Best Practices for Simulated Annealing in Multiprocessor Task Distribution Problems

Heikki Orsila, Erno Salminen and Timo D. Hämäläinen
Department of Computer Systems
Tampere University of Technology
P.O. Box 553, 33101 Tampere,
Finland

1. Introduction

Simulated Annealing (SA) is a widely used meta-algorithm for complex optimization problems. This chapter presents methods to distribute executable tasks onto a set of processors. This process is called *task mapping*. The most common goal is to decrease execution time via parallel computation. However, the presented mapping methods are not limited to optimizing application execution time because the cost function is arbitrary. The cost function is also called an objective function in many works. A smaller cost function value means a better solution. It may consider multiple metrics, such as execution time, communication time, memory, energy consumption and silicon area constraints. Especially in embedded systems, these other metrics are often as important as execution time.
A multiprocessor system requires exploration to find an optimized architecture as well as the proper task distribution for the application. Resulting very large design space must be pruned systematically with fast algorithms, since the exploration of the whole design space is not feasible. Iterative algorithms evaluate a number of application mappings for each architecture, and the best architecture and mapping is selected in the process.
The optimization process is shown in Figure 1(a). The application, the HW platform and an initial solution are fed to a mapping component. The mapping component generates a new solution that is passed to a simulation component. The simulation component determines relevant metrics of the solution. The metrics are passed to a cost function which will evaluate the badness (*cost*) of the solution. The cost value is passed back to the mapping component. The mapping component will finally terminate the optimization process and output a final solution.
The system that is optimized is shown in Figure 1(b). The system consists of the application and the HW platform. The application consists of tasks which are mapped to processing elements (PEs). The PEs are interconnected with a communication network.
The chapter has two focuses:

- optimize the cost function and
- minimize the time needed for simulated annealing.

First, the task distribution problem is an NP problem which implies that a heuristic algorithm is needed. The focus is on reaching as good as possible mapping. Unfortunately the true optimum value is unknown for most applications, and therefore the relative

Source: Simulated Annealing, Book edited by: Cher Ming Tan, ISBN 978-953-7619-07-7, pp. 420, February 2008, I-Tech Education and Publishing, Vienna, Austria

goodness of the solution to the true optimum is unknown. Experiments rely on convergence rates and extensive simulations to reduce this uncertainty. This chapter focuses on single-objective rather than multi-objective optimization.

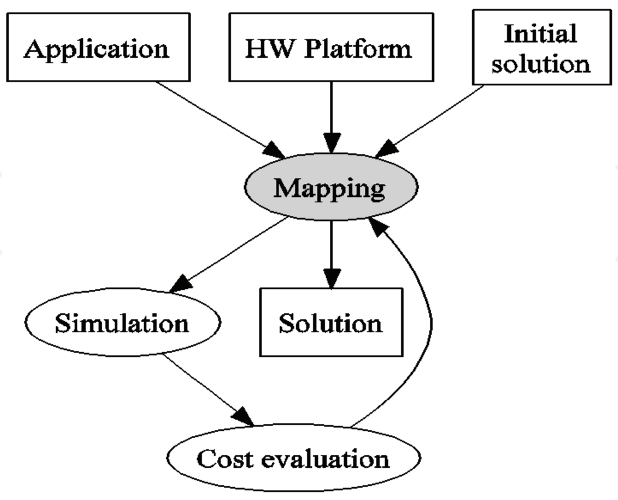

Figure 1(a). Optimization process. Boxes indicate data. Ellipses indicate operations. This chapter focuses on the mapping part.

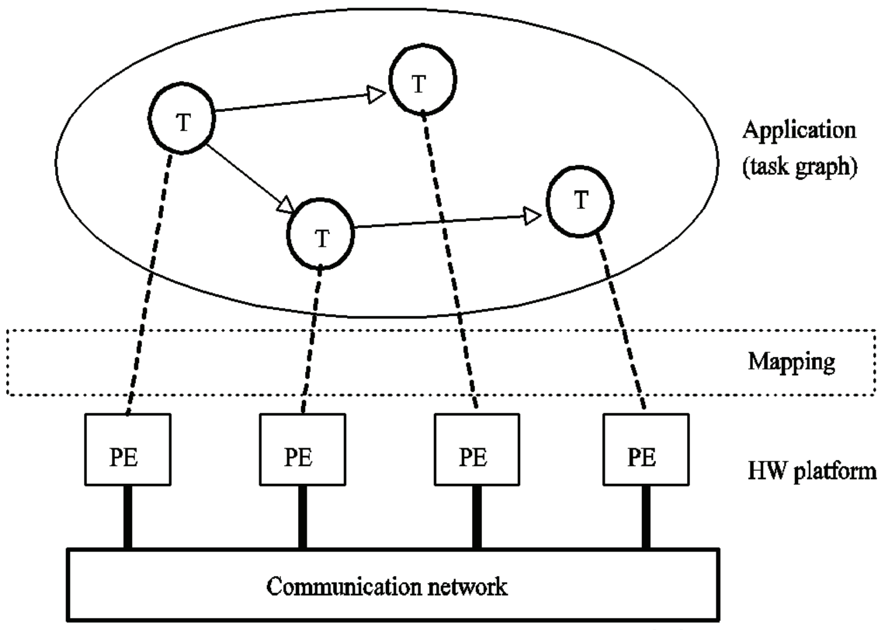

Figure 1(b). The system that is optimized. The system consists of the application and the HW platform. PE is processing element.

Second, the focus is minimizing the optimization time. A valid solution must be found in a reasonable time which depends on the application and the target multiprocessor platform.

This chapter is structured as follows. We first introduce the problem of mapping a set of tasks onto a multiprocessor system. Then, we present a generic SA algorithm and give detailed analysis how the major functions may be implemented. That is followed by an overview of reported case studies, including our own. Last we discuss the findings and present the most important open research problems.

2. Task mapping problem

The application in Figure 1(b) is divided into tasks. Tasks are defined as smallest components in the application that can be relocated to any or some PEs in the HW platform. A mapping algorithm will find a location for each task on some PE. The application model is irrelevant for the general mapping problem as long as the application model has mappable tasks. Mapping can be done on run-time or design-time. There are several types of application models that are used in literature: directed acyclic task graphs (Kwok & Ahmad, 1999), Kahn Process Networks (Wikipedia, 2008b) and others.
The mapping affects several properties of the system. Affected hardware properties are processor utilization, communication network utilization and power. Affected software and/or hardware properties are execution time, memory usage, and application and hardware context switches.

2.1 Application model

Tasks can be dependent on each other. Task *A* depends on task *B* if task *A* needs data or control from task *B*. Otherwise tasks are independent. There are application models with dependent and independent tasks. Models with independent tasks are easier to map because there is zero communication between tasks. This enables the problem to be solved in separate sub-problems. However, independent tasks may affect each other if they compete for shared resources, such as a PE or a communication network. Scheduling properties of the application model may complicate evaluating a mapping algorithm.

