Automated Memory-Aware Application Distribution for Multi-Processor System-On-Chips

Heikki Orsila, Tero Kangas, Erno Salminen, Timo D. Hämäläinen and Marko Hännikäinen
Tampere University of Technology, Institute of Digital and Computer Systems,
PO BOX 553, FIN-33101 Tampere, Finland
heikki.orsila@tut.fi

Abstract

Mapping of applications on a Multiprocessor System-on-Chip (MP-SoC) is a crucial step to optimize performance, energy and memory constraints at the same time. The problem is formulated as finding solutions to a cost function of the algorithm performing mapping and scheduling under strict constraints. Our solution is based on simultaneous optimization of execution time and memory consumption whereas traditional methods only concentrate on execution time. Applications are modeled as static acyclic task graphs that are mapped on MP-SoC with customized simulated annealing. The automated mapping in this paper is especially purposed for MP-SoC architecture exploration, which typically requires a large number of trials without human interaction. For this reason, a new parameter selection scheme for simulated annealing is proposed that sets task mapping specific optimization parameters automatically. The scheme bounds optimization iterations to a reasonable limit and defines an annealing schedule that scales up with application and architecture complexity. The presented parameter selection scheme compared to extensive optimization achieves 90% goodness in results with only 5% optimization time, which helps large-scale architecture exploration where optimization time is important. The optimization procedure is analyzed with simulated annealing, group migration and random mapping algorithms using test graphs from the Standard Task Graph Set. Simulated annealing is found better than other algorithms in terms of both optimization time and the result. Simultaneous time and memory optimization method with simulated annealing is shown to speed up execution by 63% without memory buffer size increase. As a comparison, optimizing only execution time yields 112% speedup, but also increases memory buffers by 49%.

Keywords: simulated annealing, task graph, memory optimization, mapping, multiprocessor

1. Introduction

The problem being solved is increasing performance and decreasing energy consumption of Multiprocessor System-on-Chip (MP-SoC). To achieve both goals, the overall computation should be distributed for parallel execution. However, the penalty of distribution is often an increased overall memory consumption, since multi-processing requires at least local processor caches or data buffers for maintaining efficient computing. Each Processing Element (PE) is responsible for performing computations for a subset of application tasks. Smallest memory buffers are achieved by running every task on the same PE, but it would not be distributed in that case. Therefore a trade-off between execution time and memory buffers is needed. On-chip memory is expensive in terms of area and energy, and thus memory is a new, important optimization target.

Application distribution is a problem of mapping tasks on separate processing elements (PEs) for parallel computation. Several proposals have been introduced over the years; the first ones being for the traditional supercomputing domain. An optimal solution has been proven to be an NP problem [1], and therefore a practical polynomial time heuristics is needed. In extreme cases heuristics are either too greedy to explore non-obvious solutions or not greedy enough to discover obvious. Most of the past distribution algorithms are not used to optimize memory consumption.

In a traditional super-computing domain, optimizing performance means also increasing network usage, which will at some point saturate the network capacity and the application performance. For MP-SoC, the interconnect congestion must be modeled carefully [2][3] so that the optimization method becomes aware of network capacity and performance, and is able to allocate network resources properly for all the distributed tasks.

Simulated annealing (SA) [4][5] is a widely used meta-algorithm to solve the application distribution problem. However, it does not provide a definite answer for the problem. Simulated annealing depends on many parameters, such as move heuristics, acceptance probabilities for bad moves, annealing schedule and terminal conditions. Selection of these parameters is often ignored in experimental papers. Specializations of the algorithm have been proposed for application distribution, but many crucial parameters have been left unexplained and undefined. An automated architecture exploration

tool using SA requires a parameter selection method to avoid manual tuning for each architecture and application trial. Automated SA parameter selection has not been previously addressed in the context of architecture exploration.

This paper presents three new contributions. The first is a new optimization method with a cost function containing memory consumption and execution time. It selects the annealing schedule, terminal conditions and acceptance probabilities to make simulated annealing efficient for the applied case. Second is an automatic parameter selection method for SA that scales up with application and system complexity. Third contribution is empirical comparison between three mapping algorithms, which are SA, modified group migration (Kernighan-Lin graph partitioning) algorithm and random mapping. SA without automatic parameter setting is also compared.

The outline of the paper is as follows. Section 2 presents the terminology. Section 3 analyzes problems and benefits of other systems, and compares them to our method. Section 4 explains our optimization framework where the method is applied. Section 5 presents our optimization method for automatic parameter selection and memory optimization. Section 6 presents the scheduling system that is used for the optimization method. The experiment that is used to validate and compare our method to other algorithms is presented in Section 7 and the results are presented in Section 8. Section 9 justifies the parameter selection method. And finally, Section 10 concludes the paper.

2. Terminology

Allocation, mapping and scheduling are phases in realizing application distribution on MP-SoC. Allocation determines how many and what kind of PEs and other resources there are. Mapping determines how the tasks are assigned to the allocated PEs. Task scheduling determines the execution order and timing of tasks on each PE, and communication scheduling determines the order and timing of transfers on each interconnect. The result of this process is a schedule, which determines execution time and memory consumption.

For mapping optimization, the application is modeled in this paper with static task graphs (STG). STG is a directed acyclic graph (DAG), where each node represents a finite deterministic time computation task. Simulating an application modeled as STG means traveling a directed acyclic graph from ancestors to all children unconditionally. Nodes of the graph are tasks, and edges present communication and data dependence among tasks.

Figure 1 shows an example of an STG. Weights of nodes, marked as values inside parenthesis, indicate computation costs in terms of execution time. In the example, node E has computational cost of 5. Each edge of the graph indicates a data dependency, for example B is data dependent on A. Weights of edges, marked as numbers attached to edges in the figure, indicate communication costs associated with the edges. The edge from C to F costs four units, which are data sizes for communication. The communication time will be determined by the interconnect bandwidth, its availability and data size. STGs are called static because connectivity of the graph, node weights and edge weights are fixed before run-time. Communication costs change between runs due to allocation and mapping thus affecting edge weights.

A relevant factor regarding achievable parallelism is the communication to computation ratio (CCR). It is defined as the average task graph edge weight divided by the average node weight (when converted to comparable units). By the definition, CCR over 1.0 means that computation resources can not be fully utilized because there is more communication than computation to do.

There is no conditional task execution in STG, but having conditional or probabilistic children (in selecting path in the graph) or run-time determined node weights would not change mapping algorithms discussed in this paper. However, a more complex application model, such as a probabilistic model, would make the scheduling problem harder and comparison of mapping algorithms a very large study.

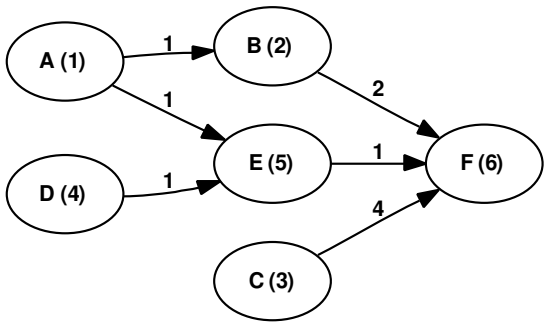


Figure 1. Example of a static task graph. Values in parenthesis inside the nodes represent execution time, and values at edges represent communication sizes. The communication time will be determined by the interconnect, its availability and data size.

Despite these limitations, STGs could be used to model, for example, a real-time MPEG4 encoder because real-time systems have bounded execution and communication costs for the worst case behavior. Many multimedia applications belong to real-time category and therefore applicability of STGs is wide. STGs are also widely used in related work, which helps comparisons.

Memory consumption of the application is partially determined by the STG. It is fully determined after the mapping and scheduling have been done. A node, or its associated PE in practice, must preserve its output data until it has been fully sent to all its children. Also, a set of nodes on the same PE must preserve input data until computation on all nodes that require the data are finished. Results or input data that are stored for a long period of time will increase memory buffer usage significantly. An alternative solution for this problem can possibly be found by task duplication or resending results. However, task duplication increases PE usage and resending results increases interconnect congestion, for which reason we do not consider these techniques in this paper. Results are sent only once and they are preserved on the target PE as long as they are needed. The results are discarded immediately when they are not needed. Thus memory consumption depends heavily on the mapping and scheduling of the application.

3. Related Work

Many existing optimization methods use stochastic algorithms such as SA [4][5] and genetic algorithms [6][7]. These algorithms have been applied on wide variety of hard optimization problems, but there is no consensus or common rules how to apply them on MP-SoC optimization. SA can be used with any cost function, but it is a meta-algorithm because parts of it have to be specialized for any given problem. The relevant mathematical properties for this paper are discussed in [19]. SA has been applied to many challenging problems, including traveling salesman problem [4], and mapping and scheduling of task graphs [8].

