Institut für Technische Informatik

Abteilung Eingebettete Systeme

Universität Stuttgart
Pfaffenwaldring 5b
D-70569 Stuttgart

Masterarbeit Nr. 01936 - 00x

Heuristics for Design Time Optimization of System-on-Chip Memory Power Consumption

Li Jinpeng

Studiengang:	INFOTECH (Information Technology)
Prüfer:	Prof. Dr.-Ing. Martin Radetzki
Betreuer:	M.Sc. Manuel Strobel
begonnen am:	20.06.2016
beendet am:	13.12.2016
CR-Klassifikation:	B.8.2

Acknowledgment

Acknowledgment goes here...

Abstract

Abstract goes here...

Contents

1 Introduction

Nowadays in the field of embedded systems design , power consumption has become one of the most important design factors especially in the domain of Systems-on-Chip. One of the important issues to design power-efficient embedded system is the power consumed by memories and memory related components. Some researchers have claimed that large fraction of power is dissipated by memories [Mai+07; BMP00]. Thus, memory power optimization plays a significant role in the design of power-efficient embedded systems. One of the most effective and common approaches to reduce memory power consumption is the memory partitioning method which is proposed in several articles and books [BMP00; His05; MBP02, p.43].

The rationale of memory partitioning is , on the one hand, to split one single large memory into several small memory instances which can be accessed individually [Mai+07]. On the other hand, according to the memory access patterns, frequently accessed address ranges are grouped to smaller memory instances while rarely accessed address ranges are grouped to the larger ones [SER16]. The memory partitioning is one of the combinatorial optimizations since the process is to find the optimal memory configuration from a set of memories and applications. There are many methods that can be applied in the domain of combinatorial optimization. One approach is the integer linear programming (ILP) which solves the optimization problem through a mathematical model described by certain integer linear relationship. This approach is proposed in [SER16] for the memory power optimization using memory partitioning method. Another commonly deployed approach is the heuristic which is based on searching mechanisms. In many classical combinatorial optimization problems such as traveling sales man (TSP) problem, heuristics have been deployed and near optimal solutions have been provided by using them.

When dealing with an optimization problem with a very large solution space, the algorithms that are used to find the exact optimal solution may be ideal. However, the required execution time of such algorithms may be unacceptable in practice. Even in some problem sets, the exact optimal solution can not be found by the conventional algorithms. In such cases, the user of heuristics can obtain a near optimal solution within a reasonable time frame. Though the solution provided by heuristics may be not the exact optimal one, it still can be considered as a valuable solution of the optimization problem. One key feature of heuristics is the trade-off between algorithm efficiency and precision. The solution quality and the execution time of the algorithm can be balanced by users according to their different requirements.

The goal of this thesis work is to apply heuristics for the purpose of the memory power optimization using memory partitioning method. The targeted problem set is the same which is used in [SER16]. Firstly, multiple potential heuristics are theoretical examined.

After the comparison between them, the most promising algorithm is identified and proposed for the optimization objective. Secondly, the selected heuristic is adapted to a existing formal power model which is proposed in [SER16]. Then the framework of the chosen algorithm along with its parameters are realized for the power model. Lastly, the evaluation of the implemented heuristic is performed. The optimal solutions found by the chosen algorithm are compared with the obtained results in [SER16].

The structure of this document is organized as following. Chapter 2 introduces the basic knowledge of the memory partitioning method and the formal memory power model. Besides, the discussion of potential heuristics are made. And in this chapter, the first goal of this thesis work is achieved by proposing the simulated annealing algorithm according to the comparison between the discussed heuristics. Chapter 3 discusses the related works for the memory power reduction and the simulated annealing algorithm. Chapter 4 represents the design details of the simulated annealing for the memory power optimization using memory partitioning method. The evaluation of the simulated annealing algorithm results is given in Chapter 5. And Chapter 6 concludes this thesis work.

2 Basics

This chapter represents the basic knowledge for this thesis work. Section 2.1 introduces the memory partitioning method with an existing formal power model. Section 2.2 discusses three potential heuristics. In Section 2.3, the discussed heuristics are compared with each other and the most promising one is selected to be used for the memory power optimization.

2.1 Memory partitioning and formal power model

This section is just a represent of the original work in the article [SER16] and no new material is included.

There are two central concepts for memory power optimization using the memory partitioning method. One concept is the allocation α which is a set of memory instances of certain memory types. The memory types are described by several parameters related to their physical characteristics. The other concept is the binding β of the application's code and data fragments to the selected memory instances. The code and data fragments of an application are referred as application profiles. And each application is represented by a set of profiles [SER16]. Every profile is characterized by some user-defined parameters. Table 2.1 and Table 2.2 describe the relevant parameters of the memory type and application profile respectively.

Parameter	Description
Size	Provided memory space
Area	Consumed on-chip area
Read current	Current required by the read operation
Deselect current	Current required when no operation is performed
Stand by current	Current consumed all the time
Write current	Current required by the write operation (RAM only)

Table 2.1: Memory Type Related Parameters

A configuration for the memory system is defined as a pair of an allocation of memory instances and the corresponding binding for the application profiles. Through the memory partitioning, an optimal configuration is expected to be found such that the average power consumption by the selected memory instances and the interconnect is the lowest under certain predefined constrains. To achieve this optimization objective, a formal power model

Parameter	Description
Duty cycle	The active time frame in the profile's period
Read probability	The probability to perform the read operation in profile's duty cycle
Write probability	Same with read probability except for write operation (RAM only)
Size	Required memory space by the profile

Table 2.2: Application Profile Related Parameters

with four constraints are defined by the authors of [SER16]. In the following, some concepts related to the existing power model along with the constraints are introduced.

Let M denotes the memory type set that can be used in the memory system. The allocation α is represented as a vector whose size is the number of memory types in the set, $\alpha \in \mathbb{R}_0^{|M|}$. Each element in α is the number of instance for the corresponding memory type and its value should be non-negative. Let A denotes the application set provided in the optimization problem. For each application, let P_a represent its profile set. Then the binding β to the corresponding α is in the form of a binary matrix with size $|P_a| \times |M|$, $\beta \in \{0,1\}^{|P_a| \times |M|}$. If the matrix element value β_{aij} of an application a is 1, it means that the profile i of application a is bound to the memory type j. Otherwise, the memory type j does not contain the profile i for application a.

The following are the four predefined constraints.

- Constraint 1
  Define a fixed integer number $mems_{max}$, The total number of the allocated memory instances should not exceed $mems_{max}$.

$$\sum_{i=1}^{|M|} \alpha_i \leq mems_{max} \tag{2.1}$$

- Constraint 2
  Define a vector A_M with size of $|M|$, $A_M \in \mathbb{R}^{|M|}$. Each element in A_M indicates the area consumption of the corresponding memory type. Define a function $A_F \colon \mathbb{N}_0 \to \mathbb{R}$. A_F outputs the area consumed by the interconnect according to the total number of the allocated memory instances. The area required by both memory instances and the interconnect should be limited to a maximum value, $area_{max}$.

$$\sum_{i=1}^{|M|} \alpha_i \cdot A_{M,i} + A_F(\sum_{i=1}^{|M|} \alpha_i) \leq area_{max} \tag{2.2}$$

- Constraint 3
  For each application, every profile should be contained in one and only one memory type. If there are multiple applications, all of them should satisfy this constraint.

$$\forall a \in [1, |A|], \forall i \in [1, |P_a|] : \sum_{j=1}^{|M|} \beta_{aij} = 1 \tag{2.3}$$

- Constraint 4
  Define a vector $\sigma^{P_a} \in \mathbb{N}_0^{|P_a|}$ whose elements indicate the memory consumed by the profiles individually. Define another vector $\sigma^M \in \mathbb{N}_0^{|M|}$ where the total memory spaces of the memories types are recorded in the corresponding elements. For every application, each memory type should have large enough memory space to contain all the profiles that are bound to it.

$$\forall a \in [1, |A|], \forall j \in [1, |M|] : \sum_{i=1}^{|P_a|} \beta_{aij} \cdot \sigma_i^{P_a} \leq \alpha_j \cdot \sigma_j^M \tag{2.4}$$

The configurations satisfy all the introduced constraints are considered as valid for the optimization problem and their average power consumption is computed by the following model. Here, the illustration for the power model is focused on ROM where only read operations are required.

$$P_j(a) = P_{read,j}(a) + P_{desel,j}(a) + P_{stdby,j} \tag{2.5}$$

The power consumption of one single memory type is consisted of three parts. Let $P_j(a)$ denotes the power consumed by memory type j for the application a. Seen from Equation 2.5, the three power fractions are:

$P_{read,j}(a)$, consumed power when reading from the memory.

$P_{desel,j}(a)$, consumed power when the memory is deselected.

$P_{stdby,j}$, power continuously consumed by the memory.

Equations 2.6 to 2.8 show how to compute the three power fractions individually. In these equations, d_i and p_{ri} are the duty cycle and read probability of application profile i respectively. $I_{r,j}(f)$, $I_{d,j}(f)$ and $I_{s,j}$ are the read, deselect and standby currents of memory type j respectively. V is the voltage of the power supply to the memory system.

$$P_{read,j}(a) = \sum_{i=1}^{|P_a|} \beta_{aij} \cdot d_i \cdot p_{ri} \cdot I_{r,j}(f) \cdot V \tag{2.6}$$

$$P_{desel,j}(a) = \left(\alpha_j - \sum_{i=1}^{|P_a|} \beta_{aij} \cdot d_i \cdot p_{ri} \right) \cdot I_{d,j}(f) \cdot V \tag{2.7}$$

$$P_{stdby,j} = \alpha_j \cdot I_{s,j} \cdot V \tag{2.8}$$

In this model, $P_j(a)$ is regarded as the unit power to calculate the total power consumed by all memories for all applications. The average power consumption P_{avg}, also includes the contribution of the interconnect. Let P_F denotes the function that outputs the power required by the interconnect. Then, P_{avg} is computed according to Equation 2.9.

$$P_{avg} = P_F(\sum_{i=1}^{|M|} \alpha_i) + \frac{1}{|A|} \sum_{a=1}^{|A|} \sum_{j=1}^{|M|} P_j(a) \tag{2.9}$$

2.2 Heuristics

There are a lot of existing heuristics. At the early time of heuristics' usage, a certain algorithm is applied to solve one particular optimization problem. These problem-dependent heuristics can not be adapted to other optimization processes. To improve the portabilities of heuristics, some algorithms are invented as parameterized interfaces that can be widely deployed for a variety of optimization problems. Such problem-independent heuristics usually are consist of a base framework with several parameters. Only the parameters are related to the optimization problems. When using one of those heuristics for different problem sets, the algorithm framework is common while the parameters should be set up according to the problem requirements. In the recent years, there is a new trend of heuristic which is called hyper-heuristic. The hyper-heuristics provide a high-level strategy to seek one or several low-level heuristics to generate a proper algorithm for solving an optimization problem. The hyper-heuristic is a cutting-edge technique and it is beyond the knowledge of this work. For the memory power optimization, the problem-independent heuristics are mainly focused because of their extensive usage.

There are a variety ways to classify the heuristics. One common classification is to differentiate the algorithms according to their searching mechanisms. To be simplified, the heuristics are divided as local search-based and non-local search-based in this work. The well known local search algorithm aims to seek for the optimal solution by iteratively moving to a better solution in the neighborhood. However, the local search algorithm is greedy and cannot guarantee providing the good enough solutions because it may trap in local optimums. The idea of local search-based heuristics is to avoid the local optimum trap through some criteria for the solution selection and improve the result's quality. Heuristics of this kind output only one single optimal solution. Some classical local search-based heuristics are simulated annealing, tabu search, guided local search, etc. Unlike local search-based heuristics, the non-local search-based heuristics usually seek for a set of good enough solutions. By manipulating some defined solution characteristics, it can guide the searching process to the global optimal region. Some typical non-local search-based heuristics are genetic algorithm, particle swarm optimization, ant colony optimization, etc. Normally, the frameworks of non-local search-based heuristics are more complicated than that of local search-based algorithms. And the expected result for memory power optimization is one optimal configuration not a set of configurations. Therefor, the local search-based heuristics are the main focuses in this work. In this section, the local search algorithm along with its local optimum trap is discussed first. Then, two typical local search-based heuristics, tabu search and simulated annealing, are represented. Lastly, the most promising algorithm is proposed to the memory power optimization according to the comparison between these heuristics.

2.2.1 Local search algorithm

Local search algorithm is one of the simplest heuristics. Given an optimization problem, it starts from an initial solution and searches in the current solution's neighborhood. If a better solution is found, the current solution is replaced by it. The searching process is repeated until there is no better solution in the current solution's neighborhood. Then it outputs the current solution as the algorithm result.

Algorithm 2.2.1: Local Search Algorithm

```
Input: an optimization problem
Output: an optimal solution
1 current solution = initial solution;
2 while not terminate do
3     generate a neighboring solution;
4     evaluate the neighboring solution;
5     if neighboring solution is better than current solution then
6         current solution = neighboring solution;
7 output current solution;
```

Algorithm 2.2.1 shows the pseudo-code of local search process. There are four main steps in the algorithm. First step is finding an initial solution and setting it as the current solution. The initial solution should be valid for the optimization problem. In the second step, a neighboring solution is generated by certain mechanisms. And the third step is to compare the neighboring solution with the current one through an object function. The object function is a method to indicate how good the solution is. The last step is the selection criterion for solutions. Local search algorithm selects the better one between the current and the neighboring solution, which is a naive criterion.