2.2 Hardware platform model

The HW platform in Figure 1(b) can be heterogeneous which means that it executes different tasks with different characteristics. These characteristics include speed and power, for example. This does not complicate the mapping problem, but affects the simulation part in Figure 1(a). The mapping problem is the same regardless of the simulation accuracy, but the mapping solution is affected. This enables both fast and slow simulation models to be used with varying accuracy. Inaccurate models are usually based on estimation techniques. Accurate models are based on hardware simulation or native execution of the system that is being optimized. Accurate models are usually much slower than inaccurate models and they may not be available at the early phase of the system design.
Depending on the application model, all PEs can not necessarily execute all tasks. Restricting mappability of tasks makes the optimization problem easier and enables shortcut heuristics to be used in optimization. The previous definition for tasks excludes application components that can not be relocated, and therefore each task has at least 2 PEs where it can be executed.

2.3 Limiting the scope of problems

We assume that communicating between two processors is much more expensive than communicating within a single processor. To generalize this idea, it is practically happening inside single processor computer systems because registers can be *100* times as fast as physical memory, and cache memory is *10* times as fast as physical memory. Multiprocessor systems could spend thousands of cycles to pass a message from one processor to other.

This trend is constantly changing as multicore and non-asymmetric computer architectures are becoming more common.
We also assume that distributed applications are not embarrassingly parallel (Wikipedia, 2008a).
Without previous two assumptions the optimization algorithms can be trivially replaced with on-demand best-effort distributed job queues.
This paper only considers the single-objective optimization case. Single-objective optimization finds the minimum for a given objective function. Multi-objective optimization tries to minimize several functions, and the result is a set of trade-offs, or so called *Pareto-optimal* solutions. Each trade-off solution minimizes some of the objective functions, but not all. Having a systematic method for selecting a single solution from the trade-off set reduces the problem into a single-objective optimization task.

2.4 Random mapping algorithm

Random mapping algorithm is a simple *Monte Carlo* algorithm that randomizes processor assignment of each task at every iteration. The Monte Carlo process converges very slowly as it does not have negative feedback for moves into worse mappings. Random mapping algorithm is important because it sets the reference for minimum efficiency of any mapping algorithm. Any mapping algorithm should be able to do better than random mapping. Simulated Annealing algorithm produces a "Monte Carlo -like" effect at very high temperatures as almost all worsening moves are accepted.

3. Simulated annealing

Simulated Annealing is a probabilistic non-greedy algorithm (Kirkpatrick et al., 1983) that explores the search space of a problem by annealing from a high to a low temperature. Probabilistic behavior means that SA can find solutions of different goodness between independent runs. Non-greedy means that SA may accept a move into a worse state, and this allows escaping local minima. The algorithm always accepts a move into a better state. Move to a worse state is accepted with a changing probability. This probability decreases along with the temperature, and thus the algorithm starts as a non-greedy algorithm and gradually becomes more and more greedy.
This chapter focuses only on using SA for mapping. The challenge is to find efficient optimization parameters for SA. (Braun et al., 2001) is a comparison of different mapping algorithms, such as *Tabu Search*, *Genetic Algorithms*, *Load Balancing* algorithms and others.
Figure 2 shows an example of SA optimization process. Optimization begins from a high temperature where the accepted cost changes chaotically. As the temperature decreases the accepted cost changes less chaotically and the algorithm becomes greedier.
Figure 3 shows the general form of Simulated Annealing algorithm pseudo-code. Table 1 shows symbols, functions and various parameters for the pseudo-code. The algorithm starts with an initial solution S_0 (state). SA iterates through solutions until a termination condition is reached. At each temperature level, SA moves one or several tasks to different PEs and evaluates the cost of the new mapping solution. Then SA either accepts or rejects the new solution. If the new solution is accepted, it is used as a basis for the next iteration. Otherwise, the new solution is thrown away.

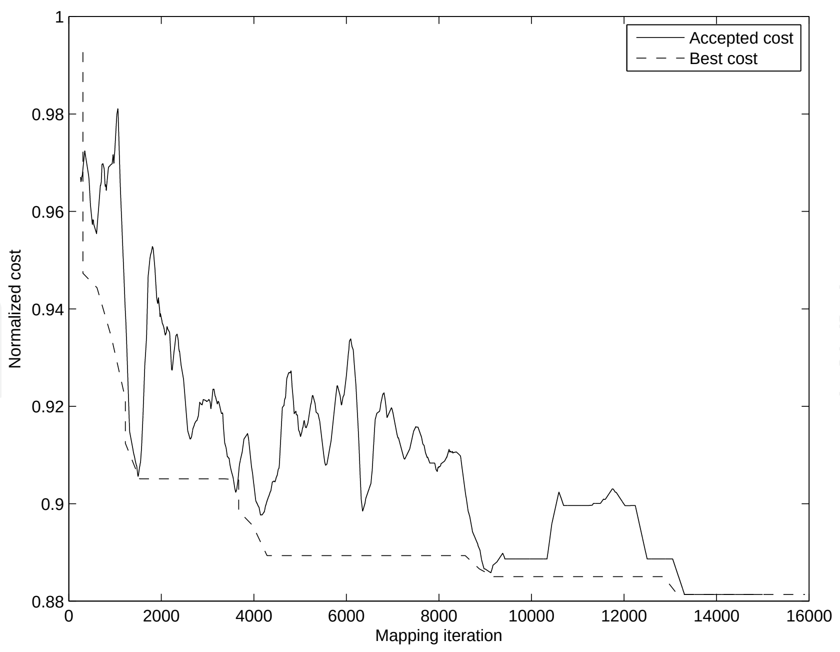

Figure 2. Cost per iteration plotted for Simulated Annealing when mapping a *100* task application to a *4* processor system. The cost is normalized so that initial cost $C_0 = 1.0$. The plot is average filtered with a *256* sample window to hide the chaotic nature of the random process. This is also the reason why accepted cost does not always seem to touch the best cost line.

```
SIMULATED_ANNEALING($S_0$)
 1  $S \leftarrow S_0$
 2  $C \leftarrow$ COST($S_0$)
 3  $S_{best} \leftarrow S$
 4  $C_{best} \leftarrow C$
 5  $R \leftarrow 0$
 6  for $i \leftarrow 0$ to $\infty$
 7      do $T \leftarrow$ TEMP($i$)
 8         $S_{new} \leftarrow$ MOVE($S, T$)
 9         $C_{new} \leftarrow$ COST($S_{new}$)
10         $\Delta C \leftarrow C_{new} - C$
11         if $\Delta C < 0$ or ACCEPT($\Delta C, T$)
12            then if $C_{new} < C_{best}$
13                    then $S_{best} \leftarrow S_{new}$
14                         $C_{best} \leftarrow C_{new}$
15                 $S \leftarrow S_{new}$
16                 $C \leftarrow C_{new}$
17                 $R \leftarrow 0$
18            else $R \leftarrow R + 1$
19         if TERMINATE($i, R$) = True
20            then break
21  return $S_{best}$
```

Figure 3. Pseudo-code of the Simulated Annealing algorithm. See Table 1 for explanation of symbols.

