning, are definitely superior in comparison to those given by even improved implementations of Simulated annealing, or by classical algorithm like KL. This is important as, for a high number of nodes, partitioning times with Simulated Annealing or KL can be prohibitively long, which hinders an efficient exploration of different design alternatives.

3 Implementation of Simple Simulated Annealing

Based on the statement in the paper and the figure as below, we divided our implementation into two parts:get the simulation statistics and realize simulated annealing algorithm.



3.1 Acquisition of the Simulation Statistics

- The input of simulation are the assembly codes which are compiled from VDHL
- The output of simulation are the statistics generated in two seperated files:NodeInfo.txt and edges.txt.The format of NodeInfo.txt and Edges.txt are shown in the below figure.Both the files use the first line to store total nodes or edges, and i_{th} node/edge start from the second line:

```
total_Nodes     total_CL
ith_Node        CLi     Nr_opi  Nr_kind_opi     L_pathi sum_w_opi


total_Edges
ith_Edge        node1i  node2i  CIi
```

For detail implementation in codes, we save a look up table which contains the instructions' set of PSOC and it's corresponding cost(cycles and bits),iterate all the instructions in the input, and search each instruction in the look up table to calculate its CL. And for CI_i on each edge,by counting the number of interaction between nodes, we can figure out the detail data.

3.2 Realization of Simulated Annealing Algorithm

In the main function , there are mainly three functions:Initialize(),cost() , simulatedAnnealing() and display() the best solution The detail implementation of the last two functions are shown as below:

- InitializeAndCost()

  Read in the simulation statistics from the NodeInfo.txt and Edges.txt
  Calculate and store the W1_N and W2_N for each node
  Initialize all the nodes are software nodes
  for edgeCount<totalEdge **do**
    if both node1 and node2 are not both HW nodes or not both SW nodes **then**
      W1_E =W1_E+currentEdge.CI*wd;
    end if
  end for
  for HW_Nodes_Count<total_HW_Nodes **do**
    Compute the total W2_N:W2_N_hw += currentNode.W2_N
    for edgeCount<totalEdge **do**
      if if one of the nodes of an edge is hw node **then**
        W2_E+= currentEdge.CI;
      end if
      Calculate part of the second term in the cost function: parallel=W2_E/currentNode.W1_N;
    end for
  end for
  for SW_Nodes_Count<total_SW_Nodes **do**
    W2_N_SW+= currentNode.W2_N;
  end for
  Use the above parameters to compute cost function
  return cost

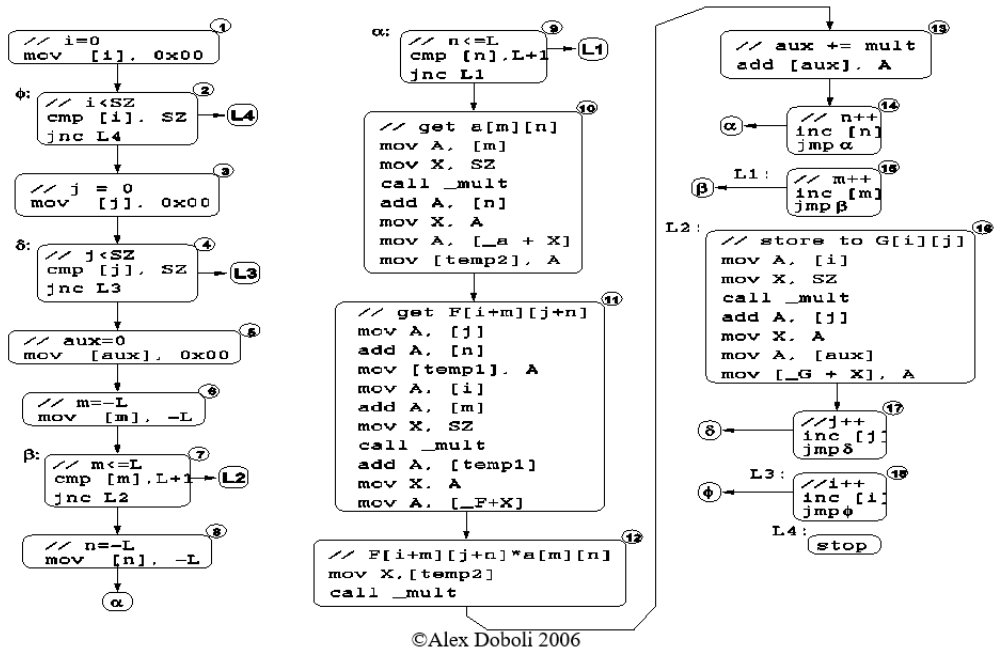
- simulateAnnealing()