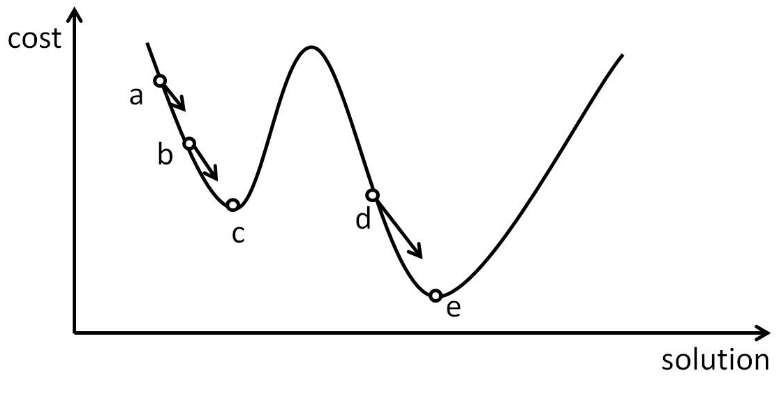


Figure 2.1: Local Optimum Trap

Though the local search algorithm is simple, the solution it provides may be the local optimal one. This is the major problem of the local search algorithm. Figure 2.1 illustrates the local optimum trap. Suppose the optimization problem is to find the solution with minimum cost, the local search algorithm starts with the initial solution a. The cost of the neighboring solution b is lower than the cost of a, then b is selected and becomes the current solution. The same searching process is repeated until the current solution reaches c. There is no better solution in c's neighborhood, thus the algorithm outputs solution c and terminates. However, solution c is only the local optimum while the global optimum is solution e which is not in c's neighborhood. In order to reach solution e, the algorithm has to move to solution d whose cost is higher than c's cost. And this violates the solution selection criterion of the algorithm. Another drawback of the local search algorithm is that the result quality is dependent on the initial solution. If the algorithm starts with solution d, the output will be the global optimum e. These two disadvantages make the local search algorithm an improper choice when global optimal solution is required for the optimization problems.

2.2.2 Tabu search algorithm

One of the improvements to the local search is the tabu search algorithm. It is based on the local search but it avoids to be stuck at the local optimal trap through a different selection strategy for solutions. As discussed in section 2.2.1, once the local search algorithm is trapped at a local optimal solution, it can not move any further due to the naive solution selection criterion. To solve this problem, Fred Glover proposes the concepts of the tabu list and the aspiration criterion in [Glo89] and [Glo90]. The following discussion is based on Fred Glover's proposal.

The key element of the tabu search algorithm is the tabu list. It imitates the memory function of human brain to guide the searching process. It is used to record the tabu objects. The tabu objects can be defined as the solutions, solution movements or values of the object function. The tabu list has a limited size which is one of the algorithm parameters. The improvement to the local search algorithm is gained from the solution selection strategy which is usually called the tabu move. There are two rules in the tabu move. The first rule is to exclude the solutions recorded in the tabu list from a set of neighboring solutions. The second rule is to select the best in the rest of the neighboring solution set. The solution chosen by the tabu move is set as the current solution. Another concept of the tabu search algorithm is the aspiration criterion During the searching process, the best-so-far solution is kept recorded in the searching history. The aspiration criterion is to examine the neighboring solution set to find out if there are solutions that are better than the current best-so-far solution. If such solutions are found, then the best of them is selected and is set as the current solution even if it is recorded in the tabu list. If no such solutions is found, the algorithm continues with the tabu move.

Algorithm 2.2.2 is the pseudo-code of the tabu search framework. At the beginning of the algorithm, it generates a valid initial solution and sets it as the current solution and the best-so-for solution. The parameters are set up according to the algorithm inputs. And a tabu list is created as empty. After the initialization, the algorithm generates a set of neighboring solutions by some certain mechanisms and evaluates it by an object function. After this step, there are two different branches. One branch is the execution of

Algorithm 2.2.2: Tabu Search Algorithm

```
Input: an optimization problem, algorithm parameters
Output: an optimal solution
1  set algorithm parameters;
2  current solution = initial solution;
3  best-so-far solution = initial solution;
4  set tabu list as empty;
5  while not terminate do
6      generate a set of neighboring solutions;
7      evaluate neighboring solutions;
8      if aspiration criterion satisfied then
9          execute aspiration criterion;
10         update current solution;
11         update tabu list;
12         update best-so-far solution;
13     else
14         execute tabu move;
15         update current solution;
16         update tabu list;
17 return current solution;
```

aspiration criterion. If the condition of the criterion is satisfied, the solution selected by the criterion is set as the current solution and the best-so-far solution. Also, the updated current solution is added to the tabu list. The other branch, tabu move, is executed when the condition of the aspiration criterion is not satisfied. The current solution and the tabu list are updated to the solution chosen by the tabu move while the best-so-far solution is not updated because there is no solutions better than it. There are two cases for the tabu list updating. At the early stage of the algorithm, the list is not full. The solutions are added into the list sequentially. When there is no space for a new recorded solution, the oldest solution in the list is replaced by the new one. The same searching process is repeated until the termination condition is satisfied and the algorithm outputs the current or the best-so-far solution as the optimization result. Some algorithm parameters are related to the termination condition. One simple method to terminate the algorithm is setting a fixed iteration number. Thus this fixed number is one parameter of the algorithm. However, this method can not guarantee the solution quality. Another common termination mechanism is to count the appearances of a particular solution. If this solution appears for a max number of time, the algorithm can terminate. Therefor, the max number is also one algorithm parameter.

Figure 2.2 illustrates how the tabu search algorithm avoids the local optimum trap. The solid line represents the execution of the aspiration criterion and the dotted line is the tabu move. The same problem set in section 2.2.1 is used. The tabu search algorithm starts from the initial solution a with an empty tabu list (assume the list size is large enough). After the neighboring solutions are generated, it finds solution b satisfies the aspiration criterion. Then b becomes the current solution and the best-so-far solution is updated to

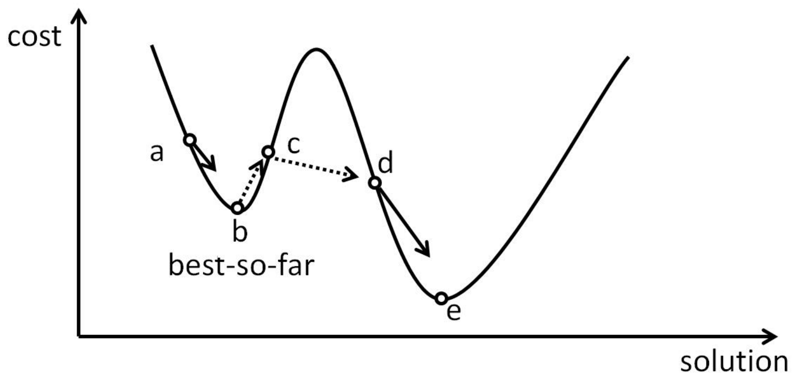


Figure 2.2: Tabu Search's Avoidance to Local Optimum Trap

b as well. Also, *b* is added to the tabu list. Known from the figure, solution *b* is a local optimum and there is no neighboring solution satisfies the aspiration criterion. Thus, the algorithm continues with the tabu move. During the tabu move, solution *b* is excluded from the generated neighboring solution set because it is stored in the tabu list. And solution *c* is found to be the best among the rests of the set. Thus, *c* becomes the current solution and it is added to the tabu list. In the next iteration, solution *b* and *d* both are *c*'s neighboring solutions. However, *b* is still in the tabu list and the aspiration criterion is not satisfied. Thus, the solution *d* is selected by the tabu move as it is the best among the rest of neighboring solutions. The same searching process is repeated until the algorithm reaches solution *e* which is the global optimum. By selecting a worser solution in the tabu move, the tabu search algorithm can move to a new searching region and avoid the local optimum trap.

2.2.3 Simulated annealing algorithm

Another enhancement to the local search is the simulated annealing algorithm. The idea of this algorithm is to imitate the metal annealing process. Three steps are performed in the annealing process. Firstly, the metal is melted at a very high temperature. The second step is to give a small disturbance to the metal and wait until the metal reaches its equilibrium state at the current temperature level. Then in the third step, the metal is cooled down slowly. The last two steps are repeated until a low temperature limit is reached at which the metal is in the ground state. Inspired by this process, the simulated annealing algorithm is proposed in [S K83]. And in the algorithm, the metropolis criterion is introduced to avoid the local optimum trap. The following discussion is based on the proposal of S. Kirkpatrick et al. in [S K83].

The metropolis criterion is a probabilistic technique to choose the solutions. In the local search algorithm, the better neighboring solutions are always selected and it is not possible to accept a worser one. In the simulated annealing algorithm, the metropolis criterion accepts the neighboring solution according to a probability which is related to a control parameter. The control parameter is the imitation of the temperature in the metal annealing precess. To be clarified, the control parameter is referred as temperature in the rest of this document.

$$p = \begin{cases} 1 & \text{, if } s_{next} \text{ is better than } s_{current} \\ \exp\left(-\frac{f(s_{next})-f(s_{current})}{t}\right) & \text{, otherwise} \end{cases} \tag{2.10}$$

Equation 2.10 represents the computation of the acceptance probability. In the equation, s_{next} is the next neighboring solution of the current solution $s_{current}$. p is the probability to accept s_{next}. And $f()$ is the object function. The value of $f()$ is referred as the solution cost. t is the temperature. It is can be seen from the equation that if s_{next} is better than $s_{current}$, the metropolis criterion behaves like the local search algorithm. And if s_{next} is worser, it can still be accepted depending on the computed probability. One thing is noticed in [S K83] is that the difference between costs of s_{next} and $s_{current}$ should be a positive value. Thus in the second case of the equation, the acceptance probability becomes smaller with the decrease of the temperature level.

The basic idea of simulated annealing algorithm is to combine the local search with the imitation of the metal annealing process. And the metropolis criterion is used to improve the solution quality. The algorithm searches better solutions in the current solution's neighborhood at different temperature levels. The searching process at the same temperature is repeated until certain termination condition is satisfied. The temperature is controlled to be reduced step by step as the same cooling procedure executed in the metal annealing process. At the early stage of the algorithm, the current solution is updated randomly because a lot of worser solutions are accepted due to the high temperature. However, with the decrease of the temperature, the probability to select worser solutions becomes smaller. This makes the searching space closer to the optimal region.

Algorithm 2.2.3 shows the pseudo-code of the simulated annealing process. In the preparing stage, the algorithm sets up the parameters according to the algorithm inputs. And the initial solution is set as the current solution. The temperature t is set up to a high enough value so that the random solutions can be selected by the metropolis criterion. The main framework of the simulated annealing can be divided into two nested loops. The outer loop is just the reduction of the temperature. The inner loop is the execution of the local search with the metropolis criterion. Firstly, the neighboring solution is generated by certain mechanisms and evaluated by the object function. Then the current solution is updated according to the result from the metropolis criterion. The inner and outer loop terminate when some predefined conditions are satisfied.

Figure 2.3 illustrates how the simulated annealing algorithm can avoid the local optimum trap. The solid line represents the acceptance of better solution while the dotted line is the acceptance of worser solution. The same problem set in Section 2.2.1 is used. Suppose the current solution is a at a proper temperature level. The algorithm finds out that the neighboring solution b is better. Through the metropolis criterion, the current solution is updated to b which is the local optimum. In the following iteration, the neighboring solution c with higher cost is generated. Because of the proper temperature, it is possible to accept c with the computation of the acceptance probability. And the same searching process is repeated until the global optimal solution e is reached. It is still possible that the

Algorithm 2.2.3: Simulated Annealing Algorithm

```
Data: an optimization problem, algorithm parameters
Result: an optimal solution
1  set algorithm parameters;
2  current solution = initial solution;
3  t = initial tempreture;
4  while not terminate do
5      while not terminate do
6          generate a neighboring solution;
7          evaluate the neighboring solution;
8          if neighboring solution is better then
9              accept neighboring solution;
10         else
11             comput the accept probability;
12             accept neighboring solution according to the accept probability;
13         update current solution;
14     decrease t;
15 return current solution;
```

algorithm moves away from e at the current temperature. However, the current temperature is assumed to be proper. And it applies that even if the global solution e is discarded by the metropolis criterion, the selected solution can not be much worser than e. If the algorithm terminates at this current temperature, a near optimal solution can be obtained. And if the algorithm continues with the decrease of the temperature and a low enough temperature is reached, it can move back to e agian and a worser solution can not be selected.

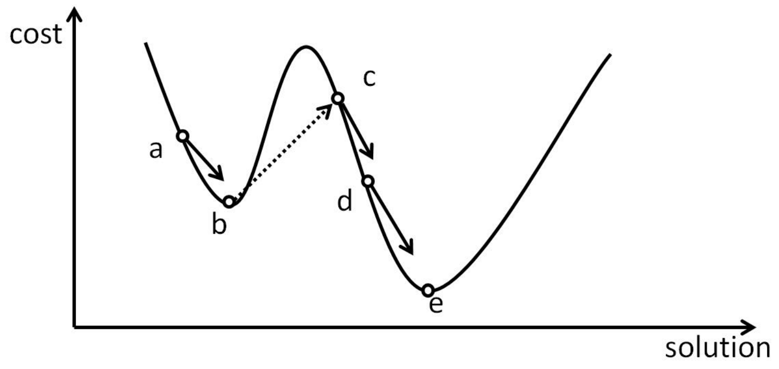


Figure 2.3: Simulated Annealing's Avoidance to Local Optimum Trap

2.3 Heuristics selection

Though the discussed heuristics in Section 2.2 seem to be the potential approaches for the memory power optimization problem. However, each of them has its own advantages and disadvantages.

The local search algorithm is simple and it can be easily adapted to the optimization process. But it can not guarantee the solution quality even for the near optimal one because of the local optimum trap. Thus, the local search algorithm is not taken into consideration according to the optimization goal.

The tabu search algorithm uses the tabu list as the memory structure to guide the searching process. By forbidding the recorded solutions, it can exclude the previously searched space to avoid the searching repetitions. As discussed in Section 2.2.2, once the algorithm traps in the local optimum, worser solutions can be selected by the tabu move. And the aspiration criterion helps the algorithm move to the new searching spaces. However, the tabu list size is the essential parameter and it can affect the solution quality and the algorithm performance. If the list size is too small, the algorithm may be trapped in a searching loop. In this case, the algorithm will stick to the local optimum trap. Because of the small tabu list size, the local optimal solution is recorded for only a short period and it is released before the algorithm can move to a better searching space. If the list size is too large, the searching time will be quite long and it may be not acceptable in practice. Besides the tabu list size, the solution quality of the algorithm is also dependent on the initial solution. And selecting a proper initial solution is a non-trivial work. Another drawback of the tabu search algorithm is the dead lock problem. In some extreme cases, there is no solution satisfies the aspiration criterion in the current solution's neighborhood. And all the generated neighboring solutions are recorded in the tabu list. Then no solution will be selected and the algorithm can not continue.