Symbol	Value range	Definition	A	B	C
$Accept(\Delta C, T)$	$\{\textbf{False}, \textbf{True}\}$	Return accept (**True**) or reject (**False**) for a worsening move		B B	
$C = Cost()$	$C > 0$	Accepted cost (to be minimized)		B	
C_0	$C_0 > 0$	Initial cost			C
C_{new}	$C_{new} > 0$	Cost of the next state			C
$\Delta C = C_{new} - C$	$\mathbb{R}$	Change of cost due to move			C
i	$i > 0$	Mapping iteration			C
L	$L > 0$	# Iterations per temperature level		B	
M	$M > 1$	Number of processors	A		
N	$N > 1$	Number of tasks	A		
q	$0 < q < 1$	Geometric temperature scaling factor		B	
R	$R \geq 0$	Number of consecutive rejected moves		B	
S	mapping space	Accepted state			C
S_0	mapping space	Initial state		B	
S_{new}	mapping space	Next state			C
$Move(S, T)$	mapping space	Returns the next state		B	
$T = Temp(i)$	$T > 0$	Return temperature T at iteration i		B	
T_0	$T_0 > 0$	Initial temperature		B	
T_f	$0 < T_f < T_0$	Final temperature		B	
T_N	$T_N > 0$	Number of temperature levels		B	
$Terminate(i, R)$	$\{\textbf{False}, \textbf{True}\}$	Return terminate (**True**) or continue (**False**)		B	
$x = random()$	$0 \leq x < 1$	Return a random value			C
α	$\alpha > 0$	The number of neighbors for each state: $\alpha = M(N-1)$	A		

Table 1. Simulated Annealing parameters and symbols. Column A indicates parameters related to the size of the mapping/optimization problem. Column B indicates parameters of the SA algorithm. Column C indicates an internal variable of the SA.

The general algorithm needs a number of functions to be complete. Most common methods are presented in following sections. Implementation effort for most methods is low, and trying different combinations requires little effort. Therefore many alternatives should be tried. Most of the effort goes to implementing the *Cost()* function and finding proper optimization parameters. The cost function is the simulation and cost evalution part in Figure 1(a). In some cases the Move heuristics can be difficult to implement.

3.1 Cost function: Cost(S)

Cost(S) evaluates the cost for any given state *S* of the optimization space. Here, each point in the optimization space defines one mapping for the application. *Cost()* can be a function of any variables. Without loss of generality, this chapter is only concerned about minimizing execution time of the application. Other factors such as power and real-time properties can be included. For example, $Cost(S) = t^{w_1} A^{w_2} P^{w_3}$, where t is the execution time of the application, A is the silicon area and P is the power, and w_1, w_2 and w_3 are user-defined coefficients.

3.2 Annealing schedule: Temp(i) function

Temp(i) determines the temperature as a function of the iteration number *i*. Initial temperature $T_0 = Temp(0)$. The final temperature T_f is determined implicitly by *Temp()* and *Terminate()* functions. *Temp()* function may also contain internal state, and have access to other annealing metrics, such as cost. In those cases *Temp()* is not a pure function. For example, remembering cost history can be used for intelligent annealing schedules.
In geometric temperature schedules the temperature is multiplied by a factor $0 < q < 1$ between each temperature level. It is the most common approach. T_N is the number of temperature levels. Define *L* to be the number of iterations on each temperature level.
There are *3* common schedules that are defined in following paragraphs.
Geometric Temperature Schedule

$$Temp(i) = T_0 q^{\left\lfloor \frac{i}{L} \right\rfloor} \tag{1}$$

$\left\lfloor \frac{i}{L} \right\rfloor$ means rounding down the fraction. The number of mapping iterations is LT_N.
Fractional Temperature Schedule

$$Temp(i) = \frac{T_0}{i+1} \tag{2}$$

The number of mapping iterations is T_N. It is inadvisable to use a fractional schedule because it distributes the number of iterations mostly to lower temperatures. Doubling the total number of iterations only halves the final temperature. Therefore, covering a wide relative temperature range $\frac{T_0}{T_f} >> 1$ is expensive. The geometric schedule avoids this problem. For this reason the geometric temperature schedule is the most common choice.
Koch Temperature Schedule

$$Temp(i) = \begin{cases} \dfrac{Temp(i-1)}{1 + \delta \dfrac{Temp(i-1)}{\sigma_{i-L,i}}} & \text{if } \operatorname{mod}(i, L) = 0 \\ Temp(i-1) & \text{if } \operatorname{mod}(i, L) \neq 0 \\ T_0 & \text{if } i = 0 \end{cases} \tag{3}$$

where

$$\sigma_{i-L,i} = stddev\{Cost(S_k) \mid i - L \le k < i\} \tag{4}$$

Koch temperature schedule (Koch, 1995; Ravindran, 2007) decreases temperature with respect to cost standard deviation on each temperature level. Deviation is calculated from the *L* latest iterations. Higher standard deviation, i.e. more chaotic the annealing, leads to lower temperature decrease between each level. The number of mapping iterations depends on the problem.

3.3 Acceptance function: Accept (ΔC,T)

$Accept(\Delta C, T)$ returns **True** if a worsening move should be accepted, otherwise **False**. An improving move ($\Delta C < 0$) is always accepted by the SA algorithm, but this is not a part of Accept() behavior (although there are some implementations that explicitly do it).
ΔC has an arbitrary range and unit that depends on system parameters and the selected cost function. Since $\frac{\Delta C}{T}$ is a relevant measure in acceptance functions, the temperature range needs to be adjusted to the ΔC range, or vice versa. Following paragraphs define *4* different acceptance functions.

3.3.1 Inverse exponential form

$$Accept(\Delta C, T) = \textbf{True} \Leftrightarrow random() < \frac{1}{1 + \exp(\frac{\Delta C}{T})} \tag{5}$$

It is important to notice that when $\Delta C = 0$, the transition happens at *50%* probability. This makes SA rather likely to shift between equally good solutions and thus find new points in space where a move to a better state is possible. Accepting a worsening move always has a probability less than *50%*. Despite this, SA is rather liberal in doing random walks even at low temperatures. Small increases in cost are allowed even at low temperatures, but significant increases in cost are only accepted at high temperatures.

Note that some implementations write the right part of (5) as $random() > \frac{1}{1 + \exp(\frac{-\Delta C}{T})}$, which is probabilistically equivalent.

3.3.2 Normalized inverse exponential form

$$Accept(\Delta C, T) = \textbf{True} \Leftrightarrow random() < \frac{1}{1 + \exp(\frac{\Delta C}{C_0 T})} \tag{6}$$

This case has all the properties of the inverse exponential form, but the cost value difference is normalized. The idea is that selecting the temperature range $[T_f, T_0]$ is easier when it is independent of the cost function and the temperature always lies inside the same range $0 < T \leq 1$. Specifically, changing the hardware platform should not make temperature range selection harder. Normalization keeps acceptance probabilities in a relevant range even if the cost function changes. Figure 4 shows specific probability curves for $\Delta C_r = \frac{\Delta C}{C_0}$ that is used inside the *exp()* function.

3.3.3 Exponential form

$$Accept(\Delta C, T) = \textbf{True} \Leftrightarrow random() < \exp(\frac{-\Delta C}{T}) \tag{7}$$

Exponential form is similar to the inverse exponential form, but $\Delta C = 0$ transition happens always whereas the inverse exponential form accepts the same move with *50%* probability. See the reasoning in inverse exponential case.