  Calculate the cost for the initialize solution
  Get a random number in the range of [1,totalNodes]

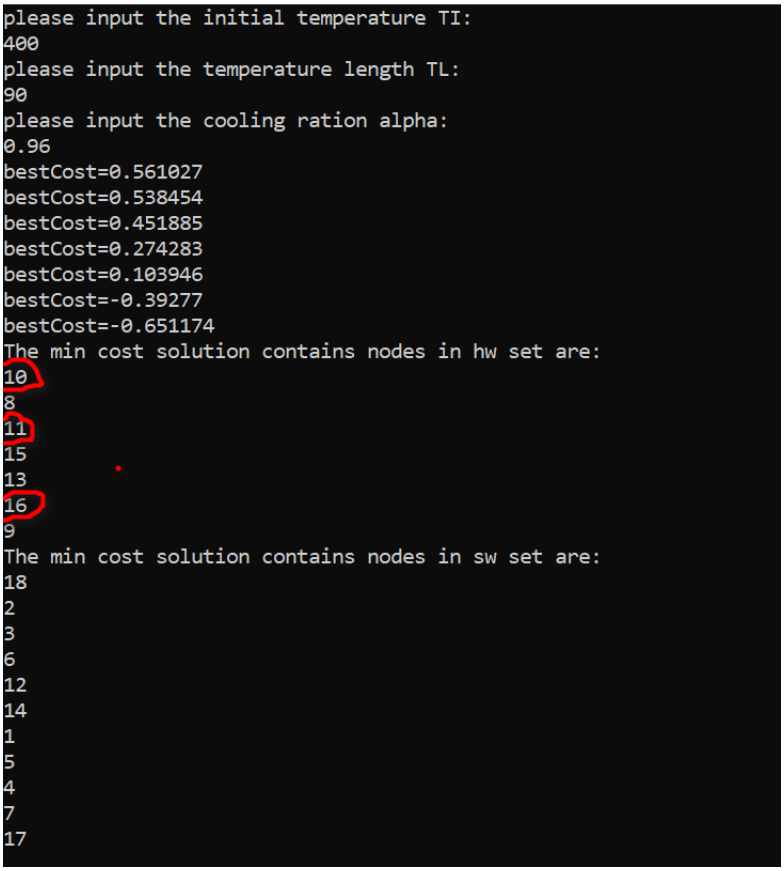
```
Initialize TL,TI
while true do
  for i< TL do
    if node random∈ sw_nodes then
      move node random to the set HW_nodes
    else
      move node random to the set SW_nodes
    end if
    Compute the cost after node random has been moved
    if newCost<bestCost then
      Update the best solution with new solution
    end if
    if newCost>curCost then
      Get a random q = rand() / double(RAND_MAX);
      if q<exp(-diffCost / T) then
        accept the new solution as current solution
      end if
    else
      accept the new solution as current solution
    end if
    if stopCount>2 then
      stop looping
    end if
  end for
end while
```

4 Experiment and Result Analysis

We used the example in lecture 4 which is shown as below to test the Simple Simulated Annealing algorithm.



The granularity in the experiment according to the execution steps in the 4 loops.They are totally 18 nodes. As stated in the section about implementation, we simulated the assembly codes in the source file to obtain the nodes and their corresponding statistics.By using the data of edges, nodes and related statistics, we can calculate the cost for each solution , which is the determining factor for the new solution in the simulated annealing algorithm.In the experiment we used TI=400,TL=90,α =0.96,and when there are no update for the new solution over 2 consecutive iteration, we consider that it is frozen, and stop looping. The best solution will be stored with minimum cost. Below are the result window after running the algorithm.



In the result screen, we can tell that the critical region 10,11,16 have been moved to the set of hardware nodes, but the weakness is that beside the critical regions some others ex,node8,9,13,15.are also divided into the set hardware nodes.During the implementation, we found that the setting of the parameters have an important effect on the hw/sw partitions. The important parameter are:M^{CL},M^{U},M^{P},M^{SO},Q_1,Q_2,Q_3,TI,TL,α, terminated condition ,and so on. It is a heuristic topic on how to design these parameters to make the algorithm work out a perfect solution.

5 Conclusion

In this project, we've just implement the simple simulated annealing algorithm , while the other two listed in the paper are much more efficient. And in order to keep the preciseness, more experiments should be done to test our implementation. All in all, improvements should be done in our implementation.