As discussed in Section 2.2.3, the simulated annealing algorithm can deal with the local optimum trap properly. And it is not sensitive to the initial solution. Because at the early stage of the algorithm, the metropolis criterion selects the solution randomly due to the high temperature. Even if the initial solution is already good enough, the algorithm may move far away from it. With the slow decrease of the temperature, the algorithm moves to the optimal searching region step by step. Nevertheless, the initial temperature is one significant parameter of the simulated annealing. It is supposed to be high enough to make the metropolis criterion accept solutions randomly. If it is too low, the algorithm behaves similarly to the local search algorithm and it may be trapped in local optimal solutions. If it is too high, an amount of time is wasted for the random searches. Another factor affects the solution quality of the simulated annealing is the cooling schedule which is reduction of the temperature. If temperature decreases too fast, the behavior of the local search occurs much earlier before the global optimal region is reached. If it decreases too slowly, the searching time will be quite long. Also, the termination conditions related to the nested loops play an important role in the simulated annealing algorithm. If the inner loop terminates too soon, the searching space is not fully explored. And if the outer loop ends too early, the acceptance probability is rather high. The final solution may be not good enough. If both loops terminate too late, the algorithm may not be efficient enough for the optimization problem.

Compared with the tabu search algorithm, there are more parameters should be set up very carefully in the simulated annealing algorithm. But there is a variety of existing mechanisms to adjust them properly. Finding a suitable initial solution for the tabu search algorithm will increase the complexity of the optimization process. And it may require the pre-analysis of the solution space. Furthermore, the dead lock problem of the tabu search algorithm may result in the algorithm failure, which is risky for the users of this algorithm. After taking all the aspects into consideration, the simulated annealing algorithm is more promising than the tabu search algorithm. And it is proposed for the memory power optimization with memory partitioning method. At this stage, the fist goal of this thesis work is achieved.

3 Related Work

The memory partitioning method is widely used for the memory power optimization problem. Many researchers have obtained good results and benefits through the usage of this approach [SER16; BMP00; Mai+07]. The detail development of the mathematic power model discussed in Section 2.1 is illustrated in [SER16]. In this article, M. Strobel et al. propose the the ILP approach to be used in the optimization of the memory partitioning. And the results from their experiments show that the usage of ILP can yield an optimal configuration for their problem set. Since the memory partitioning is one kind of the combinatorial optimization, heuristics can also be a potential approach for it.

The combinatorial optimization is one of the hottest topics which aims to search for an optimal solution in a finite solution space. Most of the conventional methods such as exhaustive search is not a proper approach for it. S. Kirkpatrick et al. find that there existing a close relationship between the statistical mechanics and the combinatorial optimization [S K83]. Based on this finding, they propose the simulated annealing algorithm as a promising approach for the combinatorial optimization in [S K83]. They introduce the metropolis criterion and explain how it can be applied to improve the solution quality provided by local search algorithm. In the article, the simulated annealing process is developed to be consisted of a parameterized framework which is already discussed in Section 2.2.3. To illustrate the usability of the simulated annealing algorithm, S. Kirkpatrick et al. deploy the algorithm in the physical design of computers to optimize the circuits partitioning, placement and wiring processes. In addition, they also apply the simulated annealing algorithm to the classical traveling salesmen problem. Multiple experiments for above optimization problems are conducted in this article and the experiment results are used to show that good solutions can be obtained by the usage of the simulated annealing algorithm. It is also pointed out by S. Kirkpatrick et al. that the accuracy and efficiency of the algorithm are much dependent on its parameters. Their suggestions for the algorithm setting are discussed together with other's work later in this chapter.

In [Joh+89], the author implement the simulated annealing algorithm for the graph partitioning problem. And a deep evaluation for the algorithm is made through a compact series of experiments in the article. They propose an alternative for the design of the cooling schedule and compared it with the original method using in [S K83]. The cooling schedule implemented by S. Kirkpatrick et al. is to reduce the temperature linearly by a colling ration. In [Joh+89], the author develop an adaptive cooling schedule which slows down the temperature reduction when the solution cost is changing fast in the current searching region. However, no improvement to the linear cooling schedule is found in their conducted experiments. For the terminations of the nested loops, they terminate the inner loop when the maximum number of iterations are achieved. And a low threshold of the acceptance probability is used for the termination of the outer loop. Unfortunately, there is no method proposed to the determination of the initial temperature in [Joh+89].

In [S K83], S. Kirkpatrick et al. suggest two methods to determine the initial temperation T_0. The first one is to use the maximum difference between two neighboring solution costs as the initial temperature. The other one is described as following. An initial acceptance probability P_0 is defined which is the expected acceptance probability at T_0. Its value should be a real number close but less than 1, typically in the range of 0.8 to 0.95. Then a random guess of T_0 is made and the inner loop of simulated annealing is performed at this temperature. The solution acceptance ration of the inner loop performance is measured. And if the acceptance ration is lower than P_0, the value of T_0 is doubled. The same process is repeated until the measured acceptance ration is higher than P_0. Based on the second suggest of S. Kirkpatrick et al., Johnson et al. provide Equation 3.1 in [Joh+91] for the estimation of the initial temperature.

$$T_0 = -\frac{\overline{\Delta E}}{\ln P_0} \tag{3.1}$$

The $\overline{\Delta E}$ in the equation is the average difference between two neighboring solution costs. The value of $\overline{\Delta E}$ can be measured by generating a set of positive solution acceptances. Another determination is proposed by the authors of [Whi84]. They estimate the initial temperature T_0 by using Equation 3.2, where "K is a fixed number in the range of 5 to 10 and σ_∞^2 is the second moment of the cost distribution when the temperature is infinite "[Ben04]. However, this approach requires the analysis base on the pre-knowledge of the solution cost distribution.

$$T_0 = K\sigma_\infty^2 \tag{3.2}$$

4 Simulated Annealing for Memory Power Optimization

In this chapter, the main contents of the thesis work are represented. Figure 4.1 shows the process of the memory power optimization using the simulated annealing algorithm. There are two kinds of input for the simulated annealing algorithm. One is the input data related to the optimization problem. These raw data is recorded in different text files. Their data structure is not suitable for the simulate annealing algorithm. Thus, a proper input data organization is defined and a parsing method is used to transform the raw data into this data organization. Section 4.1 discusses the input data organization and the parsing method in detail. The other kind of input for the algorithm is the algorithm parameter that is discussed together with the algorithm design. Section 4.2 introduces an abstract design flow of the simulated annealing. Section 4.3 to Section 4.5 discuss the design details and problems of three different approaches for the memory power optimization.

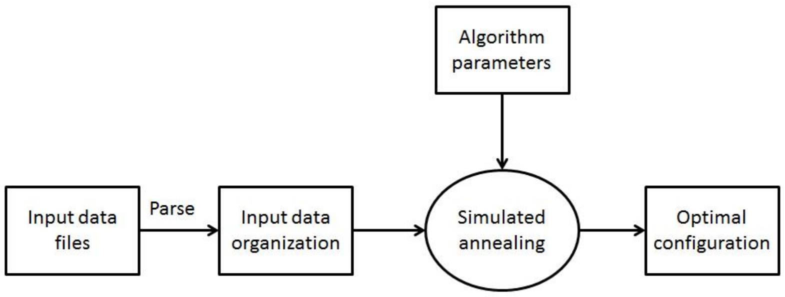


Figure 4.1: Process of The Simulated Annealing for Memory Power Optimization

4.1 Input data organization

As discussed in Section 2.1, the formal power model requires the parameters that are relevant to the memory types, the application profiles and the interconnect. Thus, these parameters are the input data to the simulated annealing algorithm. Since the object oriented programming is planned to be used for the implementation of the simulated annealing, the parameters that are related to the same type can be grouped together into one class. For example, the physical parameters that are relevant to memory types can reside in the memory class. Figure 4.2 shows the input parameters organization in the form of the UML class diagram.

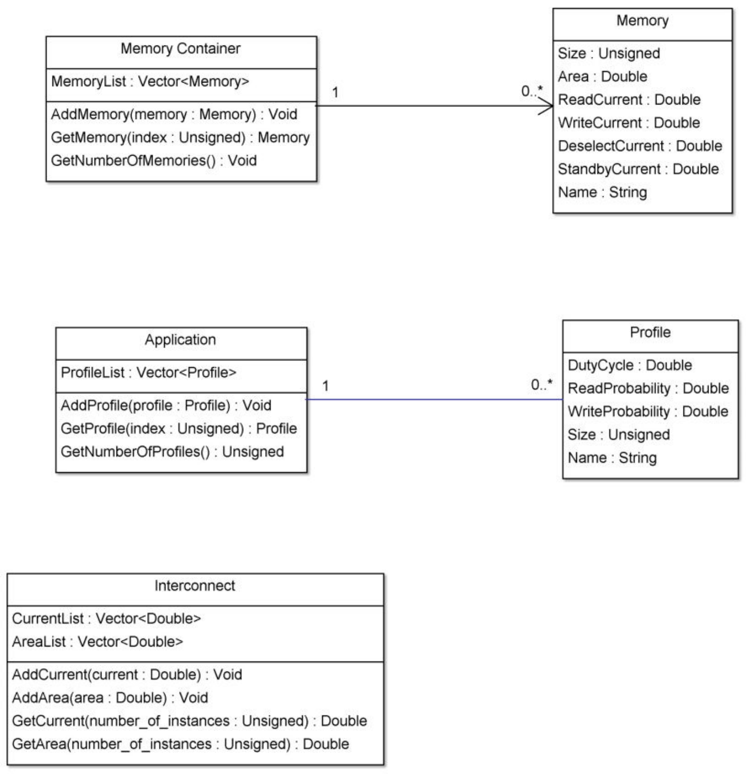
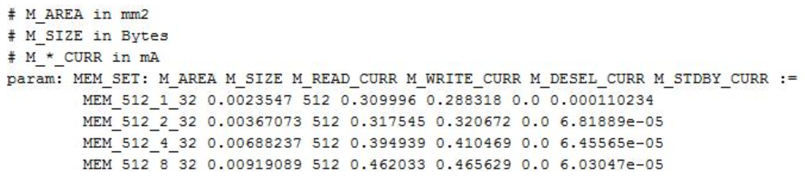

Memory Container
MemoryList : Vector<Memory>
AddMemory(memory : Memory) : Void
GetMemory(index : Unsigned) : Memory
GetNumberOfMemories() : Void
1
0..*
Memory
Size : Unsigned
Area : Double
ReadCurrent : Double
WriteCurrent : Double
DeselectCurrent : Double
StandbyCurrent : Double
Name : String
Application
ProfileList : Vector<Profile>
AddProfile(profile : Profile) : Void
GetProfile(index : Unsigned) : Profile
GetNumberOfProfiles() : Unsigned
1
0..*
Profile
DutyCycle : Double
ReadProbability : Double
WriteProbability : Double
Size : Unsigned
Name : String
Interconnect
CurrentList : Vector<Double>
AreaList : Vector<Double>
AddCurrent(current : Double) : Void
AddArea(area : Double) : Void
GetCurrent(number_of_instances : Unsigned) : Double
GetArea(number_of_instances : Unsigned) : Double

Figure 4.2: Input Parameter Organization in UML Diagram

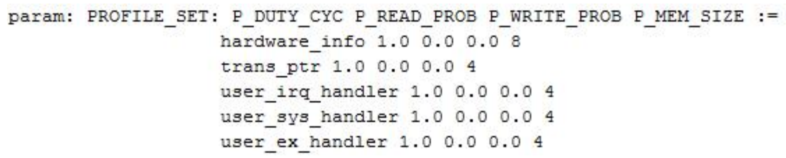
The memory class includes all the physical parameters that are used in the formal power model. Since there are a set of memory types in the data file, a set of objects of the memory class will be created in the simulated annealing algorithm as well. Thus the memory container class is defined to store memory objects. The algorithm can also retrieve the required objects and the total number of memory objects through the corresponding operations in the memory container class. The profile class and the application class are defined similarly to the memory class and the memory container class respectively. The interconnect class is different. It uses two separated lists to store the current and area parameters whose value are dependent on the total number of instances. These parameters can also be retrieved by the simulated annealing algorithm through the corresponding operations defined in the interconnect class.