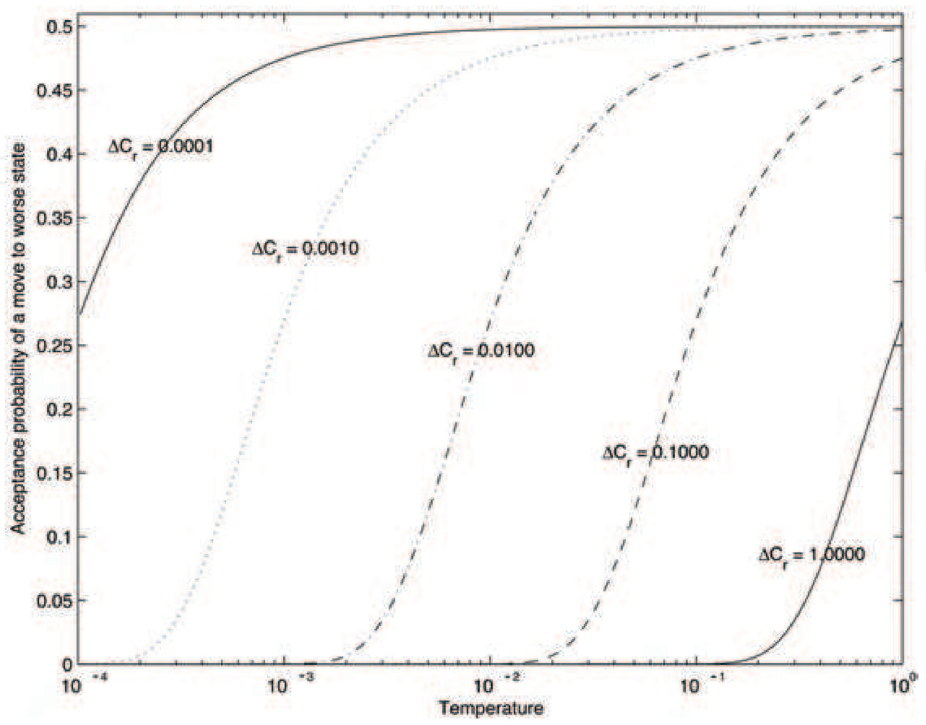

Figure 4. Acceptance probability curves for the normalized inverse exponential function (6) with $\mathbf{q = 0.95}$. The curve represents constant values of $\mathbf{\Delta C_r = \frac{\Delta C}{C_0}}$. Probability of moving to a worse state decreases when the temperature decreases. Moves to slightly worse state have higher probability than those with large degradation.

3.3.4 Normalized exponential form

$$Accept(\Delta C, T) = \mathbf{True} \Leftrightarrow random() < \exp(\frac{-\Delta C}{C_0 T}) \qquad (8)$$

This case has all the properties of the exponential form, but in addition it is implied that temperature lies in range $0 < T \leq 1$. This is reasoned in the normalized inverse exponential case.

3.4 On effective temperature range

Annealing starts with a high acceptance rate p_0 for bad moves and it decreases to a very low acceptance rate p_f. It is important to control the acceptance probability. If inverse exponential function (5) is solved with respect to *T* for a given probability *p*, we get:

$$T = \frac{\Delta C}{\ln(\frac{1}{p} - 1)} \qquad (9)$$

Assuming minimum expected cost change $\Delta C_{\min}$ and maximum expected cost change $\Delta C_{\max}$, we get the proper temperature range

$$T_f = \frac{\Delta C_{\min}}{\ln(\frac{1}{p_f} - 1)} < T < \frac{\Delta C_{\max}}{\ln(\frac{1}{p_0} - 1)} = T_0 \tag{10}$$

Initial acceptance probability p_0 should be set close to *0.5*, i.e. the maximum acceptance rate for inverse exponential function, but not too close to save optimization iterations. For example, $p_0 = 0.45$ is sufficiently close to *0.5*, but saves *58* temperature levels of iterations compared to $p_0 = 0.49$, assuming $q = 0.95$. When $\Delta C = 0$ the acceptance probability is always *50%*.
Final acceptance probability p_f can be set large enough so that a worsening move happens *n* times in the final temperature level, where n is a parameter set by the designer. If there are *L* iterations per temperature level, we set $p_f = n / L$. If we set $n = 0.1$, the final temperature level is almost entirely greedy, and a worsening move happens with *10%* probability on the temperature level for a given $\Delta C_{\min}$. The temperature range becomes

$$T_f = \frac{\Delta C_{\min}}{\ln(\frac{L}{n} - 1)} < T < \frac{\Delta C_{\max}}{\ln(\frac{1}{p_0} - 1)} = T_0 \tag{11}$$

The derivation of (10) and (11) for normalized inverse exponential, exponential and normalized exponential functions is similar.

3.5 Methods to determine the initial temperature

The initial temperature T_0 was not defined in annealing schedule functions in Section 3.2. As was explained in Section 3.3, the initial temperature is highly coupled with the acceptance function. Following paragraphs present common methods for computing the initial temperature. Note that final temperature is usually determined implicitly by the *Terminate()* function.

3.5.1 Heating

The initial temperature is grown large enough so that the algorithm accepts worsening moves with some given probability p_0. This requires simulating a sufficient number of moves in the optimization space. Either moves are simulated in the neighborhood of a single point, or moves are simulated from several, possibly random, points. The average increase in cost ΔC_{avg} is computed for worsening moves. Given an acceptance function, T_0 is computed such that $Accept(\Delta C_{avg}, T_0) = p_0$. The solution is trivial for all presented acceptance functions. An example of heating is given in Section 4.2.

3.5.2 Application and hardware platform analysis

Application and hardware platform analysis can be used to determine the initial temperature. Rapid methods in this category do not use simulation to initialize parameters, while slow but more accurate methods use simulation. An example, see (10), (11) and Section 4.3.

3.5.3 Manual tuning

Parameters can be set by manually testing different parameters. This option is discouraged for an automated optimization system where the problem varies significantly.

3.5.4 Cost change normalization

In this method the temperature scale is made independent of the cost function values. This is either accomplished by (6) or setting $T_0 = C_0$ for (5). By using (6) it is easier to use other initial temperature estimation methods.

3.6 Move function and heuristics: Move(S, T)

Move(S, T) function returns a new state based on the application specific heuristics and the current state S and temperature T . Move heuristics vary significantly. The simple ones are purely random. The complex ones analyze the structure of the application and the hardware, and inspect system load.

It should be noted that given a current state value, randomizing a new state value should exclude the current value, i.e. current PE of the moved task in this case, for randomization process. For example, in two-processor system, there is a *50%* probability of selecting the same CPU again, which means that half of the iterations are wasted. Many papers do not specify this aspect for random heuristics.

Common choices and ideas for move heuristics from literature are presented in following sections.

3.6.1 Single: move task to another processor

Choose a random task and move it to a random processor.

3.6.2 Multiple: move several tasks to other processors

Instead of choosing only a single task to move to another processor, several tasks can be moved at once. The moved tasks are either mapped to the same processor, or different processors. If these tasks are chosen at random and each of their destinations are chosen at random, this approach is less likely to find an improving move than just moving a single task. This is a consequence of combinatorics as improving moves are a minority group in all possible moves.

If a good heuristics is applied for moving multiple tasks, it is possible to climb up from a steep local minimum. A heuristics that only moves a single task is less likely to climb up from a steep local minimum.

3.6.3 Swap: swap processes between processors

Choose two different random processors, choose a random process on both processors, and swap the processes between processors.