Wild et al. [2] compared SA, Tabu Search [9] and various algorithms for task distribution to achieve speedup. Their results showed 7.3% advantage for Tabu Search against SA, but they also stated they were not able to control the progress of SA. Their paper left many SA parameters undefined, such as initial temperature, final temperature, maximum rejections and acceptance function, which raises questions about accuracy of the comparison with respect to SA. Their method uses SA with geometric temperature schedule that decreases temperature proportionally between each iteration until a final temperature is reached and then optimization is terminated. As a consequence the number of iterations is fixed for a given initial temperature, and thus, the method does not scale up with application and system complexity.

Our method increases the number of iterations automatically when application and system complexity grows, which is more practical and less error-prone than setting parameters manually for many different scales of problems. The common feature for our and Wild et al. is the use of dynamically arbitrated shared bus with multiple PEs, as well as first mapping task graphs and then scheduling with a B-level scheduler. Their system is different in having HW accelerators in addition to PEs, and they did not present details about HW accelerator performance or CCR values of the graphs they used. Our paper focuses on CCR values in the range [0.5, 1.0], because computation resources are not wasted too much in that range, and it is also possible to achieve maximum parallelism in some cases.

Braun et al. [10] compared 11 different optimization algorithms for application distribution. They also compared Tabu Search and SA, and in their results, comparing to results from Wild et al., SA was better than Tabu Search in three out of six cases. Genetic algorithms gave the best results in their experiment. Their SA method had a benefit of scaling up with application complexity in terms of selecting initial temperature with a specific method. Our approach has the same benefit implemented with a normalization factor integrated into to acceptance probabilities.

The number of iterations in Braun's method does not scale up correctly because it is not affected by the number of tasks in application. As a bad side effect of their initial temperature selection method, the number of iterations for SA is affected by the absolute time rather than relative time of application tasks. This is avoided in our method by using a normalized temperature scale, which is made possible by the normalization factor in our acceptance probability function. Braun's annealing schedule was a geometric temperature with 10% temperature reduction on each iteration, implying that the temperature decreases fast. This has the consequence that one thousandth of initial temperature is reached in just 87 iterations, after which the optimization is very greedy. Thus, the radical exploration phase in SA, which means high temperatures, is not dependent on application complexity, and therefore the exploration phase may be too short for complex applications. Their method also lacks application adaptability because the maximum rejections has a fixed value of 150 iterations regardless of the application complexity. They used random node (RN) heuristics for randomizing new mappings, which is inefficient with small amount of PEs as described in Section 5.2.

Spinellis [11] showed an interesting SA approach for optimizing production line configurations in industrial manufacturing. Obviously the two fields are very different in practice, but theoretical problems and solutions are roughly the same. As a special case, both fields examine the task distribution problem to gain efficiency. Spinellis showed an automatic method for SA to select the number of iterations required for a given problem. Their method scaled up the number of iterations for each temperature level based on the problem size. A similar method is used in our method for task distribution. Unfortunately acceptance probabilities, meaning the dynamic temperature range, were not normalized to different problems.

Our paper presents a specialization of the SA meta-algorithm that addresses SA specific problems in previously mentioned papers. Initial temperature, final temperature and acceptance probabilities are normalized to standard levels by an automatic method that scales up both with application and allocation complexity. Furthermore, the total optimization effort is bounded by a polynomial function that depends on application and allocation complexity.

Group Migration (GM), also known as Kernighan-Lin algorithm, is a very successful algorithm for graph partitioning [12]. It has been compared to SA in [22] and suggested to be used for task mapping in [14]. Mapping is done by partitioning the task graph into several groups of tasks, each group being one PE. The algorithm was originally designed to partition graphs into two groups, but our paper uses a slightly modified version of the algorithm for arbitrary number of groups while preserving the old behavior for two groups. The idea to modify the algorithm is presented in [14] and an example is presented in [15].

Sinnen and Sousa [3] presented an application model that embeds the interconnect structure and contention into the optimization process. They presented a method to schedule communication onto a heterogeneous system. Wild et al. [2] showed a similar scheduling method to insert communications onto a network to optimize communications. Task graphs in their paper had 0.1, 1.0 and 10.0 as CCRs. They concluded that CCR of 10.0 was too challenging to be distributed and that effect of communication congestion model is quite visible already in the CCR value of 1.0. These findings support our similar experience. Our method also schedules communications based on priorities of tasks that will receive communication messages.

Also, Sinnen and Sousa [16] modeled side-effects of heavy communication in SoCs by adding communication overhead for PEs. Most parallel system models are unaware of PE load caused by communication itself. For example TCP/IP systems need to copy data buffers from kernel to application memory. Sinnen and Sousas model is insufficient, however, because it does not model behavior of interrupt-driven and DMA-driven communication, which are the most common mechanisms for high performance computing. Communication overhead should be extended over execution of multiple processes in a normal case of interrupt driven communication. In order to avoid bias of specific applications, our model does not have processor overhead for communication. Our model assesses potential of various mapping algorithms rather than effects of communication model.

Other approaches for the performance, memory and power optimization include task graph partitioning to automatically select system architecture for a given application. Hou et al. [25] presented a method for partitioning task graph onto a suitable architecture. The system presented in this paper does not partition task graphs, but the method in [25] is directly applicable to our optimization system as well.

Szymanek et al. [17] presented an algorithm to simultaneously speedup executing and optimize memory consumption of SoC applications. Their method keeps track of data and program memory requirements for different PEs to achieve the goal. It is based on constraint programming that sets hard limits on memory and time requirements, whereas our method only places relative cost on memory and time but allows all solutions.

Panda et al. [24] optimized the memory architecture to fit a given application. This paper's approach penalizes using memory on the application level while optimizing for performance. Methods discussed in [24] could be applied to the system presented in this paper as well. Their method operates on the system level and ours on the application level.

Kwok et al. did a benchmark on various scheduling algorithms [13] and they explained methods and theory of the same algorithms in [18]. We chose B-level scheduling policy for our method, because it was a common element in well performing scheduling algorithms. The MPC algorithm was the best scheduling policy in their benchmark on the bounded number of processors case, and it is based on the B-level scheduling.

Ascia et al. [23] used genetic algorithms to map applications onto a SoC obtaining a pareto-front of solutions for multiple cost functions. Their system allows the designer to choose the best solution from multiple controversial design goals. Their goal is however different than ours, because our system needs to pick a single good solution as part of an automated CAD system [26]. Other than that, multi-objective optimization and genetic algorithms would fit well into the system presented in this paper.

4. MP-SoC Application Distribution Framework

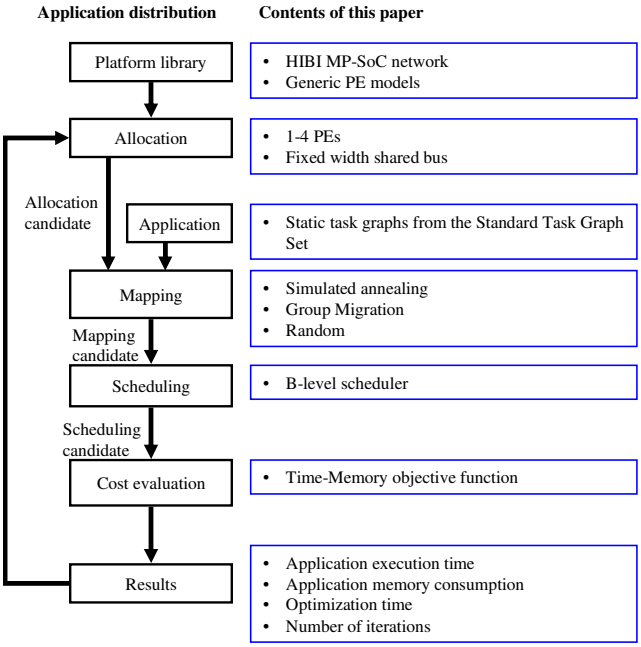


Figure 2. MP-SoC Application Distribution Framework.

Our MP-SoC application distribution framework is shown in Figure 2. The optimization process starts by selecting an allocation and an application, which are target hardware and a task graph. Allocation and the application are passed to a mapping algorithm. The mapping algorithm is chosen between group migration, random mapping or SA. The algorithm starts with an initial solution in which all task graph nodes are mapped to a single PE and then iterates through various mapping candidates to find a better solution. The mapping candidates are scheduled with a B-level scheduler to determine the execution time and memory usage. The B-level scheduler decides the execution order for each task on each PE to optimize execution time. The goodness of the candidate is evaluated by a cost function that depends on the execution time and memory consumption. At some point mapping algorithm stops and returns the final mapping solution.

In many optimization systems mapping and scheduling are dependent on each other [18], but in this system allocation, mapping and scheduling algorithms can be changed independently of each other to speedup prototyping of new optimization algorithms. The current system allows optimization of any task-based system where tasks are mapped onto PEs of a SoC consisting of arbitrary interconnection networks. Therefore dynamic graphs and complex application models can be optimized as well. Communication delays and the interconnection network can vary significantly and thus scheduling is separated from mapping. Communication delays for both shared and distributed memory architectures can be modeled as well.