In this thesis work, the same data sets provided by the authors of [SER16] are used as the input parameter data of the simulated annealing algorithm. They are recorded in multiple plain text data files. Figure 4.3 represents some fragments of these data files. Figure 4.3a is the fragment of the memory data file. Figure 4.3b shows the fragment of the application profile data file and Figure 4.3c is the fragment of the interconnect data file. In order to group this data into the designed input data organization, a parsing method is used. The basic idea of the parsing method is to read the plain text file line by line. The required data is extracted and the unnecessary data is discarded. Since the data file is text-based,



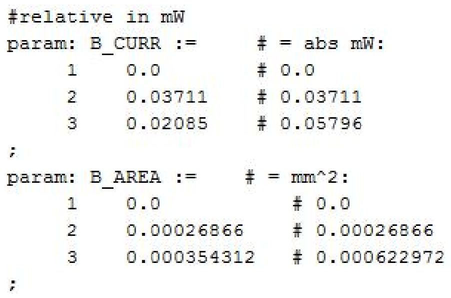
```
# M_AREA in mm2
# M_SIZE in Bytes
# M_*_CURR in mA
param: MEM_SET: M_AREA M_SIZE M_READ_CURR M_WRITE_CURR M_DESEL_CURR M_STDBY_CURR :=
        MEM_512_1_32 0.0023547 512 0.309996 0.288318 0.0 0.000110234
        MEM_512_2_32 0.00367073 512 0.317545 0.320672 0.0 6.81889e-05
        MEM_512_4_32 0.00688237 512 0.394939 0.410469 0.0 6.45565e-05
        MEM 512 8 32 0.00919089 512 0.462033 0.465629 0.0 6.03047e-05
```

(a)



```
param: PROFILE_SET: P_DUTY_CYC P_READ_PROB P_WRITE_PROB P_MEM_SIZE :=
                    hardware_info 1.0 0.0 0.0 8
                    trans_ptr 1.0 0.0 0.0 4
                    user_irq_handler 1.0 0.0 0.0 4
                    user_sys_handler 1.0 0.0 0.0 4
                    user_ex_handler 1.0 0.0 0.0 4
```

(b)



```
#relative in mW
param: B_CURR :=     # = abs mW:
     1    0.0        # 0.0
     2    0.03711    # 0.03711
     3    0.02085    # 0.05796
;
param: B_AREA :=    # = mm^2:
     1    0.0            # 0.0
     2    0.00026866     # 0.00026866
     3    0.000354312    # 0.000622972
;
```

(c)

Figure 4.3: Fragments of The Input Parameter Data Files [SER16]

the required data are converted into the corresponding data type during the extraction.

```
Algorithm 4.1.1: Parse Memory\Profile Data File
Input: a memory\profile data file
Output: a memory container\application object
1 create a memory container\application object;
2 while not end of the file do
3     read one line;
4     if relevant data is contaied in the line then
5         extract the data;
6         convert the extracted data into the corresponding type;
7         create a memory\profile object based on the converted data;
8         add the created object into the list of the memroy container\application object;
9 return the memory container\application object;
```

Algorithm 4.1.1 is the pseudo-code of the parsing method for the memory and profile data files. If the method is used to parse the memory data file, it first reads one line of the file text. Then it checks whether there is the relevant data contained in the line or not. If there is no data, the the algorithm continues reading the next line. Otherwise, the relevant data is extracted and converted into the corresponding data type. It can be seen from the file fragment in Figure 4.3a, all the parameters data related to one memory type are listed in one single line. Thus, a memory object can be created based on the parsed data for one line. Then the created object is added to the list in the memory container object. The same parsing step is repeated until the method reaches the end of the file. The parsing process of the application profile data file is similar to the parsing of the memory data file. The only differences are that it deals with the profile data file and profile objects are created and added to the list of the application object. However, the parsing method of the interconnect data file is modified slightly. Algorithm 4.1.2 shows the pseudo-code of the parsing method for the interconnect data file. From the file fragment in Figure 4.3c it can be seen that there are two parts in the interconnect data file. The first part contains the interconnect current data while the second part records the interconnect area data. Thus, after the extracted data is converted, the parsing method needs to check whether the data is related to the current or the area. Then the data is added to the corresponding list of the interconnect object.

Algorithm 4.1.2: Parse Interconnect Data File

```
Input: an interconnect data file
Output: an interconnect object
1  create an interconnect object;
2  while not end of the file do
3      read one line;
4      if relevant data is contaied in the line then
5          extract the data;
6          convert the extracted data into the corresponding type;
7          if is current data then
8              add the converted data into the current list of the interconnect object;
9          else
10             add the converted data into the area list in of interconnect object;
11 return the interconnect object;
```

4.2 Abstract SA design flow

The major design of the simulated annealing for the memory power optimization is the algorithm structure. Before the design details are introduced, an abstract design flow of the simulated annealing algorithm is provided in Figure 4.4. From the figure it can be seen that the algorithm is consisted of three parts. In the initialization of the algorithm, S_{curr} is the current solution. It is set to an initial solution S_0. C_{curr} is the cost of S_{curr} and it is calculated according to S_0. The temperature T is set to an initial value T_0. The second part of the algorithm is the inner loop. S_{neigh} is the neighboring solution generated by the method $Neighbor()$ based on S_{curr}. Its cost C_{neigh} is calculated by the cost function $Cost()$. Then the metropolis criterion $Metropolis()$ is performed to update S_{curr} according to C_{curr} and C_{neigh}. $Termination_{inner}()$ is the termination mechanism for the inner loop.

The last part of the algorithm is the outer loop. $CoolingSchedule()$ is used to decrease T. $Termination_{outer}()$ is the termination mechanism for the outer loop.

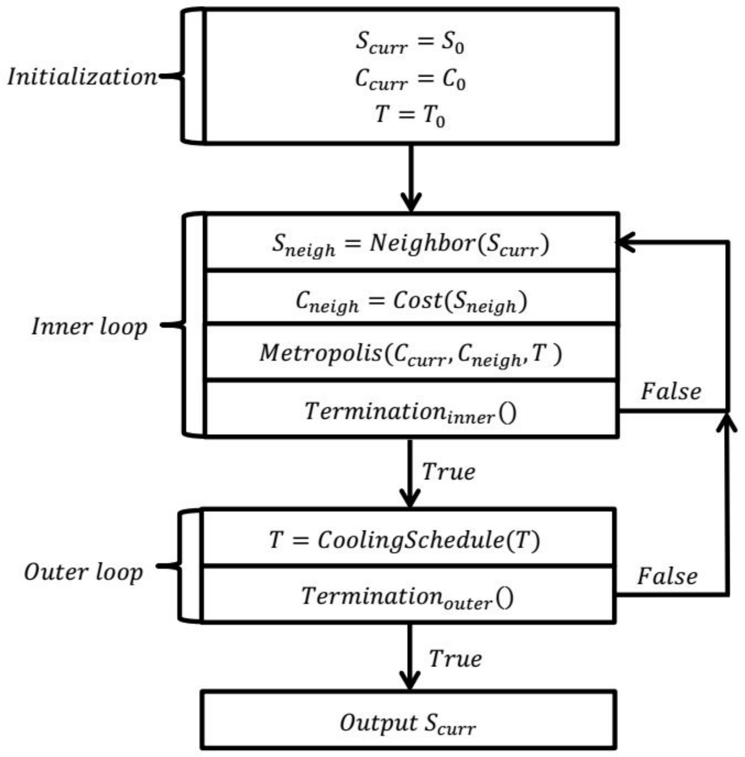


Figure 4.4: Simulate Annealing Design Flow

The design of the simulated annealing should contains the answers of the following Questions 1 to 7. To enhance the readability of the rest of this chapter, the design details will answer these questions sequentially. By providing a solution for each of these questions, the design for the simulated annealing algorithm is completed.

Question 1: What is the representation of solutions?

Question 2: How to generate neighboring solutions?

Question 3: What is the cost function?

Question 4: How to implement the metropolis criterion?

Question 5: What is the cooling schedule?

Question 6: How to determinate the initial solution and the initial temperature?

Question 7: How to terminate the inner loop and the outer loop?

4.3 Allocation and binding in single SA

This approach is to seek for the optimal memory allocation and the binding of profiles simultaneously in a single simulated annealing algorithm. Section 4.3.1 gives the answers of the design questions and Section 4.3.2 discusses the problems of this approach.

4.3.1 Design process

Question 1: What is the representation of solutions?

As discussed in Section 2.1, the result of the memory partitioning process is a configuration for the memory system. The solution provided by the simulated annealing algorithm should correspond to the memory configuration. According to this, solutions of the simulated annealing algorithm are definedas following. A solution is in the form of an integer vector that are divided into two parts. The first part of the vector corresponds to the allocation of memories. Each element in this part is the number of memory instances of the corresponding memory type. The second part of the vector contains the information about the binding of profiles. Each element in this part is the index of the memory type to which the corresponding profile is bound. Therefore, the vector length is the sum of the number of memory types and the number of profiles. To be clear, the first part of the vector is called the allocation part of the solution and the second part of the vector is called the binding part of the solution. The vector length is called the solution size. Figure 4.5 shows a solution example with two memory types and three application profiles. In the allocation part of the solution, one instance is allocated for both memory types. In the binding part, profile 0 is bound to memory type 0. Profile 1 and 2 are both bound to memory type 1.

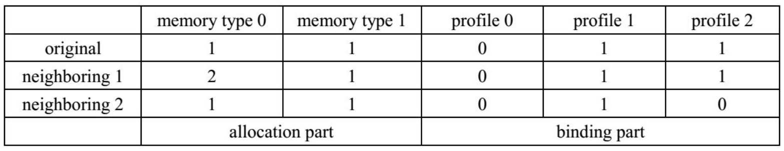
memory type 0	memory type 1	profile 0	profile 1	profile 2
1	1	0	1	1
allocation part		binding part		

Figure 4.5: A Solution Example

Question 2: How to generate neighboring solutions?

According to the solution representation, the neighboring solutions are described as following. A neighboring solution is a vector with same length of the original solution. Only one element in the neighboring solution is different from the original solution. Figure 4.6 shows an example of neighboring solutions. The original solution is the same in Figure 4.5. In neighboring solution 1, two memory instances are allocated for memory type 0 while only one instance is allocated in the original solution. The rest elements of neighboring solution 1 are the same with the original solution. In neighboring solution 2, the only difference to the original solution is that profile 2 is bound to memory type 0.

In this approach, the basic idea of $Neighbor()$ is to modify one element from the current solution. When generating neighboring solutions, the constraints of the formal power model



	memory type 0	memory type 1	profile 0	profile 1	profile 2
original	1	1	0	1	1
neighboring 1	2	1	0	1	1
neighboring 2	1	1	0	1	0
	allocation part		binding part		

Figure 4.6: Neighboring Solutions Example

Algorithm 4.3.1: *Neighbor*()

```
Input: $S_{curr}$
Output: $S_{neigh}$
1 create a new solution $S_{neigh}$;
2 while $ConstraintsCheck(S_{neigh})$ is false do
3     $S_{neigh} = S_{curr}$;
4     randomly selecte an element $element_i$ in $S_{neigh}$;
5     if $element_i$ is in the allocation part then
6         $element_i$ = Rondom number in $Range_\alpha$;
7     else
8         $element_i$ = Rondom number in $Range_\beta$;
9 return $S_{neigh}$ ;
```

should be taken into consideration. To ensure the constrain 1 and 3, the modification of the chosen element is limited in a range. As the solution is consisted of two different parts, the ranges of these two parts are also different. Let $Range_\alpha$ and $Range_\beta$ denote the modification ranges for the allocation part and the binding part respectively. The lower bound of $Range_\alpha$ is 0. The upper bound of $Range_\alpha$ is computed as following. Let $mems_{total}$ denote the total number of memory instances in the current solution. Suppose element i in the allocation part is selected to be modified. Let mem_i denotes the number of instances of memory type i in the current solution. Then the upper bound of $Range_\alpha$ equals $(mems_{max} - mems_{total} + mem_i)$. $mems_{max}$ is the predefined constraint in the Equation 2.1. For $Range_\beta$, its lower bound is 0. Let $Num_{memtype}$ denote the total number of memory types in the memory set. The upper bound of $Range_\beta$ is $(Num_{memtype} - 1)$.

Algorithm 4.3.1 is the pseudo-code of $Neighbor()$. In the algorithm, there is another method $ConstraintsCheck()$. This method is used to examine whether the generated neighboring solutions satisfy the constrain 2 (Equation 2.2) and constraint 4 (Equation 2.4) of the power model. Algorithm 4.3.2 shows the pseudo-code of $ConstraintsCheck()$. It first checks whether the area constraint $area_{max}$ is satisfied or not. $area_{total}$ is the total area consumed by the allocated memory instances and the interconnect. Then for each memory type, $ConstraintsCheck()$ examines whether enough memory space is provided for the profiles that are bound to the memory type.

Question 3: What is the cost function?

Because the solution of the simulated annealing algorithm corresponds to the memory configuration, the solution cost C is defined as the average power consumption of the configuration. The cost function $Cost()$ is designed based on the formal power model that

Algorithm 4.3.2: *ConstraintsCheck*()

```
Input: $S_{neigh}$
Output: a boolean value
compute $area_{total}$;
if $area_{total} > area_{max}$ then
    return false;
else
    foreach memory type $mt_i$ do
        find all profiles bound to $mt_i$;
        compute the memory space $space_{required,mt_i}$ required by these profiles;
        compute the memory space $space_{provided,mt_i}$ provided by $mt_i$;
        if $space_{provided,mt_i} < space_{required,mt_i}$ then
            return false;
    return true;
```

is introduced in Section 2.1. All the relevant parameter data required by the power model can be fetched from the input parameter organization.

Question 4: How to implement the metropolis criterion?

The procedure of the metropolis criterion is straightforward and its implementation is illustrated in Algorithm 4.3.3. Because the goal of the power memory optimization is to reduce the power consumption, the solution with lower cost is considered as the better one in the metropolis criterion. A random real number R is generated by the method *Random*(). This method only generates real numbers that are uniformly distributed in the range of 0 to 1 due to that R is compared with the acceptance probability P_{accept}.

Algorithm 4.3.3: Metropolis Criterion Precedure

```
Input: $C_{curr}, C_{neigh}, T$
Output: void
if $C_{curr} \leq C_{neigh}$ then
    $S_{curr} = S_{neigh}$;
    $C_{curr} = C_{neigh}$;
else
    $P_{accept} = \exp\left(-\frac{C_{neigh} - C_{curr}}{T}\right)$;
    $R = Random()$;
    if $R \leq P_{accept}$ then
        $S_{curr} = S_{neigh}$;
        $C_{curr} = C_{neigh}$;
```

Question 5: What is the cooling schedule?

For the cooling schedule, the temperature T is linearly reduced with a fixed cooling ration R_{cool}. Equation 4.1 is the design of *CoolingSchedule*(), where N is the number of outer loop iterations. The value of the R_{cool} should be a positive real number and it must be less than 1 in order to decrease T. However, it is a non-trivial task to determine a proper value for R_{cool} In this approach, the value of R_{cool} is set to 0.9 first and it can be adjusted base on experiments.

$$T_{N+1} = R_{cool} \cdot T_N \tag{4.1}$$

Question 6: How to determinate the initial solution and the initial temperature?

The initial solution S_0 of the simulated annealing algorithm can be set to a given solution manually. In this approach, S_0 is set as following. Only one larger enough memory instance is allocated so that all application profiles can be bound to it.