3.7 Heuristic move functions

A heuristic move uses more information than just knowing the mapping space structure. Some application or hardware specific knowledge is used to move or swap tasks more efficiently.

3.7.1 ECP: Enhanced critical path

Enhanced Critical Path method (Wild et al., 2003) is a heuristic move for directed acyclic task graphs. ECP favors swapping and moving processes that are on the *critical path* of the graph, or near the critical path. Critical path is the path with the largest sum of computation and communication costs in the graph.

3.7.2 Variable grain move

A variable grain move is a single task move that starts by favoring large execution time tasks statistically. Thus, tasks with large execution time are moved more likely than tasks with small execution time. The probability distribution is then gradually flattened towards equal probability for each task. At low temperatures each task is moved with the same probability.

3.7.3 Topological move

Assume tasks *A* and *B*, where *A* sends a message to *B* with a high probability after *A* has been activated. If *B* is the only task that gets a message from *A* with a high probability then it can be benefitial to favor moving them to the same processor.
This heuristics could be implemented into Single task move by favoring processors of adjacent tasks. The probability distribution for processor selection should be carefully balanced to prevent mapping all tasks to the same processor, thus preventing speedup of a multiprocessor system. If a task sends messages to more than one task with a high probability, this heuristics is at least dubious and needs experimental verification.

3.7.4 Load balancing move

This heuristics makes heavily loaded processors less likely to get new tasks, and make slightly loaded processes more likely to get new tasks. Each processor's load can be determined by a test vector simulation, by counting the number of tasks on each processor, or by using more sophisticated load calculations. Each task can be attributed a constant load based on test vector simulations, and then each processor's load becomes the sum of loads of its tasks.

3.7.5 Component move

A task graph may consist from application or system level components each having multiple tasks. Separate components are defined by the designer. Instead of mapping single tasks, all tasks related to a single component could be mapped. This could be a coarse-grain starting point for finer-grain mapping.

3.8 Other move heuristics

3.8.1 Hybrid approach

A hybrid algorithm might use all of the above move functions. For example, combine weighted task selection with weighted target PE selection (Sec 3.7.2 + 3.7.3). The move function can be selected by random on each iteration, or different move function can be used in different optimization phases.

3.8.2 Compositional approach

SA can be combined with other algorithms. The move function may use another optimization algorithm to make more intelligent moves. For example, the single move heuristics might be adapted to give more weight to the best target processor determined by actually simulating each target.

3.8.3 Optimal subset mapping move

The move function can optimize a subset of the task graph. Each move will by itself determine a locally optimal mapping for some small subset of tasks. The number of mapping combinations for a subset of N_{sub} tasks and *M* processors is $M^{N_{sub}}$ for the brute-force approach. The number of brute-combinations for a single subset should only be a tiny fraction of total number of mappings that are evaluated, that is, a large number of subsets should be optimized. A brute-force based approach may yield rapid convergence but the final result is somewhat worse than with traditional SA (Orsila et al., 2007). It is suitable for initial coarse-grain optimization.

3.8.4 Move processors from router to router

In a Network-on-Chip (NoC) system, processors can be moved from router to router to optimize communication between system components.

3.8.5 Task scheduling move

Scheduling of tasks can be done simultaneously with mapping them. Scheduling means determining the priorities of tasks on each processor separately. Priorities for tasks is determined by a permutation of all tasks. Task *A* has higher priority than task *B* if it is located before task *B* in the permutation. A permutation can be altered by swapping two random tasks in the Move function. The order of tasks is only relevant for tasks on the same processor. As an optimization for the move heuristics, most permutations need not be considered.

3.9 Termination function: Terminate(*i*, R)

Terminate(i, R) returns **True** when the optimization loop should be terminated. *R* is the number of consecutive rejected moves, $i_{\max}$ is a user-defined maximum number of iterations, and $R_{\max}$ is a user-defined maximum number of consecutive rejects. *Terminate()* function often uses the T emp() function for determining the current temperature *T*.
Following paragraphs present examples and analysis of commonly used termination functions from literature:

3.9.1 Maximum number of iterations

Annealing is stopped after i_{max} iterations:

$$Terminate(i,R) = \textbf{True} \Leftrightarrow i \geq i_{\max} \tag{12}$$

This approach is discouraged because annealing success is dependent on actual temperatures, rather than iterations. Final temperature and annealing schedule parameters can be selected to restrict the maximum number of iterations.

3.9.2 Temperature threshold

Annealing is stopped at a specific temperature T_f :

$$Terminate(i,R) = \textbf{True} \Leftrightarrow Temp(i) < T_f \tag{13}$$

This approach is discouraged in favor of coupled temperature and rejection threshold because there can be easy greedy moves left.

3.9.3 Cost threshold

Annealing is stopped when a target cost is achieved:

$$Terminate(i,R) = \textbf{True} \Leftrightarrow Cost(S) < Cost_{target} \tag{14}$$

For example, if the cost function measures real-time latency, annealing is stopped when a solution that satisfies real-time requirements is found. This heuristics should not be used alone because if the target cost is not achieved, the algorithm loops forever.

3.9.4 Rejection threshold

Annealing is stopped when $R \geq R_{\max}$:

$$Terminate(i,R) = \textbf{True} \Leftrightarrow R \geq R_{\max} \tag{15}$$

This approach is discouraged because there is a risk of premature termination.

3.9.5 Uncoupled temperature and rejection threshold

Annealing is stopped at a low enough temperature or if no improvement has occured for a while:

$$Terminate(i,R) = \textbf{True} \Leftrightarrow Temp(i) < T_f \vee R \geq R_{\max} \tag{16}$$

This approach is discouraged because there is a risk of premature termination.

3.9.6 Coupled temperature and rejection threshold

Annealing is stopped at a low enough temperature only when no improvement has occured for a while:

$$Terminate(i,R) = \textbf{True} \Leftrightarrow Temp(i) < T_f \wedge R \geq R_{\max} \tag{17}$$

This approach has the benefit of going through the whole temperature scale, and continue optimization after that if there are acceptable moves. This will probably drive the solution into a local minimum.

3.9.7 Hybrid condition

Any logical combination of conditions 3.9.1 - 3.9.6 is a valid termination condition.

4. Case studies

This section summarizes *5* relevant works on the use of SA for task mapping. Task mapping problems are not identical but comparable in terms of SA parameterization. Selected SA parameterizations are presented to give insight into possible solutions. Table 2 shows move heuristics and acceptance functions, and Table 3 shows annealing schedules for the same cases. These cases are presented in detail in following sections.

Implementation	Move Function	Acceptance Function
Braun (Sec 4.1)	Single	Normalized Inverse Exponential
Coroyer (Sec 4.2)	Single, Task Scheduling	Exponential
Orsila (Sec 4.3)	Single	Normalized Inverse Exponential
Ravindran (Sec 4.4)	Single	Exponential
Wild (Sec 4.5)	Single, ECP	N/A

Table 2. Simulated Annealing move heuristics and acceptance functions

Implementation	Annealing Schedule	T_0	End condition	L
Braun (Sec 4.1)	Geometric, $q = 0.90$	C_0	$T_f = 10^{-200}$	1
Coroyer (Sec 4.2)	Geometric, Fractional	Heuristic	Heuristic	α
Orsila (Sec 4.3)	Geometric, $q = 0.95$	Heuristic	Heuristic	α
Ravindran (Sec 4.4)	Koch	$T_0 = 1$	N/A	N/A
Wild (Sec 4.5)	Geometric, $q = N/A$	N/A	Heuristic	N/A

Table 3. Simulated Annealing schedules. See Table 1 for symbols.