The system uses a single cost function, but multi-objective systems analyzing pareto-fronts of multiple solutions [23] could implemented without affecting the optimization algorithms presented here.

5. Mapping Algorithms

5.1. Simulated Annealing Algorithm

The pseudo-code of the SA algorithm is presented in Figure 3, and explanations of symbols are given in Table 1. SA can not be used for task mapping without specializing it. The following parameters need to be chosen for a complete algorithm: annealing schedule, move heuristics, cost function, acceptance probability and the terminal condition.

The *Cost* function in Figure 3 evaluates the cost for any given state of the optimization space. Each point in the optimization space defines a mapping for the application. The optimization loop is terminated after $R_{\max}$ amount of consecutive moves that do not result into improvement in cost. The *Temperature-Cooling* function on Line 7 determines the annealing rate of the method which gets two parameters: T_0 is the initial temperature, and i is the iteration number. The *Move* function on Line 8 is a move heuristics to alter current mapping. It depends on the current state *S* and temperature *T*. The *Random* function returns a random number from the uniform probability distribution in range [0, 1). The *Prob* function determines the probability for accepting a move to a worse state. It depends on the current temperature and the increase of cost between old and new state.

```
SIMULATED-ANNEALING(S_0, T_0)
 1  S ← S_0
 2  C ← COST(S_0)
 3  S_best ← S
 4  C_best ← C
 5  R ← 0
 6  for i ← 1 to ∞
 7     do T ← TEMPERATURE-COOLING(T_0, i)
 8        S_new ← MOVE(S, T)
 9        C_new ← COST(S_new)
10        r ← RANDOM()
11        p ← PROB(C_new − C, T)
12        if C_new < C or r < p
13           then if C_new < C_best
14                   then S_best ← S_new
15                        C_best ← C_new
16                S ← S_new
17                C ← C_new
18                R ← 0
19           else  if T ≤ T_f
20                   then R ← R + 1
21                        if R ≥ R_max
22                           then break
23  return S_best
```

Figure 3. Simulated annealing algorithm.

Table 1. Summary of symbols of the simulated annealing pseudo-code.

C	Cost function value
C_{best}	Best cost function value
Cost()	Cost function
i	Iteration number
Move()	Move heuristics function
p	Probability value
Prob()	Probability function of accepting a bad move
r	Random value
Random()	Random variable function returning value in range [0, 1)
R	Number of consecutive move rejections
$R_{\max}$	Maximum number of rejections
S	Optimization state (mapping)
S_0	Initial state
S_{best}	Best state
T	Current temperature in range (0, 1]
T_0	Initial temperature (1.0 in this method)
T_f	Final temperature
Temperature-Cooling()	Annealing schedule function

5.2. Automatic parameter selection for SA

The choice of annealing schedule (*Temperature-Cooling* function) is an important factor for optimization [19]. Mathematical theory establishes that infinite number of iterations is needed to find a global optimum with SA. However, the mapping problem is finite but it is also an NP problem implying that a practical compromise has to be made that runs in polynomial time. Annealing schedule must be related to the terminal condition. We define the dynamic temperature scale to be $r = \log(\frac{T_0}{T_f})$ and assert that a terminal condition must not be true before a large scale of temperatures has been tried, because only a large scale r allows careful exploration of the search space. High temperatures will do rapid and aggressive exploration, and low temperatures will do efficient and greedy exploration. A common choice for the annealing schedule is a function that starts from an initial temperature, and the temperature decreases proportionally between each level until the final temperature is reached. The proportion value p is close to but smaller than 1. Thus the number of temperature levels is proportional to r. We will use this schedule.

The terminal condition of annealing must limit the computational complexity of the algorithm so that it is reasonable even with larger applications. The maximum number of iterations should grow as a polynomial number of problem factors rather than as an exponential number that is required for true optimum solution. The relevant factors for the number of iterations are initial temperature T_0, final temperature T_f, the number of tasks to be mapped N and the number of PEs M.

Further requirement for task mapping is that annealing schedule must scale with application and allocation complexity with respect to optimization ability. This means that the amount of iterations per temperature level must increase as the application and allocation size grows. For N tasks and M PEs one must choose a single move from $N(M-1)$ different alternatives, and thus it is logical that the amount of iterations per temperature level is at least proportional to this value. Considering these issues we define the system complexity as

$$L = N(M-1) \quad (1)$$

Thus the complexity measure is a product of application and allocation complexity. Furthermore we require that the number of iterations for each temperature level is L, because that is the number of choices for changing a single mapping on any given state. Also, we select the number of temperature levels according to dynamic temperature scale r.

It must be noted that SA will very unlikely try each alternative mapping for a temperature level even if the number of iterations is L because the heuristics move is a random function. Also, if a move is accepted then the mapping alternatives after that are based on the new state and therefore trying L different alternatives will not happen. In the case of frequent rejections due to bad states or low temperature, the L amount of moves gives similar behavior as the group migration algorithm, but not exactly the same because SA makes chains of moves instead of just trying different move alternatives. Also, SA does not share the limitations of group migration because it is not greedy until low temperature levels are reached.

The chosen terminal condition is true when the final temperature is reached and a specific amount of consecutive rejections happen. The maximum amount of consecutive rejections $R_{\max}$ should also scale with system complexity, and therefore it is chosen so that $R_{\max} = L$.

Based on these choices we get a formula to compute the total number of iterations for SA. The number of different temperature levels x in Equation (2) depends on the proportion p and the initial and final temperatures as

$$T_0 p^x = T_f \Rightarrow x = \frac{\log \frac{T_f}{T_0}}{\log p}. \quad (2)$$

The total number of iterations I_{total} is computed in Equation (3) based on the number of different temperature levels x and the number of tasks and PEs of the system as

$$\begin{aligned} I_{total} &= xL + R_{\max} = xL + L = (x+1)L \\ &= \left(\frac{\log \frac{T_f}{T_0}}{\log p} + 1\right) N(M-1) \end{aligned} \quad . \quad (3)$$

Therefore the total number of iterations is a function of N tasks, M PEs and the temperature scale r.

Equation (4) shows the annealing schedule function decreasing temperature geometrically every L iterations as

$$\text{Temperature-Cooling}(T_0, i) = T_0 p^{\left\lfloor \frac{i}{L} \right\rfloor}. \quad (4)$$

A common choice for acceptance probability is shown in Equation (5) as

$$\text{Basic-Probability}(\Delta C, T) = \frac{1}{1 + \exp \frac{\Delta C}{T}}. \quad (5)$$

ΔC is the increase in cost function value between two states. A bigger cost increase leads to lower probability, and thus moving to a much worse state is less probable than moving to a marginally worse state. The problem with this function is that it gives a different probability scale depending on the scale of the cost function values of a specific system. The system includes the platform and applications, and it would be a benefit to automatically tune the acceptance probability to the scale of the cost function.

We propose a solution to normalize the probability scale by adding a new term to the expression shown in Equation (6). C_0 is the initial cost function value of the optimization process. The term $\frac{\Delta C}{C_0}$ makes the state change probability relative to the initial cost C_0. This makes annealing comparable between different applications and cost functions. An additional assumption is that temperature T is in range (0, 1].

$$\text{Normalized-Probability}(\Delta C, T) = \frac{1}{1 + \exp \frac{\Delta C}{0.5 C_0 T}} \quad (6)$$

Figure 4 presents acceptance probabilities for the normalized probability function for relative cost function changes and temperatures. The probabilities lie in the range (0, 0.5]. As the relative cost change goes to zero, the probability goes to 0.5. Thus the function has a property of easily allowing SA to take many small steps that have only a minor worsening effect on the cost function. When the temperature decreases to near zero, these small worsening steps become very unlikely.

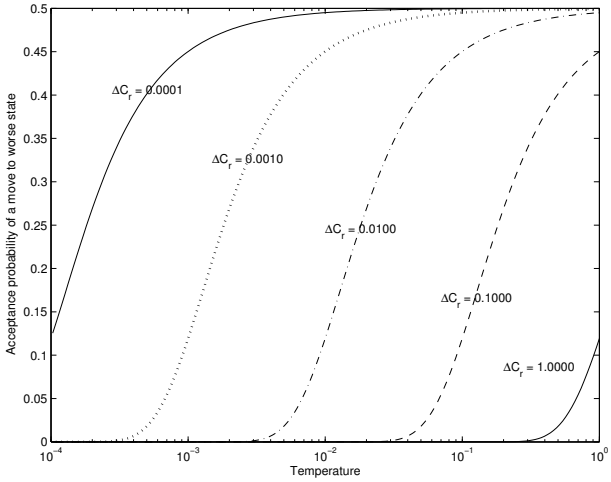


Figure 4. Acceptance probabilities for the normalized probability function with respect to relative cost function change $\Delta C_r = \frac{\Delta C}{C_0}$

A common choice for move heuristics [2][6][10] is a function that chooses one random task and maps that to a random PE. It is here named as the RN (Random Node) move heuristics and presented in Figure 5. It has the problem that it may do unnecessary work by randomizing exactly the same PE again for a task. In 2 PE allocation case probability for that is 50%. Despite this drawback, it is a very common choice as a move heuristics.

```
RN-MOVE(S, T)
1  S_new ← S
2  S_new[RANDOM-TASK()] ← RANDOM-ELEMENT(PEs)
3  return S_new
```

Figure 5. RN move heuristics moves one random task to a random PE.