For the determination of the initial temperature T_0, the initial acceptance probability P_0 is defined. As Kirkpatrick et al. propose in the original article, P_0 is the expected acceptance probability of worser solutions at T_0 [S K83]. In this approach, T_0 is computed by Equation 4.2 based on the metropolis criterion (Equation 2.10). In the equation, $(C_{neigh} - C_{curr})_{max}$ is the maximum cost difference between a worser neighboring solution and the current solution. It can be measured by generating a set of worser neighboring solutions of the current solution randomly. The value of P_0 should be a positive real number less than 1. It should be close to 1 because the neighboring solutions are randomly accepted at T_0. P_0 is set to 0.9 in this approach and it can be adjusted based on experiments.

$$T_0 = -\frac{(C_{neigh} - C_{curr})_{max}}{lnP_0} \tag{4.2}$$

Question 7: How to terminate the inner loop and the outer loop?

For the inner loop of the simulated annealing algorithm, the number of iterations is limited to a maximum number $Max_{iteration}$. During the execution of the inner loop, the number of iterations $Num_{iteration}$ is counted. When $Num_{iteration}$ becomes larger than $Max_{iteration}$, the inner loop terminates. The value of $Max_{iteration}$ is related to the size of the neighborhood $Size_{neighbor}$. $Size_{neighbor}$ is defined as the number of neighboring solutions of the current solution. In this approach, it is fixed to the solution size for simplification. Equation 4.3 illustrates how to calculate the value of $Max_{iteration}$. From the equation it can be seen that $Max_{iteration}$ is proportional to $Size_{neighbor}$ with a factor $F_{iteration}$. The value of $F_{iteration}$ should be a positive real number. In this approach, $F_{iteration}$ is set to be 1 and it can also be modified based on experiments.

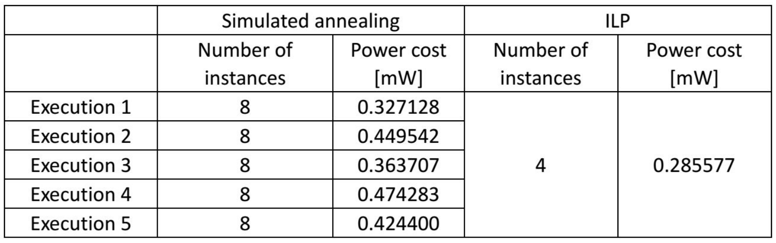
$$Max_{iteration} = F_{iteration} \cdot Size_{neighbor} \tag{4.3}$$

For the outer loop termination, a low temperature limit T_{low} is defined. When the temperature T is decreased to T_{low}, the outer loop terminates. The determination of T_{low} is similar to the determination of T_0. P_{low} is defined as the expected acceptance probability at T_{low}. Then T_{low} is computed according to Equation 4.4. In the equation, $(C_{neigh} - C_{curr})_{min}$ is the minimum cost difference between a worser neighboring solution and the current solution. It can be measured by generating a set of worser neighboring solutions randomly as well. The value of P_{low} should be a positive real number close to 0 because the simulated annealing algorithm behaves like the local search algorithm at T_{low}. The worser neighboring solutions are seldom accepted. In this approach, the value of P_{low} is set to 0.1 and it can be adjusted base on experiments.

$$T_{low} = -\frac{(C_{neigh} - C_{curr})_{min}}{lnP_{low}} \tag{4.4}$$

4.3.2 Problem discussion

For the evaluation of the design approach, the input data and the ILP results provided by the authors of [SER16] are used. The input data includes the parameters of 79 different memory types and the parameters of the interconnect. There are 4 applications, namely $IPreassembly$, $IPcheck$, $MD5$ and $Huffman$ [SER16]. Each application has RAM and ROM modes. The input data also contains the profile details for each of the applications. In this approach, the following testing scenario is designed. Parameter data of all 79 memory types along with the interconnect data are input to the simulated annealing algorithm. The application $IPreassembly$ in ROM mode with its 241 profiles are used. The algorithm parameters are set according to the discussion in Section 4.3.1. The maximum number of instances $mems_{max}$ is limited to 8. There is no area constraint. $area_{max}$ is set to a large enough value. Figure 4.7 shows the results of the designed simulated annealing algorithm and they are compared with the ILP results. After analyzing the algorithm execution process and its results, the following problems are found.



	Simulated annealing		ILP	
	Number of instances	Power cost [mW]	Number of instances	Power cost [mW]
Execution 1	8	0.327128		
Execution 2	8	0.449542		
Execution 3	8	0.363707	4	0.285577
Execution 4	8	0.474283		
Execution 5	8	0.424400		

Figure 4.7: Results Comparison Between Simulated Annealing and ILP

Problem 1: Solutions always reaches $mems_{max}$.

From Figure 4.7 it can be seen that every solution provided by the designed simulated annealing algorithm contains 8 memory instances. This is due to the $ConstraintsCheck()$ in $Neighbor()$. When S_{neigh} is generated by modifying an element in the allocation part of S_{curr}, $ConstraintsCheck()$ makes the probability of increasing the element higher than the probability of decreasing the element. Increasing the element satisfies the constraints because it allocates more memory instance than required by the profiles. However, decreasing the element may violate the constraints because it reduces the number of memory instances. This may result in that the memory type does not have enough memory space for the profiles.

Another phenomenon found in the algorithm behavior is that the searching process will stick to a very small region once the solution allocates 8 memory instances. The reason of this phenomenon is described as following. Firstly, a memory type is called allocated if it has at least one memory instance. Let $memtypes_{allocated}$ denote the set of allocated memory types in S_{curr}. Let $memtypes$ denote the set of all memory types provided by the input data. According to the testing scenario, $|memtypes_{allocated}| \leq 8$, and $|memtypes| = 79$. Suppose S_{curr} already contains 8 memory instances. S_{neigh} is generated by modifying an element from the allocation part of S_{curr}. The probability of selecting one allocated memory type in S_{curr} is $\frac{|memtypes_{allocated}|}{|memtypes|}$. In the best case, $|memtypes_{allocated}| = 8$. The probability is $\frac{8}{79} \approx 0.1$. This means that most of the time, $Neighbor()$ selects a memory type that is not allocated. It can neither increase nor decrease the element because there are already 8 memory instances allocated. Even if it is lucky to select an allocated memory type, the modification range $Range_{\alpha}$ of the element is rather small. For this reason, the searching process sticks to a small neighborhood of S_{curr}.

Problem 2: Generating neighboring solutions is time consuming.

After monitoring the execution time for different parts of the algorithm, it is found that the large fraction of the execution time is consumed by $Neighbor()$. It is still because of the $ConstraintsCheck()$ in $Neighbor()$. When generating S_{neigh} by modifying an element in the binding part of S_{curr}, the probability of changing the selected element to the index of an allocated memory type is still calculated by $\frac{|memtypes_{allocated}|}{|memtypes|}$. In the testing scenario, it is $\frac{8}{79} \approx 0.1$ for the best case. This means that most of the time, the selected profile is bound to a memory type that is not allocated. Even if the profile is lucky to be bound to an allocated memory type, it may violate the constraints as well because the allocated memory type may have no enough space for it. Besides, Problem 1 also contributes to the time consumption. These two problems make the overall execution time of the designed simulated algorithm rather long.

Problem 3: No convergence for the power cost.

In Figure 4.7, the results of the simulated annealing is different for each execution. Because in each execution, the searching process sticks to different small region due to Problem 1. This is similar to the local optimum trap. Besides, the quality of the binding part is not guaranteed as well.

The above problems make the designed simulated annealing algorithm an improper approach for the memory power optimization. However, in the next section, a nested simulated annealing structure is introduced to solve these problems.

4.4 Allocation and binding in nested SA

From the approach discussed in Section 4.3, finding the optimal allocation and the binding at the same time seems to be problematical. In this approach, the memory partitioning is divided into two related sub-processes. One is the optimization for the memory allocation and the other one is the optimization for the profile binding. The allocation optimization is dependent on the result of the binding optimization. The simulated annealing algorithm is used for both sub-processes. To be simplified, let SA_α denote the simulated annealing algorithm for the allocation optimization and let SA_β denote the simulated annealing algorithm for the binding optimization. Then the basic idea of this approach is to use SA_β as the cost function of SA_α. Section 4.4.1 answers the seven design questions. Section 4.4.2 discusses the problems of this approach.

4.4.1 Design process

Question 1: What is the representation of solutions?

In the approach discussed in Section 4.3, the solution for the simulated annealing algorithm is consisted of the allocation part and the binding part. In this approach, the allocation part is defined as the solution of SA_α while the binding part is defined as the solution of SA_β. To be simplified, let S_α and S_β denote the solutions of SA_α and SA_β respectively.

Question 2: How to generate neighboring solutions?

Let $Neighbor_\alpha$ and $Neighbor_\beta$ denote the neighboring solution generating method of SA_α and SA_β respectively. Instead of modifying the chosen element according to a limited range, $Neighbor_\alpha$ increases or decreases the element by one. The probabilities of the increase and decrease are both 0.5. But if the original value of the chosen element is 0, it can only be increased because the element should be positive. Algorithm 4.4.1 is the pseudo-code for $Neighbor_\alpha()$. In the algorithm, $S_{\alpha,curr}$ is the current solution of SA_α and $S_{\alpha,neigh}$ is its neighboring solution. Similar to the Algorithm 4.3.1, the method $ConstraintsCheck_\alpha()$ is used to check whether $S_{\alpha,neigh}$ satisfies the constraints or not. In this approach, the constraints of the power model are divided to two groups as well. Specially, a two-step implementation of the constraint 4 verification is made in this approach. Let $space_{provided}$ denotes the memory spaces provided by all allocated memory types in the current solution. Let $space_{required}$ denote the memory space required by all profiles. Then the first step of constraint 4 verification is to examine whether $space_{provided}$ is less than $space_{required}$ or not. This step is called constraint 4-1 verification. The second step is the same procedure of $ConstraintsCheck()$ (Algorithm 4.3.2) discussed in Section 4.3.1. But it excludes the checking for area constraint. This step is called constraint 4-2 verification. Then constraint 1, 2 and 4-1 are grouped together and they are called $Constraint_\alpha$. Constraint 3 and 4-2 are called $Constraint_\beta$. Algorithm 4.4.2 shows the pseudo-code for $ConstraintsCheck_\alpha()$. In the algorithm, $mems_{total}$ is the total number of memory instances allocated in $S_{\alpha,neigh}$ and $area_{total}$ is the total area consumed by the memory instances and the interconnect. $mems_max$ and $area_max$ are the predefined constraints in the formal power model.

For SA_β, $Neighbor_\beta()$ is more complicated than $Neighbor_\alpha()$. It is based on the solution of SA_α. The basic idea of $Neighbor_\beta()$ is to randomly select one profile and bind it to another allocated memory type. To make the $Neighbor_\beta()$ clear, an example is given in Figure 4.8. From the figure it can be seen that there are three memory types in S_α and there are three profiles in S_β. The allocated memory types are memory type 0 and 2 because they have at

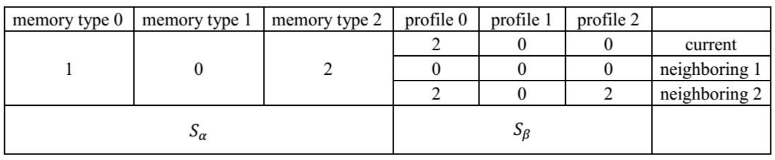
Algorithm 4.4.1: $Neighbor_\alpha()$

```
Input: $S_{\alpha,curr}$
Output: $S_{\alpha,neigh}$
1 create a new solution $S_{\alpha,neigh}$;
2 while $ConstraintsCheck_\alpha(S_{\alpha,neigh})$ is false do
3     $S_{\alpha,neigh} = S_{\alpha,curr}$;
4     randomly selecte an element $element_i$ in $S_{\alpha,neigh}$;
5     increase or decrease $element_i$ by 1;
6 return $S_{\alpha,neigh}$ ;
```

Algorithm 4.4.2: $ConstraintsCheck_\alpha()$

```
Input: $S_{\alpha,neigh}$
Output: a boolean value
1 compute $mems_{total}$;
2 compute $area_{total}$;
3 compute $space_{provided}$;
4 compute $space_{required}$;
5 if $mems_{total} \leq mems_{max}$ & $area_{total} \leq area_{max}$ & $space_{provided} \geq space_{required}$ then
6     return true;
7 else
8     return false;
```

least one instances. In the current solution of SA_β, profile 0 is bound to memory type 2. When generating neighboring solution 1, profile 0 is chosen and it is changed to be bound to memory type 0. In neighboring solution 2, profile 2 is changed to be bound to memory type 2 as well. The selected profiles can not be bound to memory type 1 because it is not allocated.



memory type 0	memory type 1	memory type 2	profile 0	profile 1	profile 2	
1	0	2	2	0	0	current
			0	0	0	neighboring 1
			2	0	2	neighboring 2
S_α			S_β			

Figure 4.8: Neighboring Solutions Example for SA_β

Algorithm 4.4.3 shows the pseudo-code of $Neighbor_\beta()$. In the algorithm, $memtypes_{allocated}$ is the set of allocated memory types in S_α. $S_{\beta,curr}$ and $S_{\beta,neigh}$ are the current and neighboring solutions of SA_β. The method $ConstraintsCheck_\beta$ is also used to check whether $S_{\beta,neigh}$ satisfy the $Constraint_\beta$. It is same to Algorithm 4.3.2 without the area constraint verification. Through binding the chosen profile to an allocated memory type, many invalid neighboring solutions are avoided. This reduces the execution time consumed by $Neighbor_\beta()$.