Single move (Sec 3.6.1) and the Geometric annealing scheduling (1) are the most common choices. They should be tested in every new experiment. All the cases use a single move so it is not covered in each case. Other choices are explicitly documented.

4.1 Braun case

(Braun et al., 2001) uses an inverse exponential form (5) as an acceptance function. However, the method uses it to actually implement a normalized inverse exponential form (6) by setting $T_0 = C_0$.

A geometric temperature schedule (1) with $q = 0.90$ and $L = 1$ is used.

The termination condition is an uncoupled temperature and rejection threshold (16). Optimization is terminated when $T_f = 10^{-200}$ or when $R_{\max} = 200$ consecutive solutions are identical. The choice for L and T_f values are not explained. If the HW platform or the number of tasks were changed, then trivially the number of iterations should be adjusted as well.

The initial mapping used was a random mapping of tasks.

The paper compares SA to ten other heuristics for independent task mapping problem. SA got position *8/11*, where *1/11* is the best position received by a genetic algorithm. We believe SA was used improperly in this comparison. Based on (11), we think T_f was set too low, and L should be much larger than *1*.

4.2 Coroyer case

(Coroyer & Liu, 1991) do both single and task scheduling (Sec 3.8.5) moves.

The acceptance function is exponential (7) accompanied with a heating process that puts acceptance probabilities to a relevant range. Initial temperature is set high enough so that $p_0 = 0.95$ of new mappings are accepted. If ΔC_{avg} is the average increase in cost for generating new solutions, the initial temperature is set to $T_0 = \frac{-\Delta C_{avg}}{\ln p_0}$. This approach depends on the exponential acceptance function, but it can easily be adopted for other acceptance functions. The average increase is determined by simulating a sufficient number of moves. See Section 3.5.1.

Both fractional (2) and geometric (1) temperature schedules are used with various parameters. The number of mapping iterations per temperature level is $L = \alpha = N(M-1)$.
The termination condition is an uncoupled temperature and rejection threshold (16). Optimization is terminated when $T_f \leq 10^{-2}$ or when $R_{\max} = 5\alpha$ consecutive solutions are identical. Also, a given temperature threshold (13) is used.
The initial mapping used was a random mapping of tasks.
They show that SA gives better results than priority-based heuristics for task mapping and scheduling, but SA is also much slower.
Systematic methods are not used to tune parameters.

4.3 Orsila case

This case presents methods to derive SA parameters systematically from the problem parameters (Orsila et al., 2006).

The annealing schedule is geometric with $q = 0.95$. The number of iterations per temperature level is $L = \alpha = N(M-1)$.

The initial and final temperature range $[T_f, T_0] \subset (0,1]$ is defined with

$$T_0 = \frac{k t_{\max}}{t_{\min sum}} \tag{18}$$

$$T_f = \frac{t_{\min}}{k t_{\max sum}} \tag{19}$$

where $t_{\max}$ and $t_{\min}$ are the maximum and minimum execution time for any task (when it is activated) on any processor, $t_{\min sum}$ is the sum of execution times for all tasks on the fastest processor in the system, $t_{\max sum}$ is the sum of execution times for all tasks on the slowest processor in the system, and $k \geq 1$ is a coefficient.
The temperature range is tied to a slightly modified version of (6). The factor 0.5 is the only difference.

$$Accept(\Delta C, T) = \mathbf{True} \Leftrightarrow random() < \frac{1}{1 + \exp(\frac{\Delta C}{0.5 C_0 T})} \tag{20}$$

The rationale is choosing an initial temperature where the longest single task will have a fair transition probability of being moved from one processor to another, and the same should hold true for the shortest single task with respect to final temperature.

Coefficient k has an approximate relation to p_f. Substituting $\frac{\Delta C_{\min}}{0.5 C_0}$ in place of $\Delta C_{\min}$ to make (10) compatible with (20) gives

$$T_f = \frac{\Delta C_{\min}}{0.5 C_0 \ln(\frac{1}{p_f} - 1)} < T \tag{21}$$

Now, $\frac{\Delta C_{\min}}{0.5C_0}$ is approximated with $\frac{t_{\min}}{t_{\max sum}}$ from (19)

$$T_f \sim \frac{t_{\min}}{t_{\max sum} \ln(\frac{1}{p_f} - 1)} < T \qquad (22)$$

Now comparing (19) and (22) we get the relation

$$k \sim \ln(\frac{1}{p_f} - 1) \qquad (23)$$

Solving (23) with respect to p_f gives us

$$p_f \sim \frac{1}{e^k + 1} \qquad (24)$$

For $k = 1$ the probability p_f to accept a worsening move on the final temperature level given a cost change of order $t_{\min}$ is approximately 27%. For $k = 2$, probability is 12%. As *k* increases p_f decreases exponentially. Suitable values for *k* are in range [1, 9] unless *L* is very large (hundreds of thousands or even millions of iterations). The temperature range implied by $k = 1$ is shown in Figure 5. The temperature range is calculated with (18) and (19). (Orsila et al., 2007) uses $k = 2$ and reaches are a local minimum more likely in the end, but it is more expensive than $k = 1$.

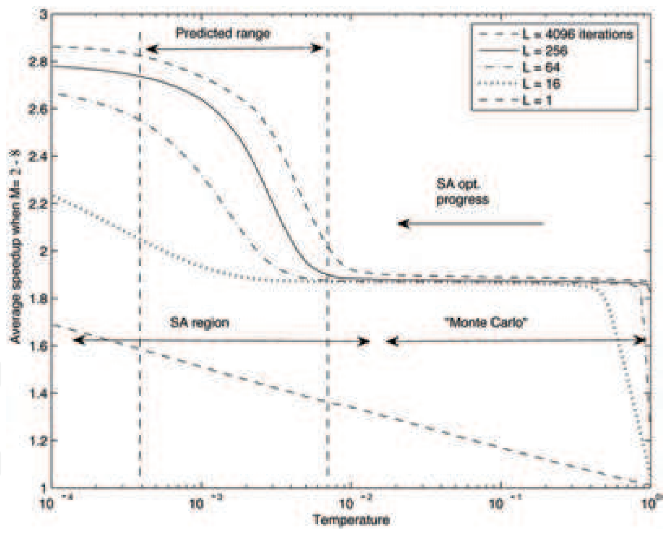

Figure 5. Averaged speedup with respect to temperature for *300* node graphs with different *L* values. The temperature given with (18)(19) *k = 1* is labeled „predicted range“. Notice that temperature and the number of iterations increase in different directions. The number of mapping iterations increases as the temperature decreases.

The end condition is the coupled temperature and rejection threshold (17) with $R_{\max} = \alpha$.

4.4 Ravindran case

(Ravindran, 2007) uses an exponential acceptance function (7).

A Koch temperature schedule (3) was used with parameters, including initial and final temperature, set manually. Termination condition is the temperature threshold (13).
Systematic methods are not used to tune parameters. However, the Koch temperature schedule is mitigating factor since it affects the number of temperature levels and iterations based on the problem.