The obvious deficiency in the RN heuristics is fixed by our RM move (Random Modifying move) heuristics presented in Figure 6. It avoids choosing the same PE and, thus, has a clear advantage over the RN heuristics when only a few PEs are used. This small detail has often been left unreported on other publications, but it is worth pointing out here.

```
RM-MOVE(S, T)
1  S_new ← S
2  t ← RANDOM-TASK()
3  S_new[t] ← RANDOM-ELEMENT(PEs without S[t])
4  return S_new
```

Figure 6. RM move heuristics moves one random to a different random PE.

Figure 7 shows an example of annealing process for optimizing execution time of a 50 node STG on 2 PEs. The effect of annealing from high temperature to a low temperature can be seen as the cost function altering less towards the end. At the end the optimization is almost purely greedy, and hence allows moves to a worse state with a very low probability. This figure shows the cost function value of the current accepted state rather than cost function values of all tried states. A figure showing cost function values of all tried states would be similar to a white noise function.

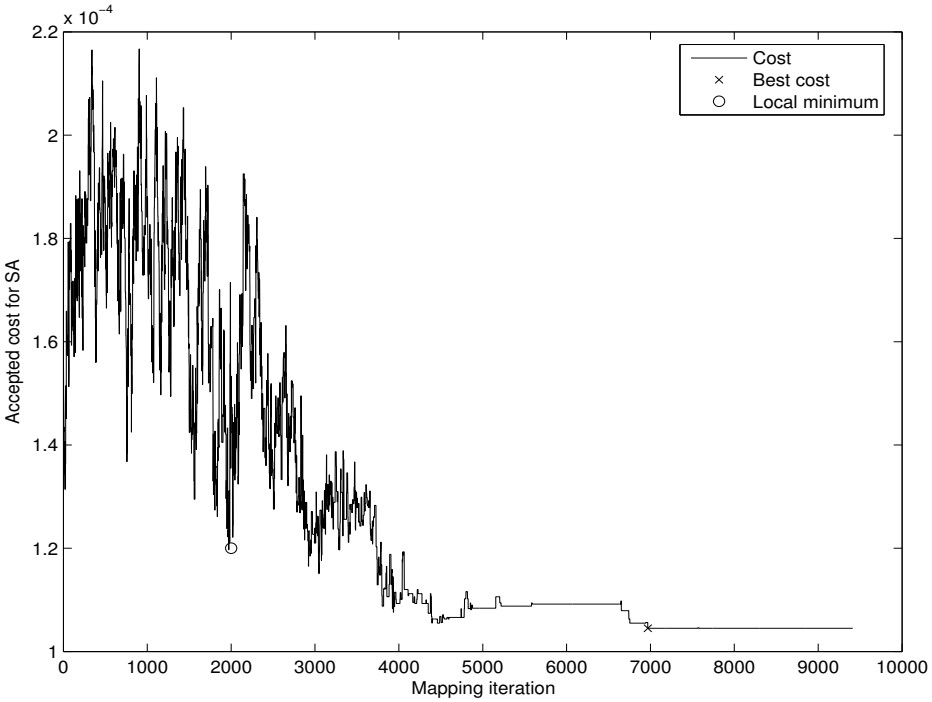


Figure 7. An example of annealing from a high to a low temperature state for a 50 node STG with 2 PEs. x marks the mapping iteration that has the lowest cost function value.

5.3. Group Migration Algorithm

Group migration is used to map task graphs onto multiple PEs in a greedy fashion. This is a generalization of the original algorithm [12] that partitioned graphs to two disjoint subsets. Our modified version partitions the graph into arbitrarily many disjoint subsets.

The algorithm is greedy because it only accepts moves to better positions, and thus it always gets stuck into a local minimum in the optimization space. The algorithm consists of migration rounds. Each round either improves the solution or keeps the original solution. Optimization ends when the latest round is unable to improve the solution. Usually the algorithm converges into a local minimum in less than five rounds [12].

A move in GM means moving one specific task to another PE. A round consists of sub-rounds which try all tasks on all other PEs. With N nodes and M PEs it is $(M-1)N$ tries. The best move on the sub-round is chosen for the next sub-round. Tasks, which have been moved as best task candidates on previous sub-rounds, are not moved anymore until the next round comes. There are at most N sub-rounds. This results into at most $(M-1)N^2$ moves per round. If no improving move is found on a sub-round, the round is terminated, because all possible single moves were already tried ($(M-1)N$ tries). The pseudo-code of the modified group migration algorithm is shown in Figure 8 and variables that are new compared to SA are explained in Table 2.

```
GROUP-MIGRATION(S)
 1  while True
 2     do S_new ← GM-ROUND(S)
 3        if COST(S_new) < COST(S)
 4           then S ← S_new
 5           else  break
 6  return S

GM-ROUND(S_0)
 1  S ← S_0
 2  C ← COST(S)
 3  Moved ← Boolean array of N falses
 4  for i ← 1 to N
 5     do D_task = NIL
 6        D_PE = NIL
 7        for t ← 0 to N − 1
 8           do if Moved[t] = True
 9                 then continue
10              A_old ← S[t]
11              for A ← 0 to M − 1
12                 do if A = A_old
13                       then continue
14                    S[t] ← A
15                    if COST(S) ≥ C
16                       then continue
17                    C = COST(S)
18                    D_task = t
19                    D_PE = A
20              S[t] ← A_old
21        if D_PE = NIL
22           then break
23        Moved[D_task] ← True
24        S[D_task] ← D_PE
25  return S
```

Figure 8. A modified group migration algorithm for arbitrary number of PEs.

Table 2. Summary of new variables used in the group migration algorithm. Other variables are the same as in SA.

A	PE
A_{old}	Old PE
D_{PE}	PE associated with the best migration candidate
D_{task}	Task associated with the best migration candidate
Moved	Array of Booleans indicating tasks that have been moved

The main loop of the pseudo-code begins on Line 1 of *Group-Migration* function. It calls *GM-Round* function as long as it can improve the mapping. *GM-Round* will change at most one mapping per round. If it does not change any, the optimization will terminate. *GM-Round* first computes the initial cost of initial state S_0 on Line 2. The *Cost* function is the same function as with SA. Line 3 initializes an array that marks all tasks as non-moved. The function can only move each task once from one PE to another, and this table keeps record of tasks that have been moved. Moved tasks can not be moved again until the next call to this function. The upper limit of loop variable *i* on Line 4 signifies that each task can only be moved once. Increasing the upper limit would not break or change the functional behavior. Line 7 begins a sub-round which ends at Line 20. The sub-round tries to move each task from one PE to every other PE. The algorithm accepts a move on Line 15, if it is better than any other previous move. The effect of any previous moves is discarded on Line 19, even if a move improved mapping. However, the best improving move is recorded for later use on Lines 17-18. Each move is a separate try that ignores other moves. If no improving move was found in the loop beginning on Line 7, the search is terminated for this round on Line 21. Otherwise, the best move is applied on Lines 23-24. The best found is returned on Line 25.

5.4. Random Mapping Algorithm

Random mapping is an algorithm that selects a random PE separately for all tasks on every invocation of the algorithm. The algorithm is a useful baseline comparison algorithm [14] against other algorithms since it is a policy neutral mapping algorithm that converges like Monte Carlo algorithms. The random mapping exposes the inherent parallelizability of any given application for a given number of iterations. It should be compared with other algorithms by giving the same number of iterations for both algorithms. Random mapping results are presented here to allow fair comparison of the SA and GM methods against any other mapping algorithms. Cost function ratio of SA and random mapping can be compared to the ratio of any other algorithm and random mapping.

6. Scheduling and Cost Functions

6.1. B-level Scheduler

The scheduler decides the execution order of tasks on each PE. If a task has not been executed yet, and it has all the data required for its execution, it is said to be *ready*. When several tasks are ready on a PE the scheduler selects the most important task to be executed first.

Tasks are abstracted as nodes in task graphs, and communication is abstracted as edges. Node and edge weights, which are cycle times and data sizes respectively, are converted into time units before scheduling. The conversion is possible because allocation and mapping are already known.

The scheduler applies B-level scheduling policy for ordering tasks on PEs. The priority of a node is its B-level value [18]. Higher value means more important. The B-level value of a node is the longest path from that node to an exit node, including exit nodes weight. Exit node is defined as a node that has no children. The length of the path is defined as the sum of node and edge weights along that path. For example, in Figure 1 the longest path from A to F is *1+1+5+1+6=14*, and thus B-level value of node A is *14*.