Algorithm 4.4.3: $Neighbor_\beta()$

```
Input: $S_\alpha, S_{\beta,curr}$
Output: $S_{\beta,neigh}$
1 create a new solution $S_{\beta,neigh}$;
2 while $ConstraintsCheck_\beta(S_{\beta,neigh})$ is false do
3     $S_{\beta,neigh} = S_{\beta,curr}$;
4     randomly selecte an element $element_i$ in $S_{\beta,neigh}$;
5     identify the set of allocated memory types $memtypes_{allocated}$ in $S_\alpha$;
6     randomly selecte an memory type $memtype_j$ in $memtypes_{allocated}$;
7     $element_i$ = index of $memtype_j$ in $S_\alpha$;
8 return $S_{\beta,neigh}$ ;
```

Question 3: What is the cost function?

In SA_β, the solution cost is still the memory power consumption that can be calculated according to the power model. The cost function is the same one introduced in the first approach.

In SA_α, there is no real cost for the its solution S_α because S_α only contains the information about the memory allocation. Thus, the memory power consumption can not be calculated according to the power model. However, the cost function of SA_α evokes the procedure of SA_β. Then SA_β optimizes the binding based on the current solution of SA_α. After SA_β terminates, it provides a binding with the minimum memory power consumption to the cost function of SA_α. SA_α sets the minimum power consumption as its solution cost. Figure 4.9 shows the nested simulated annealing procedure. In the figure, C_α and C_β are the solution cost for SA_α and SA_β respectively.

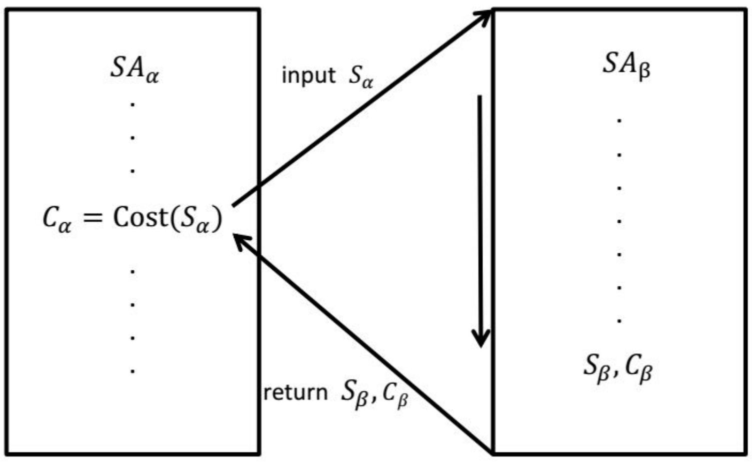


Figure 4.9: Nested Simulated Annealing Procedure

Question 4: How to implement the metropolis criterion?

In this approach, the metropolis criterion of both SA_α and SA_β are implemented the same with the first approach.

Question 5: What is the cooling schedule?

The same cooling schedule (Equation 4.1) introduced in the first approach is used for both SA_α and SA_β.

Question 6: How to determinate the initial solution and the initial temperature?

Both SA_α and SA_β use the same mechanism introduced in the first approach.

Question 7: How to terminate the inner loop and the outer loop?

For the inner loop termination, Equation 4.3 is still used. However, the neighborhood sizes of SA_α and SA_β are described as following. The neighborhood size $Size_{neighbor,\alpha}$ is calculated according to Equation 4.5. The length of S_α is actually the total number of memory types in the memory set.

$$Size_{neighbor,\alpha} = 2 \cdot \text{length of } S_\alpha \tag{4.5}$$

Because SA_β is embedded in SA_α, $Size_{neighbor,\beta}$ is based on the solution of SA_α. It is calculated according to Equation 4.6. In the equation, $|memtypes_{allocated}|$ is the number of allocated memory types in the current solution of SA_α. The length of S_β is actually the total number of profiles of the application.

$$Size_{neighbor,\beta} = |memtypes_{allocated}| \cdot \text{length of } S_\beta \tag{4.6}$$

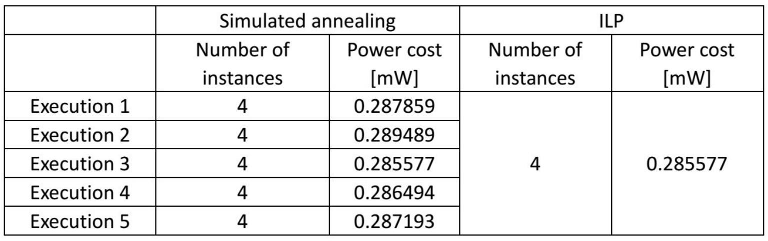
For the outer loop termination, T_{low} is still used for both SA_α and SA_β. The determination of T_{low} is the same discussed in the first approach.

4.4.2 Problem discussion

The implementation of the nest simulated annealing algorithm is divided to two steps. The fist step is to implement SA_β with a given valid memory allocation. For the evaluation of SA_β, the same testing scenario introduced in approach 1 is used. The best memory allocation provided by the ILP approach is given to SA_β. The results of SA_β are shown in Figure 4.10 and they are compared with the ILP results. From the figure, it can be seen that SA_β can yield the optimal or the near optimal of solutions under the precondition that the optimal solution of SA_α is given. However, the following problems are found during the implementation of SA_α.

Problem 1: The execution time of the nest simulated annealing is unacceptable in practice.

Because SA_α and SA_β is nested, the overall execution time $t_{overall}$ can be calculated according to Equation 4.7. In the equation, $iteration_{out,\alpha}$ and $iteration_{in,\alpha}$ are the number of the outer loop iterations and the number of the inner loop iterations in SA_α. t_β is the execution time of SA_β. According to Equation 4.3 and Equation 4.5, $iteration_{in,\alpha}$ equals to $2 \cdot$ length of S_α $\cdot F_{iteration}$. Based on the testing scenario, length of $S_\alpha = 79$



	Simulated annealing		ILP	
	Number of instances	Power cost [mW]	Number of instances	Power cost [mW]
Execution 1	4	0.287859		
Execution 2	4	0.289489		
Execution 3	4	0.285577	4	0.285577
Execution 4	4	0.286494		
Execution 5	4	0.287193		

Figure 4.10: Results Comparison Between SA_β and ILP

and $F_{iteration} = 1$. Thus, $iteration_{in,\alpha}$ equals to 158. The measured average t_β is around 30 seconds. Assume $iteration_{out,\alpha} = 1$, then $t_{overall} = 1 \cdot 158 \cdot 30 = 4740$ seconds ≈ 1.32 hours. This is the overall execution time when the outer loop of SA_α is performed only once. It is already an unexpected time for the optimization process. However, the average $iteration_{out,\alpha}$ in the testing scenario is always greater than 50. Even if t_β is decreased by optimizing SA_β from programming side, $t_{overall}$ is still unacceptable in practice. The cost function in the simulated annealing algorithm should be the mathematical computations that can be executed very fast by computers. However, the cost function in SA_α is a simulated annealing algorithm in this approach. This is the reason of this problem.

$$t_{overall} = iteration_{out,\alpha} \cdot iteration_{in,\alpha} \cdot t_\beta \tag{4.7}$$

Problem 2: Invalid S_α is provided to SA_β.

The verification of constraint 4-1 can not guarantee the validity of S_α. In $ConstraintsCheck_\alpha$ (Algorithm 4.4.2), the verification of constraint 4-1 only compares $space_{provided}$ and $space_{required}$. However, satisfying constraint 4-1 does not ensure S_α is valid. There is one fact ignored by constraint 4-1. The fact is that one profile can not be distributed to different memory types. Consider the following situation. There are two memory types provided, namely M_{512} and M_{256}. The subscripts of their names are the memory sizes. Suppose a profile P with size 700 needs to be bound to the memory type and only 2 instances are allowed. Figure 4.11 shows the different combinations of the memory instances and their validities according to constraint 4-1 and the original constraint 4. In the case that one instance is allocated for both memory types, the allocation is valid according to constraint 4-1 because $space_{provided} > space_{required}$. However, this allocation is not valid according to the original constraint 4 due to the fact that profiles can not be distributed to multiple memory types.

Because of the above problems, this approach only obtains the success for SA_β. Due to the unacceptable execution time, the evaluation of SA_α is not performed. In the next section, an enhancement of the nested simulated annealing algorithm is introduced.

4.5 Allocation and binding in independent SAs

This approach is to solve the problems of nested simulated annealing algorithm discussed in Section 4.4. The basic idea of this approach is still using SA_α and SA_β to optimize

M_{256}	M_{512}	$space_{provided}$	$space_{required}$	Constraint 4-1	Constraint 4
0	2	1024	700	valid	valid
1	1	768		valid	invalid
2	0	512		invalid	invalid

Figure 4.11: Example For Generating Invalid S_α

the memory allocation and the profile binding respectively. However, SA_α and SA_β are independent from each other.

Instead of evoking SA_β, SA_α in this approach calculates the memory power consumption in its cost function. But the power consumption can not be calculated with only the memory allocation. To solve this problem, the method *SortedBind*() is used to provide an assumed binding information to SA_α. Another purpose of *SortedBind*() is to avoid generating invalid S_α due to constraint 4-1 verification. Algorithm 4.5.1 shows the pseudo-code of *SortedBind*(). In the algorithm, d is the profile duty cycle. p_r and p_w are the profile read and write probability respectively. The free space of a memory type is the memory space that is not be occupied. The details of *SortedBind*() and its basis are introduced later in Section 4.5.1.

Algorithm 4.5.1: *SortedBind*()

```
Input: $S_\alpha$,$S_\beta$
Output: a boolean value
1  sort memory types in ascending order according to size;
2  if ROM mode then
3      sort profiles in descending order according to $d \cdot p_r$;
4  else
5      sort profiles in descending order according to $d \cdot (p_r + p_w)$;
6  foreach $profile_i$ in $S_\beta$ do
7      in $S_\alpha$, find the smallest memory type $mt$ that has enough free space for $profile_i$;
8      bind $profile_i$ to $mt$;
9  if $S_\beta$ satisfies constraint 4 then
10     return true;
11 else
12     return false;
```

The common problem of the first two approaches is that the number of the allocated memory instances always reaches constraint $mems_{max}$. To avoid this problem, this approach introduces the level concept to the memory allocation optimization. In each allocation optimization level, only a fixed number of memory instances are allowed. Let $mems_{allowed}$ denote this fixed number. Then only the allocations with exact $mems_{allowed}$ instances are taken into consideration for a certain level. For example, suppose $mems_{allowed} = 4$ in an allocation optimization level. Then all the potential allocations contain exactly 4 memory instances. SA_α is performed for each optimization level to obtain the corresponding optimal memory allocation. Then SA_β is used to seek for the optimal profile binding of these allocations. Lastly, the pair of allocation and binding with minimum power cost is output as the final result of the optimization process. Figure 4.12 shows an allocation optimal level structure with the constraint $mems_{max} = 4$.

This approach mainly focus on the design of SA_α and the enhancement for SA_β because SA_β can already yield good results. In the following, the seven design questions are answered in Section 4.5.1. Section 4.5.2 discusses the findings and problems of this approach.

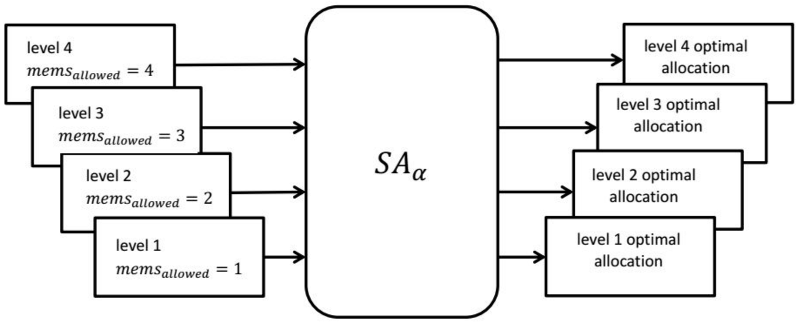


Figure 4.12: Allocation Optimization Levels with $mems_{max} = 4$

4.5.1 Design process

Question 1: What is the representation of solutions?

In this approach, the representation of solutions for both SA_α and SA_β are the same with the nested simulated annealing algorithm.

Question 2: How to generate the neighboring solutions?

Because the number of allocated memory instances is fixed for solutions in SA_α, the method of generating S_α should follow this rule. There are two steps in $Neighbor_\alpha()$. The first step is to select one allocated memory type in $S_{\alpha,curr}$ and decrease its number of instances by one. The second step is to choose an arbitrary memory type in $S_{\alpha,curr}$ and increase its number of instances by one. Algorithm 4.5.2 shows the pseudo-code of $Neighbor_\alpha()$. In the algorithm, $ConstraintsCheck_\alpha()$ is the same one in the nested simulated annealing algorithm. $SortedBind()$ is added to the check condition to guarantee the validity of $SA_{\alpha,neigh}$.

For SA_β, neighboring solutions are generated by the same mechanism used in the nested simulated annealing (Algorithm 4.4.3).

Question 3: What is the cost function?

Algorithm 4.5.2: *$Neighbor_{\alpha}()$*

```
Input: $S_{\alpha,curr}$, $S_{\beta}$
Output: $S_{\alpha,neigh}$
1 create a $S_{\alpha,neigh}$;
2 while ($ConstraintsCheck_{alpha}(S_{\alpha,neigh})$ & $SortedBind(S_{\alpha,neigh},S_{\beta})$) is false do
3     $S_{\alpha,neigh} = S_{\alpha,curr}$;
4     select an allocated memory type $mt_{allocated}$ in $S_{\alpha,curr}$;
5     select an arbitrary memory type $mt_{arbitrary}$ in $S_{\alpha,curr}$;
6     delete an instance for $mt_{allocated}$ in $S_{\alpha,neigh}$;
7     add an instance for $mt_{arbitrary}$ in $S_{\alpha,neigh}$;
8 return $S_{\alpha,neigh}$;
```

In this approach, the costs of both S_{α} and S_{β} are the memory power consumption. They can be calculated based on the formal power model. The cost function of SA_{β} is the same one in the nested simulated annealing algorithm. However, the cost function of SA_{α} needs the binding information provided by *SortedBind*() in order to calculate the memory power cost.