4.5 Wild case

(Wild et al., 2003) use a geometric annealing schedule (1) with unknown parameters.
The termination condition is the uncoupled temperature and rejection threshold (16).
They show that an ECP move heuristics (Sec 3.7.1) is significantly better than the single move with directed acyclic graphs.
Systematic methods are not used to tune parameters.

5. Analysis and discussion

Following sections analyze the effect of iterations per temperature level, saving the number of iterations, give best practices for SA, and finally, SA is compared to two greedy algorithms and random mapping.

5.1 Iterations per temperature level

Figure 6 shows speedup of a $N = 300$ task directed acyclic graph with respect to iterations per temperature level L. Speedup is defined as $\frac{t_1}{t}$, where t is the execution time of the optimized solution on multiprocessor system and t_1 is the execution time on a single processor system.

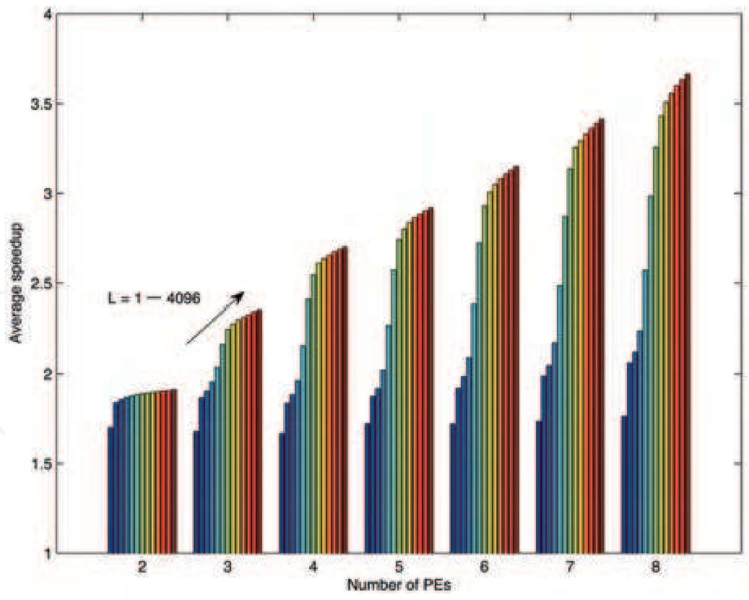

Figure 6. Averaged speedups for 300 node graphs with M=*2-8* processing elements and different L values (L = *1, 2, 4, ..., 4096*) for each processing element set.

Figure 7 shows the speedup and the number of iterations for each L. These figures show that having $L \geq \alpha = N(M-1) = [300, 600, 900, \ldots, 2100]$ for the number of processors $M = [2, 3, \ldots, 8]$ does not yield a significant improvement in performance but optimization time is increased heavily. Parameter $L = 1$ performs very poorly (Orsila et al., 2006).

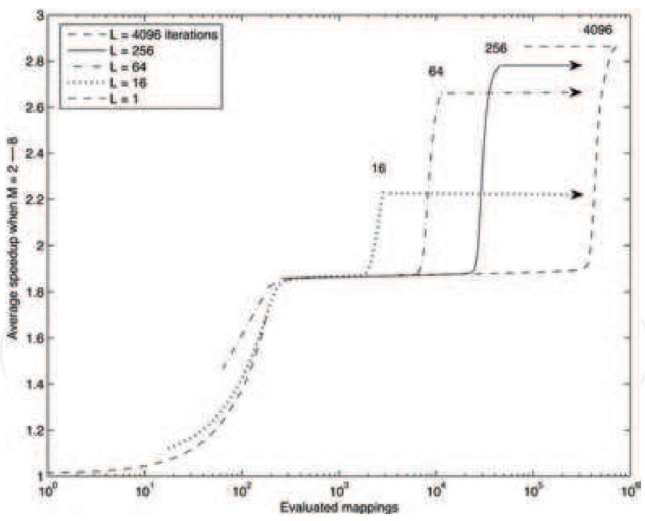

Figure 7. Averaged speedup with respect to mapping evaluations for 300 node graphs with different L values.

5.2 Saving optimization effort

Choosing initial temperature T_0 and final temperature T_f is crucial for saving optimization iterations. With too high an initial temperature the optimization process is practically Monte Carlo which means it converges very slowly, and thus, initial iterations are practically wasted because bad moves are accepted with too high a probability. This effect is visible in Figure 5 at high temperatures, i.e. $T > 10^{-2}$. Also, too low a probability reduces the annealing to greedy optimization. Greedy optimization becomes useless after a short time because it can not espace local minima. Therefore the final temperature must be set as high as possible without sacrificing the greedy part in optimization. This is the rationale for (Orsila et al., 2006) in Section 4.3.

5.3 Simulated annealing best practices

Based on our experiments, we have identified few rules of thumb for using SA to task mapping.

1. Choose the number of iterations per temperature level $L \geq \alpha = N(M-1)$, where N is the number of tasks and M is the number of PEs. Thus, α is the number of neighbouring mapping solutions because each of the N tasks could be relocated into at most $M-1$ alternatives.
2. Use geometric temperature schedule with $0.90 \leq q \leq 0.98$. This is the most common choice.
3. Device a systematic method for choosing the initial and final temperatures. As an example, see (10).
4. Use coupled temperature and rejection threshold as the end condition (Section 3.9.6) with $R_{\max} = L$ (the number of iterations per temperature level)
5. If in doubt, use the single task move (Sec 3.6.1). This is the most common choice. Other move heuristics can be very useful depending on the system. For example, ECP heuristics (Sec 3.7.1) is efficient for directed acyclic task graphs.
6. Use normalized inverse exponential function (6) as the acceptance function. This implies that temperature is always in range *(0, 1]*. This also means that convergence of

separate annealing problems can be compared with each other, and thus, effective annealing temperatures become more apparent through experiments.

7. Optimize the same problem many times. On each optimization run start with the best known solution so far. As simulated annealing is a probabilistic algorithm it can happen that the algorithm drives itself to a bad region in the optimization space. Running the algorithm several times reduces this risk.
8. If in doubt of any of the parameters, find them experimentally
9. Record the iteration number when the best solution was reached. If the termination iteration number is much higher than the best solution iteration, maybe the annealing can be made more efficient without sacrificing reliability.

5.4 Comparing SA to greedy algorithms

Figure 8 compares SA to two greedy algorithms and Random Mapping (Orsila et al., 2007). A *300* task application is distributed onto *8* processors to optimize execution time. Group Migration (GM) is a deterministic greedy algorithm that converges slowly. GM needs many iterations to achieve any speedup, but once that occurs, the speedup increases very rapidly. Optimal Subset Mapping (OSM) is a stochastic greedy algorithm that converges very rapidly. It reaches almost the maximum speedup level with very limited number of iterations. SA convergence speed is between GM and OSM but in the end it reaches a better solution. Random mapping saturates quickly and further iterations are unable to provide any speedup. Note that SA follows the random mapping line initially as it resembles a Monte Carlo process at high temperatures. Random mapping is the base reference for any mapping algorithm since any intelligent algorithm should do better than just random.