As an additional optimization for decreasing schedule length, the edges, which represent communication, are also scheduled on the communication channels. A priority of an edge is the weight of the edge added with B-level values of child nodes. Thus, children who have higher priority may also receive their input data more quickly.

Figure 9 shows pseudo-code for computing B-level priorities for an STG. Line 1 does a topological sort on the task graph. The topological sort means ordering the task graph into a list where the node A is before the node B if A is a descendant of B. Thus the list is ordered so that children come first. Line 2 iterates through all the tasks in that order. Line 3 sets default B-level value for a node, which is only useful if the node is an exit node. Array B contains known B-level values so far. If a node is not an exit node, the B-level is computed on Lines 4-9 to be the maximum B-level value with respect to its children. A B-level value with respect to a child is the sum of child's B-level value, edge weight towards the child, and the node weight of the parent node.

```
B-LEVEL-PRIORITIES(STG)
 1  TaskList ← TOPOLOGICAL-SORT(STG)
 2  for each task t in TaskList
 3      do B[t] ← NODE-WEIGHT(STG, t)
 4          for each child c of node t
 5              do w ← B[c]
 6                  w ← w + EDGE-WEIGHT(STG, t, c)
 7                  w ← w + NODE-WEIGHT(STG, t)
 8                  if B[t] < w
 9                    then B[t] ← w
10  return B
```

Figure 9. Algorithm to compute B-level values for nodes of a task graph.

It should be noted that although B-level value is dependent on communication costs, it does not model communication congestion. Despite this limitation, Kwok et al. show in [13] that the best bounded number of processor (BNP) class scheduling algorithm is based on ALAP (as late as possible) time. ALAP time of a node is defined as the difference of the critical path length and the B-level value of the node. An unscheduled node with the highest B-level value is by definition on the dynamic critical path, and therefore B-level priority determines ALAP time uniquely, and therefore B-level priority is an excellent choice.

The algorithmic complexity of the B-level priority computation is $O(E+V)$, where E is the number of edges and V is the number of nodes. However, the list scheduling algorithm complexity is $O((E+V)\log V)$, which is higher than the

complexity of computing B-level priorities, and therefore the complexity of the whole scheduling system is $O((E+V)\log V)$.

6.2. Cost function

In this paper, the optimization process measures two factors from a mapped system by scheduling the task graph. Execution time *T* and memory buffer size *S* are the properties which determine the cost function value of a given allocation and mapping.

The scheduler system simulates the task graph and architecture by executing each graph node parent before the child node is executed as well as delaying the execution of the child node until results from the parent nodes have arrived, thus determining the execution time *T* by behavioral simulation. The system is however not limited to behavioral simulation, but exact models on the underlying system could be used by changing the scheduler part.

The scheduler keeps track of buffers that are needed for the computation and communication to determine memory buffer size *S*. When a PE starts receiving a data block from another PE it needs to allocate a memory buffer to contain that data. The PE must preserve that data buffer as long as it is receiving the data or computing something based on that data. The receiving buffer is freed after the computation. When a PE starts computing it needs to allocate memory for the result. The result is freed when the computation is done and the result has been sent to other PEs.

As a result, when the full schedule of an STG has been simulated, the scheduler knows memory size S required for the whole architecture and the total execution time *T*. The mapping algorithm is orthogonal to the scheduler part in our system, which was the design goal of the optimization framework, and thus other optimization parameters could be easily added by just changing the scheduler part without affecting the mapping algorithms.

The cost function is chosen to optimize both execution time and required memory buffers, that is, minimize the cost function $aT+bS$, where a and b are constants. When both time and memory are optimized, parameters a and b are chosen so that on a single PE case both aT and bS are 0.5 and thus cost function has the value 1.0 in the beginning.

The motivation for including memory buffer size factor into the cost function is to minimize expensive on-chip buffer memory required for parallel computation. An embedded system designer may balance cost function factors to favor speedup or memory saving depending on which is more important for the given application. Adding more factors into the cost function will motivate for research on multi-objective optimization and can take advantage of pareto-optimization methods such as [23].

7. Experiments

7.1. Setup for the Experiment

The optimization software is written for the UNIX environment containing 3,600 lines of C code. On a 2.0 GHz AMD Athlon 64 computer, the software is able to evaluate 12000 mappings in a second for a 50 node STG with 119 edges, or 600000 nodes or 1.43 million edges in a second. The software did 2,094 mappings in a second for a 336 node STG with 1211 edges, or 703584 nodes or 2.54 million edges in a second. Thus the method scales well with the number of nodes and edges. Scalability follows from the computational complexity of the scheduling method because a single mapping iteration without scheduling is in $O(1)$ complexity class.

The experiment is divided into two categories. The first category is the execution time optimization, where the cost function is the execution time of the application. The second category is the execution time and memory buffer optimization, where the cost function is a combination of execution time and memory buffers. Both categories are tested with SA, GM and random mapping algorithms. Each algorithm is used with 1 to 4 PEs. The single PE case is the reference case that is compared with other allocations. For each allocation, 10 graphs with 50 nodes and 10 graphs with 100 nodes are tested. The graphs were random STGs with random weights. Each graph is optimized 10 times with exactly the same parameters. Thus, the total amount of optimizations is 2 (categories) * 3 (algorithms) * 3 (PE allocations) * 20 (STGs) * 10 (independent identical runs) = 3600 runs. Figure 10 depicts the whole experiment as a tree. Only one branch on each level is expanded in the figure, and all branches are identical in size.

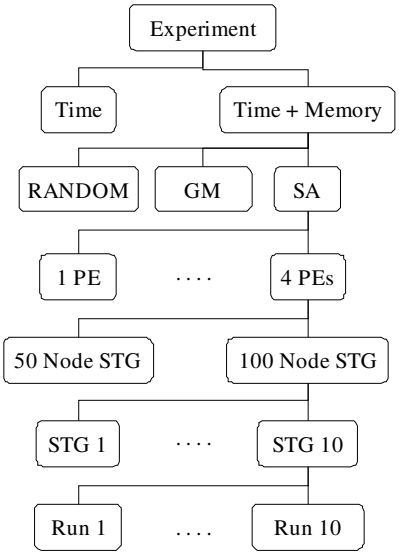


Figure 10. Outline of the experiment procedure.

7.2. Simulated Annealing Parameters

SA parameters are presented in Table 3. Final temperature is only 0.0001, and therefore SA is very greedy at the end of optimization as shown by the acceptance function plot in Figure 4.

Table 3. Parameters for simulated annealing.

Temperature proportion (*p*)	0.95
Initial temperature (T_0)	1.0
Final temperature (T_f)	0.0001
Iterations per temperature level	*L=N(M-1)* (Eq. (1))
Move heuristics	RM-move
Annealing schedule (*Temperature-Cooling*)	Temperature decreases proportionally once in *L* iterations.
Acceptance probability (*Prob*)	Normalized probability (Eq. (6))
Terminal condition	Final temperature is reached and at least *L* consecutive rejections are observed.

Table 4 shows the total number of mappings for one SA descend based on the parameters shown in Table 3.

Table 4. Total number of mappings for one simulated annealing descend.

	2 PEs	3 PEs	4 PEs
52 tasks	9300	18700	28000
102 tasks	18300	36600	54900

7.3. Group Migration and Random Mapping Parameters

Group migration algorithm does not need any parameters. This makes it a good algorithm to compare against other algorithms because it is also easy to implement and suitable for automatic architecture exploration. Relative advantage of the SA method over GM can be compared to any other algorithm over GM.

Random mapping tries N^2M^2 random mappings independently, which is much more than with SA. Random mapping is defined as choosing a random PE separately for all tasks. Random mapping presents a neutral reference to all other algorithms both in terms of behavior and efficiency. It does worse than other algorithms, as experiment will show, but it shows how much can be optimized without any knowledge of system behavior.

7.4. Static Task Graphs

STGs used in this paper are from Standard Task Graph Set [20]. The experiment uses 10 random STGs with 50 nodes and 10 random STGs with 100 nodes. The task graph numbers from 50 node graphs are: 2, 8, 18, 37, 43, 49, 97, 124, 131 and 146. The task graph numbers from 100 node graphs are: 0, 12, 15, 23, 46, 75, 76, 106, 128 and 136. Edge weights had to be added into the graphs because the Standard Task Graph collection graphs do not have them. In this case study, each edge presents communication of $Normal(64,16^2)$ bytes. That is, the edge weights are normally distributed with mean of 64 bytes and standard deviation of 16 bytes. The node weights were multiplied by a factor of 32 to have communication to computation ratio (CCR) at a reasonable level. The CCR is defined as the average edge weight divided by the average node weight. Summary of the properties is presented in Table 5.