The following discussion is about the basis of *SortedBind*(). In order to evaluate the memory allocation provided by SA_{α}, the corresponding binding is required. But for a given allocation, there are lots of potential bindings of a set of application profiles. Different bindings yield different memory power costs. This will affect the evaluation results. Thus, a standard should be defined for selecting the bindings for the memory allocation. For the same set of application profiles, if the best profile binding is selected for all potential memory allocations, the evaluation becomes uniform. For defining the best binding, the following assumptions are made. The best application profile binding is obtained by mapping frequently accessed profiles to small memory types. The profile access frequency is dependent on the product of the profile duty cycle and the profile access probability. The profile access probability in ROM mode is the profile read probability. The profile access probability in RAM mode is the sum of profile read probability and write probability. These assumptions are made based on the principle of memory partitioning and the ILP results provided by the authors of [SER16]. In algorithm 4.5.1, *SortedBind*() sorts the memory types and the application profiles. Then it binds each profile to the memory types according to these assumptions. If all profiles are bound successfully, the quality of the memory allocation provided in SA_{α} can be evaluated based on this binding.

Question 4: How to implement the metropolis criterion?

The implementation of the metropolis criterion for both SA_{α} and SA_{β} are the same with the nested simulated annealing algorithm.

Question 5: What is the cooling schedule?

The same cooling schedule in the nested simulated annealing algorithm is used for both SA_{α} and SA_{β}.

Question 6: How to determinate the initial solution and the initial temperature?

In this approach, the initial solution for both SA_α and SA_β are generated randomly instead of being set manually. This is for the verification of the feature that simulated annealing is not sensitive to the initial solution.

To improve the accuracy of initial temperature T_0, a second step is add to the determination of T_0. After computing T_0 according Equation 4.2, the inner loop of the simulated annealing algorithm is performed once at T_0. During the execution of the inner loop, the acceptance probability is measured. Then the measured acceptance probability $P_{measured}$ is compared with the expected acceptance probability P_0. If $P_{measured}$ is smaller than P_0, T_0 is doubled. The same process is repeated until $P_{measured}$ is larger than P_0. It makes sure that T_0 is high enough so that random solutions can be accepted in the metropolis criterion.

Question 7: How to terminate the inner loop and the outer loop?

The inner loop termination mechanism of the nested simulated annealing algorithm is used for both SA_α and SA_β in this approach. However, $Size_{neigh,\alpha}$ is different because $Neighbor_\alpha()$ is modified in this approach. Equation 4.8 shows the computation for $Size_{neigh,\alpha}$ in this approach. $|memtypes_{allocated}|$ is the number of allocated memory types in the current solution of SA_α. The length of S_α is actually the total number of memory types in the memory set. The computation of $Size_{neigh,\beta}$ is still the same one of the nested simulated annealing because $Neighbor_\beta()$ is not changed in this approach.

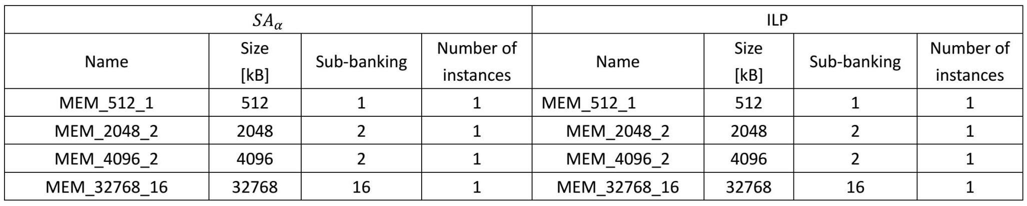
$$Size_{neigh,\alpha} = |memtypes_{allocated}| \cdot \text{ length of } S_\alpha \tag{4.8}$$

For the termination of the outer loop of simulated annealing algorithm, a new mechanism is introduced in this approach. During the execution of inner loop, the acceptance probability is measured. A low threshold $P_{threshold}$ for the acceptance probability is defined. If the measured acceptance probability is smaller than $P_{threshold}$ in five successive iterations, the outer loop terminates. The value of $P_{threshold}$ should be a small positive real number close to 0. In this approach, $P_{threshold}$ is set to 10^{-3} and it can be adjusted by experiments.

4.5.2 Problem discussion

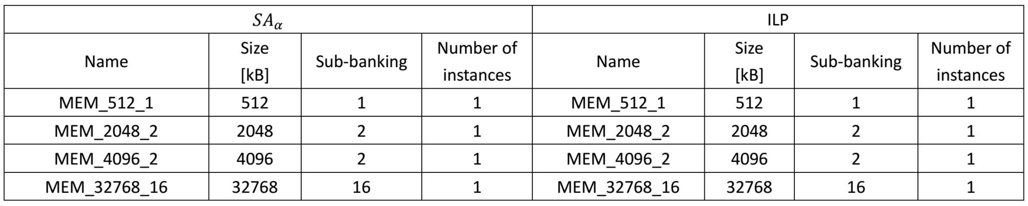
For the evaluation of this approach, the same testing scenario introduced in the previous approach is used. As discussed in the nested simulated annealing algorithm, SA_β can yield good enough profile bindings for a given memory allocation. Therefore, the functionality of SA_α is mainly focused in this approach. The evaluation details of SA_β are represented in Chapter 5.

To verify its functionality, SA_α is executed multiple times under the same testing scenario. Figure 4.13 shows the result. From the figure it can be seen that the memory allocation provided by SA_α is same with the allocation provided by ILP. However, SA_α is executed with only one testing scenario. To verify its functionality further, the same application in RAM mode is used. Figure 4.14 shows the memory allocations provided by SA_α and ILP. From the figure it is can be seen that the two memory allocations are different from each other. By analyzing the algorithm execution process and its results, the following problems are found.



SA_α				ILP			
Name	Size [kB]	Sub-banking	Number of instances	Name	Size [kB]	Sub-banking	Number of instances
MEM_512_1	512	1	1	MEM_512_1	512	1	1
MEM_2048_2	2048	2	1	MEM_2048_2	2048	2	1
MEM_4096_2	4096	2	1	MEM_4096_2	4096	2	1
MEM_32768_16	32768	16	1	MEM_32768_16	32768	16	1

Figure 4.13: Memory Allocation Comparison Between SA_α and ILP (ROM)



SA_α				ILP			
Name	Size [kB]	Sub-banking	Number of instances	Name	Size [kB]	Sub-banking	Number of instances
MEM_512_1	512	1	1	MEM_512_1	512	1	1
MEM_2048_2	2048	2	1	MEM_2048_2	2048	2	1
MEM_4096_2	4096	2	1	MEM_4096_2	4096	2	1
MEM_32768_16	32768	16	1	MEM_32768_16	32768	16	1

Figure 4.14: Memory Allocation Comparison Between SA_α and ILP (RAM)

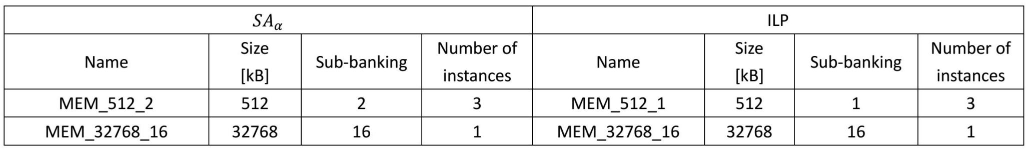
Problem 1: Lager memory types are always allocated.

After executing SA_α multiple times, it is found that SA_α always allocates instances for memory types with larger size. For example, in Figure 4.14, two instances are allocated for memory type with size 4194304 in SA_α, which is much larger than the memory type allocated in ILP. To illustrate this problem clearly, let $allocation_{SA}$ and $allocation_{ILP}$ denote the memory allocations provided by SA_α and ILP respective. The data in Figure 4.14 is used for $allocation_{SA}$ and $allocation_{ILP}$. Assume the current solution of SA_α is $allocation_{SA}$ and the searching process is guided towards to $allocation_{ILP}$. In order to reach $allocation_{ILP}$, the following steps should be performed according to $Neighbor_\alpha$. The first step is to delete one instance from the four allocated memory types in $allocation_{SA}$. The second step is to add one instance for memory type MEM_1048576_16. These two steps are repeated until eight instances are allocated for MEM_1048576_16. In the testing scenario, the total number of memory types is 79. The probability of selecting MEM_1048576_16 out of 79 memory types is $\frac{1}{79}$. In addition, MEM_1048576_16 should be selected eight times continuously. The probability of moving from $allocation_{SA}$ to $allocation_{ILP}$ is $\frac{1}{79}^{8} \approx 0$. This means that it is almost impossible to reach $allocation_{ILP}$ from $allocation_{SA}$. This problem is caused by the mechanism for generating neighboring solutions in SA_α.

After examining the input application profile file for RAM carefully, another phenomenon is found. There are some profiles with extremely large size. For example, there is a profile with size 7973948 in above testing scenario. In order to satisfy the constraints, at least one memory type with large enough memory space should be allocated. In Figure 4.14, both SA_α and ILP provide such memory type. Suppose $Neighbor_\alpha()$ selects MEM_4194304_16 and deletes one instance first. If an instance is added for a memory type with size smaller than 7973948, the generated neighboring solution becomes invalid. Because there is no memory type providing enough memory space required by the large profile. Thus, $Neighbor_\alpha()$ always adds instances for lager memory types.

Problem 2: Memories with same size but different sub-bankings can not be differentiated.

In the method $SortedBind()$, the memory types are sorted according to their sizes. However, their sub-banking parameters are not taken into consideration. This will affect the solution quality provided by SA_α. Figure **??** shows the memory allocations provided by SA_α and ILP for $Huffman$ application in ROM mode. From the figure, it is can be seen that SA_α allocates instances for MEM_512_2 while ILP allocates instances for MEM_512_1. Both memory types have the same size but their numbers of sub-bankings are different.



SA_α				ILP			
Name	Size [kB]	Sub-banking	Number of instances	Name	Size [kB]	Sub-banking	Number of instances
MEM_512_2	512	2	3	MEM_512_1	512	1	3
MEM_32768_16	32768	16	1	MEM_32768_16	32768	16	1

Figure 4.15: Memory Allocation Comparison Between SA_α and ILP (Huffman, RAM)

Problem 3: This approach depends on unproven assumptions.

The assumptions made for $SortedBind()$ in this approach are based on the principle and the observation of ILP results provided by the authors of [SER16]. Their usage should be verified scientifically.

5 Evaluation

Discussed in Chapter 2, using the simulated annealing algorithm to find an optimal memory allocation and the corresponding optimal application profile binding simultaneously is problematic. However, a partial success has been achieved. Given a memory allocation, the simulated annealing algorithm for binding SA_β yields an good enough profile binding for applications. This chapter presents the evaluation of SA_β. The evaluation scenarios are described as following. SA_β is implemented according to the design in approach 3 introduced in Section 4.5.1. According to this algorithm design, SA_β contains the following parameters:

- the threshold of the acceptance probability $P_{threshold}$;
- the cooling ration R_{cool};
- the expected acceptance probability P_0 at the initial temperature;
- factor $F_{iteration}$ that used for computing the maximum number of the inner loop iterations

In all evaluation scenarios, the parameters are set as following:

- $P_{threshold} = 0.001$;
- $P_{0=0.9}$;
- $R_{cool} = 0.9$;
- $F_{iteration} = 1$

The data sets and ILP resutls provided by the authors of [SER16] are used as the input for SA_β. The data sets contain the informations of following contents:

- A set of 79 memory types;
- A set of 4 applications, namely $IPreassembly$, $IPcheck$, $MD5$, $Huffman$ [SER16];
- 4 sets of profiles corresponding to the set of applications;
- An interconnect.

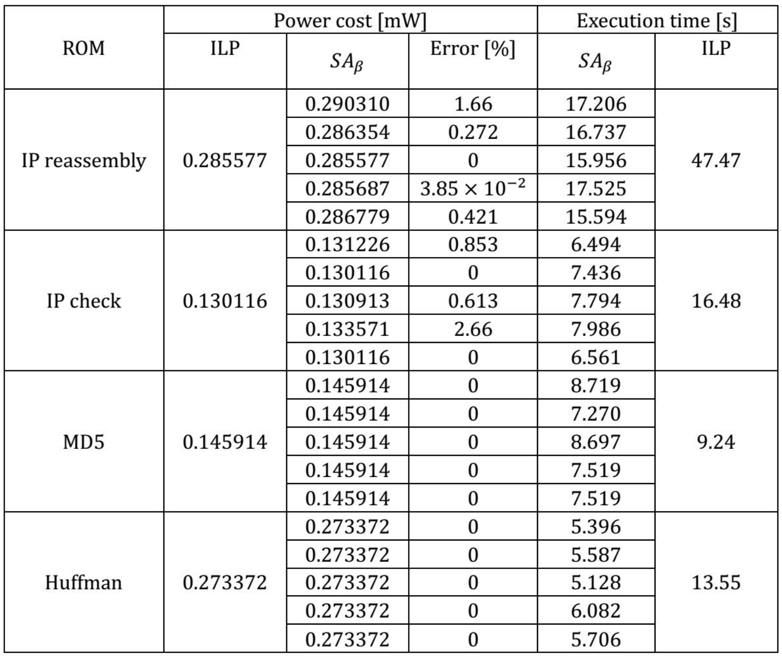
For each application, SA_β is performed individually on an $Intel^{®}$ $Core^{™}2$ $Quan$ CPU $Q9300$ system clock at 2.5 GHz and having 7.7 GB main memory. The corresponding optimal memory allocation provided by the ILP result is given to SA_β. In Section 5.1, the results of SA_β is compared with the results provided by ILP. Section 5.2 evaluates SA_β by adjusting the algorithm parameters.