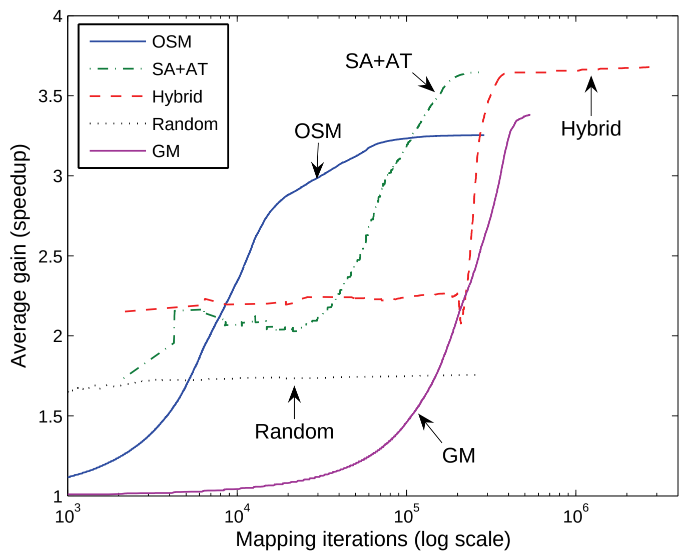

Figure 8. SA convergence speed compared to GM, OSM and Random Mapping algorithms for mapping *300* tasks to *8* processors. SA+AT is a Simulated Annealing algorithm presented in Section 4.3. GM and OSM are greedy heuristics.

SA yields *8%* better result than GM, *12%* better than OSM, and *107%* better than random mapping. SA is better than the greedy algorithms because it can espace local minima. However, when measuring the best speedup divided with the number of iterations needed to achieve the best result for each algorithm the relative order is different. We normalize the results so that random mapping gets value *1.00*. SA gets *2.58*, OSM *6.11* and GM *1.21*. That is, OSM is *2.4* times as efficient as SA is in terms of speedup divided by iterations. SA is *2.1* times as efficient as GM. Thus, we note that greedy local search methods can converge much faster than SA.

6. Open research challenges

This section identifies some open research challenges related to using SA for task mapping. The challenges are in order of importance.
What is the optimal annealing schedule for task mapping given a hardware, application model and a trade-off between solution quality and speed? The hardware and application model determine all possible cost changes in the system, and this is tied to probabilistic SA transitions. Not all temperatures are equally useful, so iterations can be saved by not annealing on irrelevant temperatures. For example, it is not benefitial to use lots of iterations at high temperatures because the process is essentially a Monte Carlo process which converges very slowly.
What are the best move heuristics for each type of application and hardware model? For example, ECP (Sec 3.7.1) is useful for application models that have the concept of critical path.
What is the optimal transition probability for $\Delta C = 0$? The probability is *0.5* in (5) and *1.0* in (7), but it can be selected arbitrarily. This probability determines the tendency at which SA travels equally good solutions in the neighborhood. Is there advantage to using either (5) or (7) due to this factor?
Can SA be made faster or better by first doing coarse-grain optimization on the application level and then continue with finer-grain optimization? Current optimization strategies are concerned with sequential small changes rather than employ a top-level strategy.
What are the relevant test cases for comparing SA to other algorithms, or other SA implementations? (Barr et al., 1995) have laid out good rules for comparing heuristics.
Excluding optimization programs, is there a problem where running SA as the main loop of the program would be benefitial? Each *Cost()* call would go one or several steps further in the program. In other words, is SA a feasible for run-time optimization rather than being used as an offline optimizer? Even small problems can take significant amount of iterations to get parameters correctly. The application must also tolerate slowdowns.

7. Conclusions

This chapter presents an overview of using SA for mapping application tasks to multiprocessor system. We analyze the different function variants needed in SA. Many choices are suboptimal with respect to iteration count or discouraged due to poor optimization results. We find that SA is a well performing algorithm if used properly, but in practice it is too often used badly. Hence, we present best practices for some of those and review the most relevant open research challenges.
For best practices we recommend following. Iterations per temperature level should depend on the problem size. Systematic methods should be used for the temperature range. Normalized inverse exponential function should be used.
For open research challenges we prioritize following. Find an optimal annealing schedule, move function and transition probabilities for each type of common task mapping problems. For example, it is possible to do critical path analysis for some task mapping problems.

8. References

Barr, R. S.; Golden, B. L & Kelly, J. P. & Resende, M. G. C. & Stewart, W. R. (1995). Designing and Reporting on Computational Experiments with heuristic Methods, Springer Journal of Heuristics, Vol. 1, No. 1, pp. 9-32, 1995.

Braun, T. D.; Siegel, H. J. & Beck, N. (2001). A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Systems, IEEE Journal of Parallel and Distributed Computing, Vol. 61, pp. 810-837, 2001.

Cerny, V. (1985). Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm, Journal of Opt. Theory Appl., Vol. 45, No. 1, pp. 41-51, 1985.

Coffman, E. G. Jr. & Graham, R. L. (1971). Optimal Scheduling for Two-Processor Systems, Springer Acta Informatica, Vol. 1, No. 3, pp. 200-213, September, 1971.

Coroyer, C. & Liu, Z. (1991). Effectiveness of Heuristics and Simulated Annealing for the Scheduling of Concurrent Tasks - An Empirical Comparison, Rapport de recherch´ de l'INRIA - Sophia Antipolis, No. 1379, 1991.

Kirkpatrick, S.; Gelatt, C. D. Jr. & Vecchi, M. P. (1983). Optimization by simulated annealing, Science, Vol. 200, No. 4598, pp. 671-680, 1983.

Koch, P. (1995). Strategies for Realistic and Efficient Static Scheduling of Data Independent Algorithms onto Multiple Digital Signal Processors. Technical report, The DSP Research Group, Institute for Electronic Systems, Aalborg University, Aalborg, Denmark, December 1995.

Kwok, Y.-K. & Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv., Vol. 31, No. 4, pp. 406-471, 1999.

Orsila, H.; Kangas, T. & Salminen, E. & Hämäläinen, T. D. (2006). Parameterizing Simulated Annealing for Distributing Task Graphs on Multiprocessor SoCs, International Symposium on System-on-Chip 2006, Tampere, Finland, November, pp. 1-4, 2006.

Orsila, H.; Salminen, E. & Hännikäinen, M. & Hämäläinen, T. D. (2007). Optimal Subset Mapping And Convergence Evaluation of Mapping Algorithms for Distributing Task Graphs on Multiprocessor SoC, International Symposium on System-on-Chip 2007, Tampere, Finland, November, pp. 1-6, 2007.

Ravindran, K. (2007). Task Allocation and Scheduling of Concurrent Applications to Multiprocessor Systems, PhD Thesis, UCB/EECS-2007-149, [online] http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-149.html, 2007.

Wikipedia. (2008). Embarrassingly Parallel, [online] http://en.wikipedia.org /wiki/Embarrassingly_parallel

Wikipedia. (2008). Kahn Process Network, [online] http://en.wikipedia.org /wiki/Kahn_Process_Network

Wild, T.; Brunnbauer, W. & Foag, J. & Pazos, N. (2003). Mapping and scheduling for architecture exploration of networking SoCs, Proc. 16th Int. Conference on VLSI Design, pp. 376-381, 2003.