Table 5. Summary of graph properties for the experiment.

Graphs	10 times 50 node STGs, 10 times 100 node STGs, from Standard Task Graph Set
Edge weights	Normally distributed: $Normal(64,16^2)$
Node weights	32 times the Standard Task Graph Set values

7.5. MP-SoC Execution Platform

The MP-SoC execution platform on which experiments are run is assumed to have a number of identical PEs as shown in Figure 11. Each PE is a 50 MHz general purpose processor (GPP), and thus it can execute any type of task from the TG and the mapping space does not have constraints. Task execution on a GPP is uninterruptible by the other tasks. IO operations are carried out in the background and they are assumed to be interrupt-driven. Task execution time is one or more GPP cycles. This is converted into time units by dividing the cycle number by the associated PE's clock frequency before scheduling.

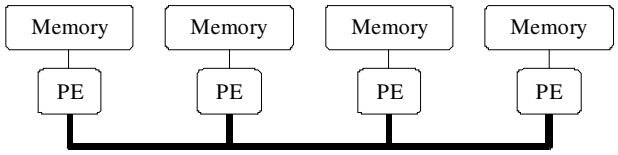


Figure 11. Block diagram of the MP-SoC execution platform.

The PEs are connected with a single dynamically arbitrated shared bus, and all the PEs are equally arbitrated on the bus. Transaction is defined as transferring one or more data words from one PE to another. Transactions are assumed to be atomic, meaning that they can not be interrupted. Arbitration policy for transactions is FIFO. Broadcast is not used in the system, i.e. there is exactly one receiver for each send. Bus data width is assumed to be 32 bits, and the bus can transfer its data width bits every cycle. Initial transfer latency for a transaction is modeled to be 8 bus cycles. This is the minimum latency for each bus transaction. This latency is used to prevent unrealistic bus utilization levels that do not occur in the real world. The bus operates on $2.5M$ MHz clock (M is the amount of PEs). The bus frequency is scaled with M because the amount of required communication is proportional to M. Not scaling with M would mean a significant bottleneck in the system. Summary of the system architecture is shown in Table 6.

Table 6. MP-SoC execution platform data.

PEs	1 to 4
PE frequency	50 MHz
Bus type	Dynamically arbitrated shared bus
Bus width	32 bits
Bus throughput	Bus width bits per cycle
Bus arbitration latency	8 cycles
Bus frequency	$2.5M$ MHz

It must be noted here that interconnect and GPP frequency are not relevant without the context of task graph parameters. The hardware was fixed at the specified level, and then task graph parameters were tuned to show relevant characteristics of optimization algorithms. With too fast hardware there would not be much competition among algorithms and distribution results would be overly positive. With too slow a hardware nothing could be distributed while gaining performance and algorithms would be seen as useless. Many experiments were carried out to choose these specific parameters.

Edge loads are converted from communication bytes into time units by

$$t = \frac{Lat + \left\lceil \frac{8Si}{W} \right\rceil}{f}, \quad (7)$$

where *Lat* is the arbitration latency of the bus in cycles, W is the bus width in bits, *Si* is the transfer size in bytes and *f* is the bus frequency.

CCR is computed by dividing the average edge weight with the average node weight of the graph. CCR values on 50 and 100 node graphs for 2, 3 and 4 PE cases are 0.98, 0.65 and 0.49 respectively. As the rationale in related work section explains, values near 1.0 are in the relevant range of parallel computing. CCR values could be chosen arbitrarily, but values lower than 0.1 would mean very well parallelizable problems, which are too easy cases to be considered here. However, values much higher than 1.0 would mean applications that can not be speeded up substantially. It should be noted that the CCR decreases with respect to the number of PEs in this paper, because the interconnect frequency is proportional to the number of PEs in the allocation phase.

8. Results

Results of the experiment are presented as speedup, gain and memory gain values. Speedup is defined as the execution time for a single PE case divided by the optimized execution time. Gain is defined to be the cost function value for the single PE case divided by the optimized cost function value. Memory gain measures memory usage for the single PE case divided by the optimized case. A higher gain value is always better. The following results compare two cases, which are the time optimization case and the memory-time optimization case. In the time optimization case, the cost function depends only on the execution time of the distributed application, but in the memory-time optimization case, the cost function depends on both execution time and memory buffer size. The complexity of algorithms is similar so that the number of iterations determines the runtime directly.

8.1. Time Optimization Comparison

Figure 12 presents the speedup values for each algorithm in the time optimization case with 2 to 4 PEs for 50 and 100 node graphs. The average speedups in combined 50 and 100 node cases for SA, GM and random mapping are 2.12, 2.03 and 1.59, respectively. Thus, SA wins GM by 4.4%, averaged over 20 random graphs that are each optimized 10 times. Also, SA finds the best speedup in 34% of iterations compared to GM, as shown in Figure 17. This means that SA converges significantly faster than GM. SA wins random mapping by 75%, and converges to the best solution in 74% of iterations. Random mapping is significantly worse than others.

8.2. Memory-time Optimization Comparison

Figure 13 shows the gain values for the memory-time optimization case with 2 to 4 PEs for 50 and 100 node task graphs. Average gains in combined 50 and 100 node cases for SA, GM and random mapping are 1.234, 1.208 and 1.051, respectively. Thus, SA wins GM by 2.2% and random by 17%. SA reaches the best solution in 12% of iterations compared to GM, which is significantly faster, as shown in Figure 18. The memory-gain units are small as numeric values, and their meaning must be analyzed separately in terms of speedup and memory usage.

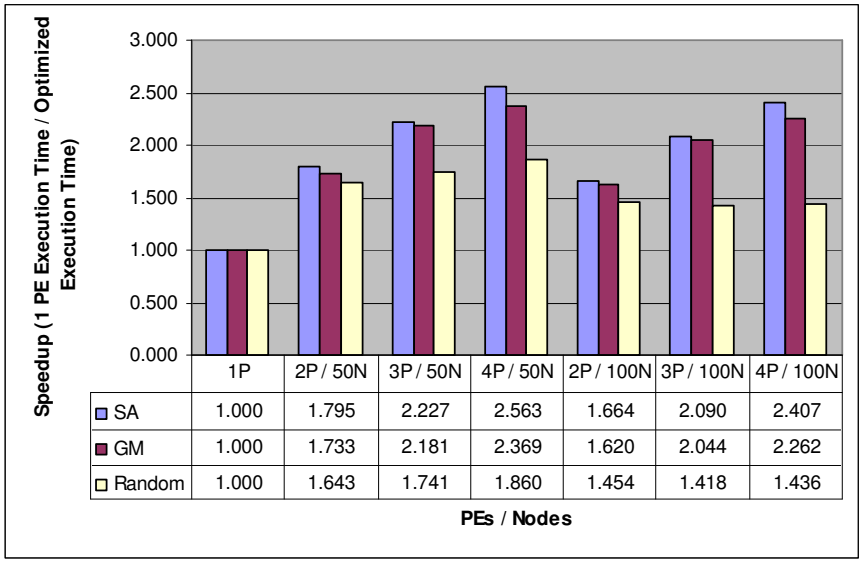
Figure 16 shows the memory gain values for the memory-time optimization case, and Figure 15 shows the same values for the time optimization case. Average memory gain values for the memory-time optimization case for SA, GM and random mappings are 1.00, 1.02 and 0.94, respectively. These numbers are significant, because computation parallelism was achieved without using more data buffer memory that needs to be expensive on-chip memory. Using external memory for data buffers would decrease throughput and increase latency, which should be avoided.

The GM case is interesting because it shows that computation actually uses 2% less memory for data buffers than a single PE. This happens because it is possible to off-load temporary results away from originating PE and then free buffers for other computation. Now comparing memory gain values to the time optimization case, where the averages memory gains are 0.67, 0.67 and 0.69, we can see that time optimization case uses 49%, 52% and 36% more data buffers to achieve their results. To validate that decreasing memory buffer sizes is useful the speedup values have to be analyzed as well. Figure 14 shows speedup values for memory-time optimization case. Average speedups for SA, GM and random mapping are 1.63, 1.52 and 1.36 respectively, which are 23%, 25% and 14% less than their time optimization counterparts. Therefore our method can give a noticeable speedup without need for additional on-chip memory buffers.