5.1 Results comparison between SA_β and ILP

To verify its functionality, SA_β is performed for all applications. ROM and RAM modes are both taken into consideration. Due to the fact that SA_β is a random searching algorithm, its output may be different for each execution. According to this, SA_β is executed for five times for each application. Figure 5.1 and Figure 5.2 show the results for ROM mode and RAM mode respectively. In the figures, the ILP results are provided by the authors of [SER16] and they are used as the references for the SA_β evaluation. In the figures, the error percentage is defined by Equation 5.1, where P_{SA_β} and P_{ILP} are the power costs of the SA_β results and ILP results respectively.

$$Error\ percentage = \frac{P_{SA_\beta} - P_{ILP}}{P_{ILP}} \tag{5.1}$$

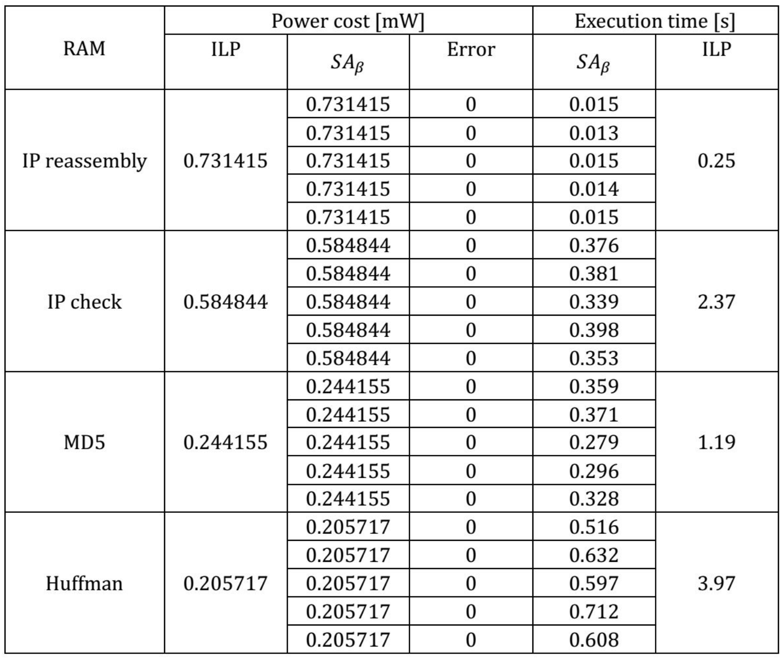
For the simplification of comparing SA_β results and ILP results, the power cost and the the error percentage are mainly focused. If the power costs P_{SA_β} obtained by SA_β is close enough to the power cost P_{ILP} provided by ILP, the binding output by SA_β is considered as good enough because ILP yields the optimal binding. More specifically, the bindings with power error percentage lower than 5% are considered to be near optimal. The details of bindings are not shown in the results.



ROM	Power cost [mW]			Execution time [s]	
	ILP	SA_β	Error [%]	SA_β	ILP
IP reassembly	0.285577	0.290310	1.66	17.206	47.47
		0.286354	0.272	16.737	
		0.285577	0	15.956	
		0.285687	3.85×10^{-2}	17.525	
		0.286779	0.421	15.594	
IP check	0.130116	0.131226	0.853	6.494	16.48
		0.130116	0	7.436	
		0.130913	0.613	7.794	
		0.133571	2.66	7.986	
		0.130116	0	6.561	
MD5	0.145914	0.145914	0	8.719	9.24
		0.145914	0	7.270	
		0.145914	0	8.697	
		0.145914	0	7.519	
		0.145914	0	7.519	
Huffman	0.273372	0.273372	0	5.396	13.55
		0.273372	0	5.587	
		0.273372	0	5.128	
		0.273372	0	6.082	
		0.273372	0	5.706	

Figure 5.1: Results Comparison between SA_β and ILP (ROM)

From the results of ROM mode shown in Figure 5.1, it is can be seen that for application MD5 and Huffman, P_{SA_β} are exactly the same with P_{ILP} in each execution of SA_β. For application IPreassembly anb IPcheck, P_{SA_β} is slightly different from P_{ILP} in some



RAM	Power cost [mW]			Execution time [s]	
	ILP	SA_β	Error	SA_β	ILP
IP reassembly	0.731415	0.731415	0	0.015	0.25
		0.731415	0	0.013	
		0.731415	0	0.015	
		0.731415	0	0.014	
		0.731415	0	0.015	
IP check	0.584844	0.584844	0	0.376	2.37
		0.584844	0	0.381	
		0.584844	0	0.339	
		0.584844	0	0.398	
		0.584844	0	0.353	
MD5	0.244155	0.244155	0	0.359	1.19
		0.244155	0	0.371	
		0.244155	0	0.279	
		0.244155	0	0.296	
		0.244155	0	0.328	
Huffman	0.205717	0.205717	0	0.516	3.97
		0.205717	0	0.632	
		0.205717	0	0.597	
		0.205717	0	0.712	
		0.205717	0	0.608	

Figure 5.2: Results Comparison between SA_β and ILP (RAM)

executions of SA_β. However, the error percentages in such cases are rather small. The bindings output by SA_β in these cases are near optimal. From the results of RAM mode shown in Figure 5.2, it is clear that P_{SA_β} is exactly the same with P_{ILP} in each execution of SA_β for all applications. The output bindings are optimal. Therefore, the functionality of SA_β is verified.

For the execution time, there is no comparability between SA_β and ILP because of the pre-condition that the optimal memory allocation is given to SA_β. This pro-condition makes the execution time of SA_β shorter than that of ILP for all applications.

5.2 Parameters adjustment of SA_β

One important feature of simulated annealing algorithm is the trade-off between efficiency and accuracy by adjusting the algorithm parameters. To verify this feature, the following testing scenarios are created. Application IPreassembly in ROM mode is used for all testing scenarios. In each testing scenario, only one parameter is adjusted while the others are kept the same with the setting used in Section 5.1. For each testing scenario, SA_β is performed five times. The average power error percentage and the average execution are measured. The corresponding results are represented in Figure 5.3 to Figure 5.6.

Figure 5.3 shows the results of using different value for $P_{threshold}$ in SA_β. From the figure it is can be seen that if $P_{threshold}$ is increased, the average power percentage increases as well. This means that SA_β terminates too early due to the high $P_{threshold}$. The possibility to accept a worser binding is also very high. Thus, SA_β may not provide a good enough

IPreassembly(ROM) $R_{cool} = 0{,}9;\ F_{iteration} = 1;\ P_0 = 0.9$		
$P_{threshold}$	Average power error [%]	Average execution time [s]
0.001	0.361	15.745
0.01	0.273	13.934
0.1	0.530	6.877
0.15	0.941	5.213
0.2	6.503	3.340
0.3	50.640	1.660

Figure 5.3: Results of $P_{threshold}$ Adjustment

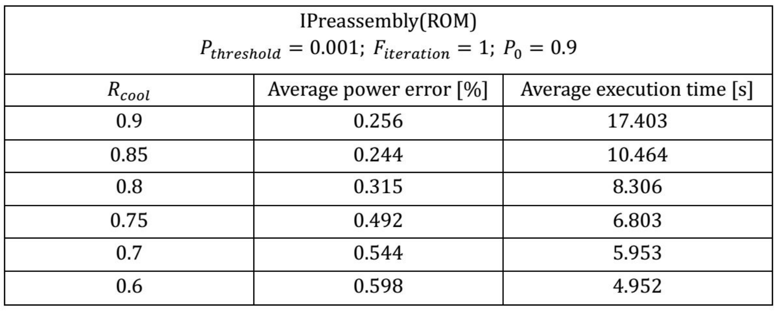
binding. In the $P_{threshold}$ equals to 0.2, the average power error percentage is lager than 5%. The binding provided by SA_β in this case is not near optimal. The result in the case of $P_{threshold}$ equals to 0.3 is much worser and it is unacceptable.

From Figure 5.3 it is also can be seen that the average execution time decreases with the increase of the average power error percentage. This means that by scarifying the accuracy, a better efficiency can be obtained in SA_β. It is noticed that in the first 4 cases shown in the figure, the average power error percentages are all lower than 1%. The bindings with such average power error percentages are considered as valuable. But the execution times of case $P_{threshold}$ equals to 0.1 and $P_{threshold}$ equals to 0.15 are much shorter than that of the other two. The performance of SA_β can be improved without losing too much accuracy if the $P_{threshold}$ is set in the range of 0.1 to 0.15 for this testing scenario.

IPreassembly(ROM) $P_{threshold} = 0.001;\ F_{iteration} = 1;\ R_{cool} = 0.9$		
P_0	Average power error [%]	Average execution time [s]
0.9	0.304	17.763
0.85	0.309	16.480
0.8	0.320	15.465
0.75	0.443	15.317
0.7	0.483	15.522
0.6	0.617	15.483

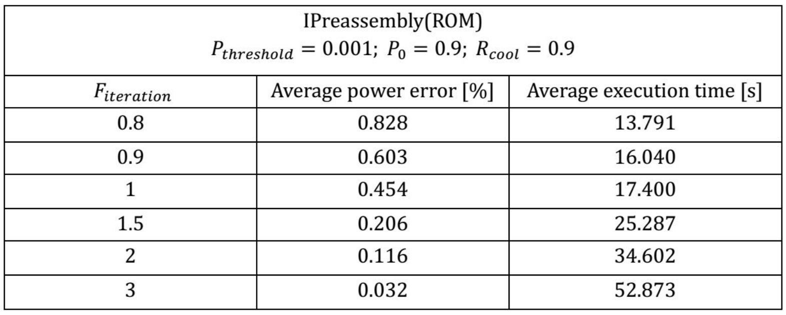
Figure 5.4: Results of P_0 Adjustment

Figure 5.4 shows the results of setting P_0 to different values. From the figure it is can be seen that with the adjustment of P_0, there is no huge change for neither the average power error percentage nor the average execution time. The efficiency and accuracy of SA_β are not affected by the value of P_0.



IPreassembly(ROM) $P_{threshold} = 0.001;\ F_{iteration} = 1;\ P_0 = 0.9$		
R_{cool}	Average power error [%]	Average execution time [s]
0.9	0.256	17.403
0.85	0.244	10.464
0.8	0.315	8.306
0.75	0.492	6.803
0.7	0.544	5.953
0.6	0.598	4.952

Figure 5.5: COOLING



IPreassembly(ROM) $P_{threshold} = 0.001;\ P_0 = 0.9;\ R_{cool} = 0.9$		
$F_{iteration}$	Average power error [%]	Average execution time [s]
0.8	0.828	13.791
0.9	0.603	16.040
1	0.454	17.400
1.5	0.206	25.287
2	0.116	34.602
3	0.032	52.873

Figure 5.6: FACTOR

6 Conclusion

A Appendix

Appendix goes here...

B List of Figures

C List of Algorithms

D List of Tables

E List of Abbreviations

MPSoC	Multiprocessor System-on-Chip

Bibliography

[Ben04] Walid Ben-Ameur. “Computing the Initial Temperature of Simulated Annealing”. In: *Computational Optimization and Applications* 29.3 (2004), pp. 369–385. ISSN: 1573-2894. DOI: 10.1023/B:COAP.0000044187.23143.bd. URL: http://dx.doi.org/10.1023/B:COAP.0000044187.23143.bd.

[BMP00] L. Benini, A. Macii, and M. Poncino. “A recursive algorithm for low-power memory partitioning”. In: *Low Power Electronics and Design, 2000. ISLPED ’00. Proceedings of the 2000 International Symposium on.* July 2000, pp. 78–83. DOI: 10.1145/344166.344518.

[Glo89] Fred Glover. “Tabu Search—Part I”. In: *ORSA Journal on Computing* 1.3 (1989), pp. 190–206. DOI: 10.1287/ijoc.1.3.190. eprint: http://dx.doi.org/10.1287/ijoc.1.3.190. URL: http://dx.doi.org/10.1287/ijoc.1.3.190.

[Glo90] Fred Glover. “Tabu Search—Part II”. In: *ORSA Journal on Computing* 2.1 (1990), pp. 4–32. DOI: 10.1287/ijoc.2.1.4. eprint: http://dx.doi.org/10.1287/ijoc.2.1.4. URL: http://dx.doi.org/10.1287/ijoc.2.1.4.

[His05] Jason D. Hiser. “Effective Algorithms for Partitioned Memory Hierarchies in Embedded Systems”. AAI3169653. PhD thesis. Charlottesville, VA, USA, 2005. ISBN: 0-542-05748-4.

[Joh+89] David S. Johnson et al. “Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning”. In: *Operations Research* 37.6 (1989), pp. 865–892. DOI: 10.1287/opre.37.6.865. eprint: http://dx.doi.org/10.1287/opre.37.6.865. URL: http://dx.doi.org/10.1287/opre.37.6.865.

[Joh+91] David S. Johnson et al. “Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning”. In: *Oper. Res.* 39.3 (May 1991), pp. 378–406. ISSN: 0030-364X. DOI: 10.1287/opre.39.3.378. URL: http://dx.doi.org/10.1287/opre.39.3.378.

[Mai+07] Songping Mai1 et al. “An application-specific memory partitioning method for low power”. In: *2007 7th International Conference on ASIC.* Oct. 2007, pp. 221–224. DOI: 10.1109/ICASIC.2007.4415607.

[MBP02] Alberto Macii, Luca Benini, and Massimo Poncino. *Memory Design Techniques for Low Energy Embedded Systems.* Springer Science & Business Media, 2002.

[S K83] M. P. Vecchi S. Kirkpatrick C. D. Gelatt. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680. ISSN: 00368075, 10959203. URL: http://www.jstor.org/stable/1690046.

[SER16] Manuel Strobel, Marcus Eggenberger, and Martin Radetzki. “Low power memory allocation and mapping for area-constrained systems-on-chips”. In: *EURASIP Journal on Embedded Systems* 2017.1 (2016), p. 2. ISSN: 1687-3963. DOI: `"10.1186/s13639-016-0039-5"`. URL: `http://dx.doi.org/10.1186/s13639-016-0039-5`.

[Whi84] Steve R White. “Concepts of scale in simulated annealing”. In: *The Physics of VLSI*. Vol. 122. 1. AIP Publishing. 1984, pp. 261–270.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.

Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.

Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.

Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Stuttgart, 13.12.2016

Declaration

I hereby declare that the work presented in this thesis is entirely my own.

I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.

Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.

The electronic copy is consistent with all submitted copies.

Signature:

Stuttgart, 13.12.2016