SoC design implications of memory-time optimization method can be analyzed by looking at absolute requirements for on-chip memory buffers. Consider a SoC video encoder performing motion estimation on 2 PEs with 16x16 pixel image blocks with 8 bits for each pixel [21]. The video encoder would have at least 10 image blocks stored on both PEs leading to total memory usage of 5120 bytes of memory, which also sets a realistic example for SoC data buffers. To compare that

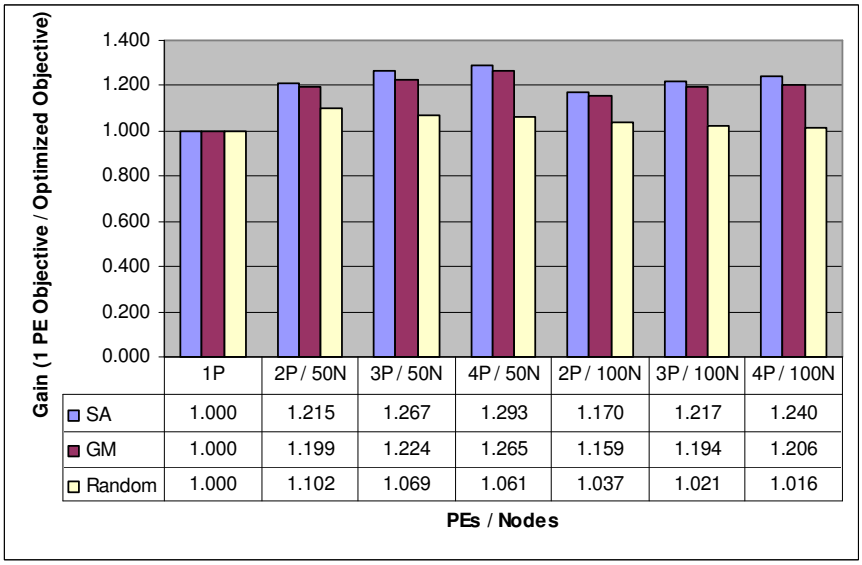
to our random graphs, 2312 bytes of memory buffers were used on average for 50 node case with 1 PE. Parallelizing that with the memory-time optimization method did not increase required memory, but optimizing purely for time increased memory buffer usage to 3451 bytes on average. Therefore 1139 bytes or 33% of on-chip memory was saved with our method, but approximately 23% of speed was lost compared to pure time optimization. This means that a realistic trade-off between fast execution time and reasonable memory usage is possible by choosing a proper cost function.

Random graphs used in this paper avoid bias towards specific applications, and the 33% of saved memory buffers are independent of the absolute size of the application. Thus bigger applications would save more memory in absolute terms. Similar results were obtained by Szymanek et al. [17], who optimized random graphs by a constraint based optimization method that penalized memory usage in a cost function. They compared the constraint based method to a greedy algorithm that optimized only execution time. The constraint method resulted into 18% less data memory but also 41% less speed compared to the greedy algorithm. As a difference to our method, Szymanek et al. also optimized code memory size. Their method was able to decrease code size by 33%. The code size does not apply to random graphs from Standard Task Graph Set, because they do not have functions defined for each node, and therefore it has to be assumed that each node is a separate function. Consequently, changing the mappings does not affect total code size.



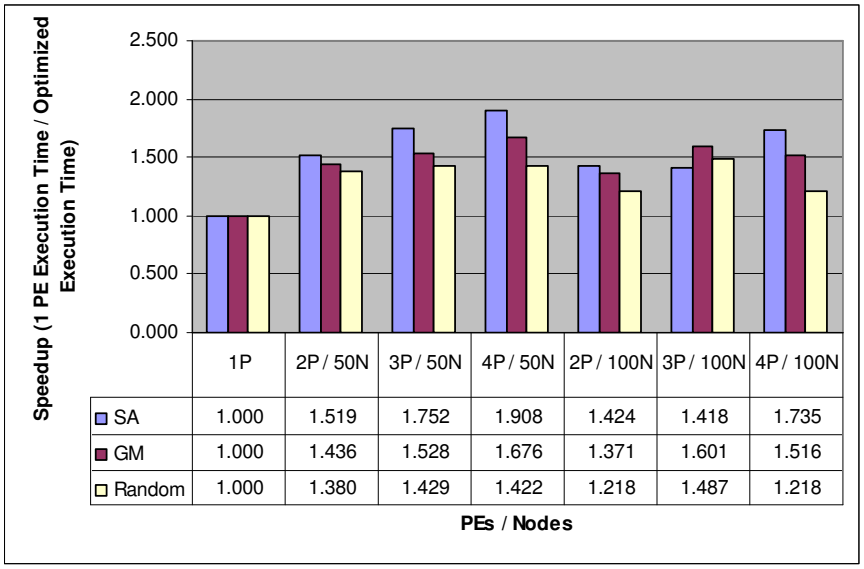
	1P	2P / 50N	3P / 50N	4P / 50N	2P / 100N	3P / 100N	4P / 100N
SA	1.000	1.795	2.227	2.563	1.664	2.090	2.407
GM	1.000	1.733	2.181	2.369	1.620	2.044	2.262
Random	1.000	1.643	1.741	1.860	1.454	1.418	1.436

Figure 12. Mean speedups on time optimization for 50 and 100 node graphs.



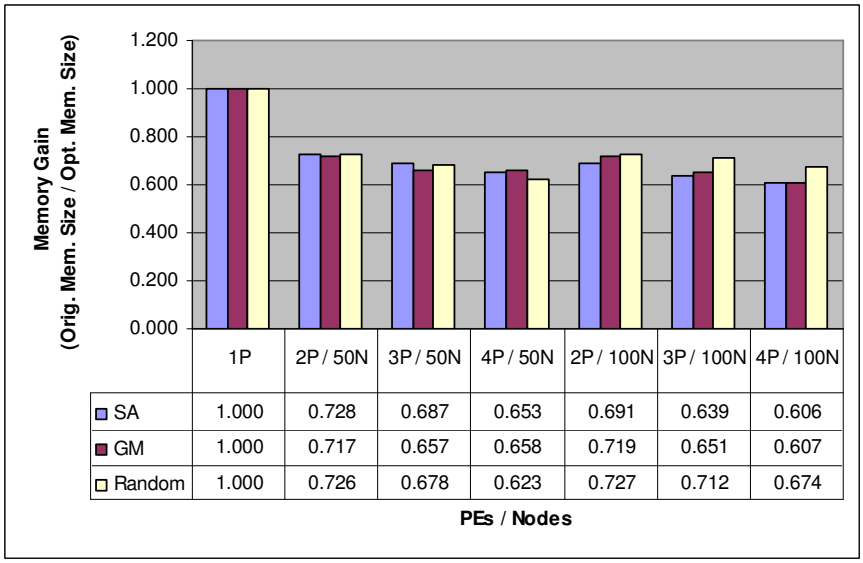
	1P	2P / 50N	3P / 50N	4P / 50N	2P / 100N	3P / 100N	4P / 100N
SA	1.000	1.215	1.267	1.293	1.170	1.217	1.240
GM	1.000	1.199	1.224	1.265	1.159	1.194	1.206
Random	1.000	1.102	1.069	1.061	1.037	1.021	1.016

Figure 13. Mean gains on memory and time optimization for 50 and 100 node graphs.



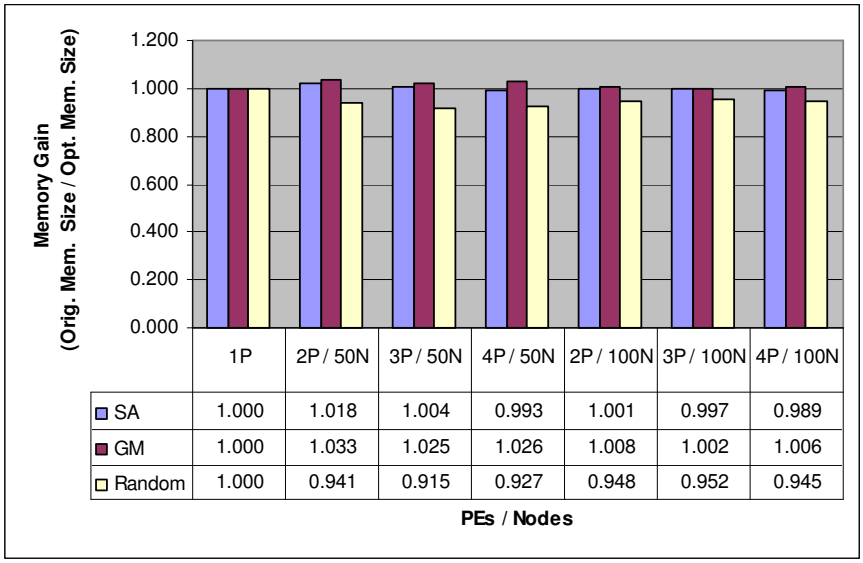
	1P	2P / 50N	3P / 50N	4P / 50N	2P / 100N	3P / 100N	4P / 100N
SA	1.000	1.519	1.752	1.908	1.424	1.418	1.735
GM	1.000	1.436	1.528	1.676	1.371	1.601	1.516
Random	1.000	1.380	1.429	1.422	1.218	1.487	1.218

Figure 14. Mean speedups on memory and time optimization for 50 and 100 node graphs.



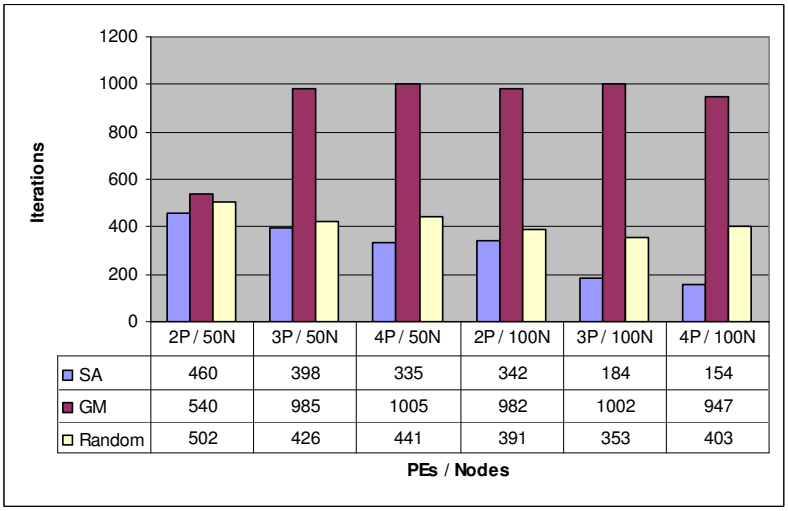
	1P	2P / 50N	3P / 50N	4P / 50N	2P / 100N	3P / 100N	4P / 100N
SA	1.000	0.728	0.687	0.653	0.691	0.639	0.606
GM	1.000	0.717	0.657	0.658	0.719	0.651	0.607
Random	1.000	0.726	0.678	0.623	0.727	0.712	0.674

Figure 15. Mean memory gains on time optimization for 50 and 100 node graphs.



	1P	2P / 50N	3P / 50N	4P / 50N	2P / 100N	3P / 100N	4P / 100N
SA	1.000	1.018	1.004	0.993	1.001	0.997	0.989
GM	1.000	1.033	1.025	1.026	1.008	1.002	1.006
Random	1.000	0.941	0.915	0.927	0.948	0.952	0.945

Figure 16. Mean memory gains on memory and time optimization for 50 and 100 node graphs.



	2P / 50N	3P / 50N	4P / 50N	2P / 100N	3P / 100N	4P / 100N
SA	460	398	335	342	184	154
GM	540	985	1005	982	1002	947
Random	502	426	441	391	353	403

Figure 17. Mean iterations to reach best solution on time optimization for 50 and 100 node graphs.

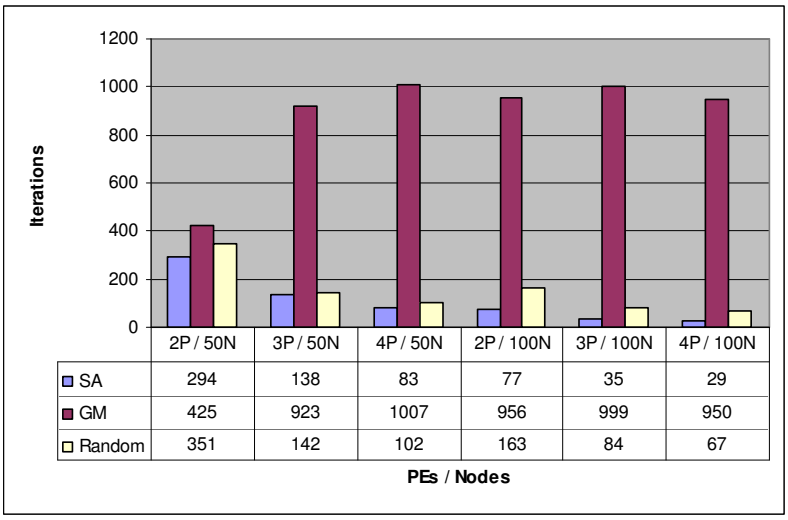


Figure 18. Mean iterations to reach best solution on memory and time optimization for 50 and 100 node graphs.

9. Parameter selection

To analyze the effect of Equation (1) on speedup and gain, we ran Section 7 experiment for 50 and 100 node graphs with different values of $L \in \{1, 2, 4, \ldots, 4096\}$ (powers of two). Each graph was optimized 10 times independently to estimate the statistical effect of the L parameter.

By theory, it is trivial that no fixed L value can perform well for arbitrary sized graphs and architectures because not even trivial mappings can be tried in a fixed number of iterations. Therefore, L must be a function of graph and architecture size. This was the origin of the parameter selection scheme, and the experimental evidence is given below.

Figure 19 and Figure 20 show the effect of L parameter for memory and time optimization with 50 and 100 nodes respectively. In the 100 node case the gain value curve increases steeper than in the 50 node case, as L increases from one to more iterations. This shows clearly that more nodes requires more iterations to reach equal gain. These figures also show that increasing M, the number of PEs, makes the climb steeper. This implies that more iterations are needed as M increases. Moreover, these figures show that selecting $L = N(M-1)$ is enough iterations to climb the steepest hill. The behavior is similar for the time optimization case as shown in Figure 21.

Table 7 shows that relative gain of 85% to 94% is achieved in 2.4% to 7.3% iterations with the parameter selection scheme when compared to selecting 4096 iterations. Speedup and gain are almost saturated at 4096 iterations, and thus, it is the reference for maximum obtainable gain and speedup. This shows that the parameter selection scheme yields fast optimization with relatively good results. From the figures, it must be stressed that L must be at least linear to graph and architecture size to reach good gains.

The impact of our parameter selection scheme is also evaluated in [27].

Table 7 Effect of L parameter on speedup and gain for time and memory+time optimization cases respectively. Relative gain shows the effect of L parameter with respect to maximum iterations 4096. N is the number of nodes and M is the number of PEs.

Opt. type	N	M	$L = N(M-1)$	$\frac{L}{4096}$	Gain (L)	Gain (4096)	Relative gain $\frac{G(L)-1}{G(4096)-1}$
Memory + Time	50	2	50	1.2%	1.210	1.246	0.854
Memory + Time	50	3	100	2.4%	1.268	1.303	0.884
Memory + Time	50	4	150	3.7%	1.295	1.331	0.891
Memory + Time	100	2	100	2.4%	1.172	1.195	0.882
Memory + Time	100	3	200	4.9%	1.217	1.240	0.904
Memory + Time	100	4	300	7.3%	1.240	1.261	0.920
Time	100	2	100	2.4%	1.661	1.709	0.932
Time	100	3	200	4.9%	2.085	2.152	0.942
Time	100	4	300	7.3%	2.409	2.492	0.944

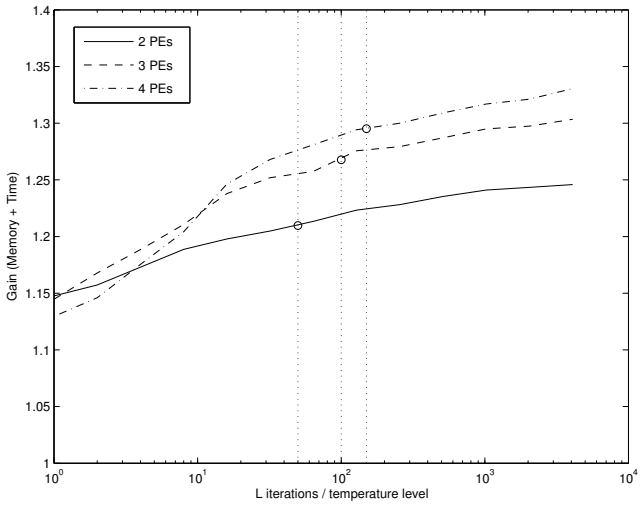


Figure 19 Effect of L parameter (log scale) for memory and time optimizing 50 node graphs.

The circles mark $L = N(M-1)$ case for each PE number.

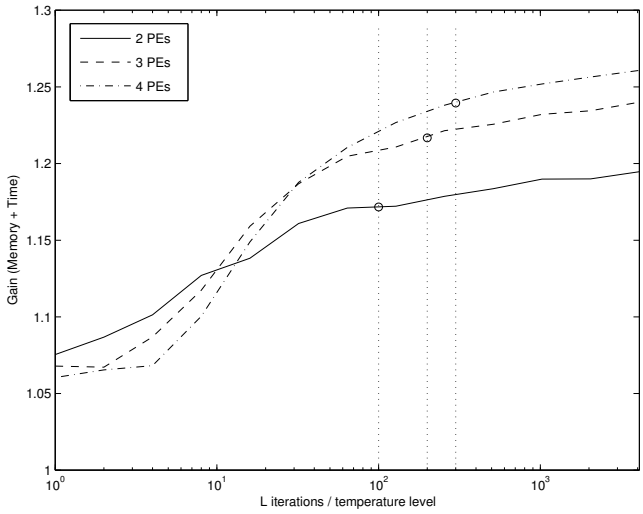


Figure 20 Effect of L parameter (log scale) for memory and time optimizing 100 node graphs.

The circles mark $L = N(M-1)$ case for each PE number.

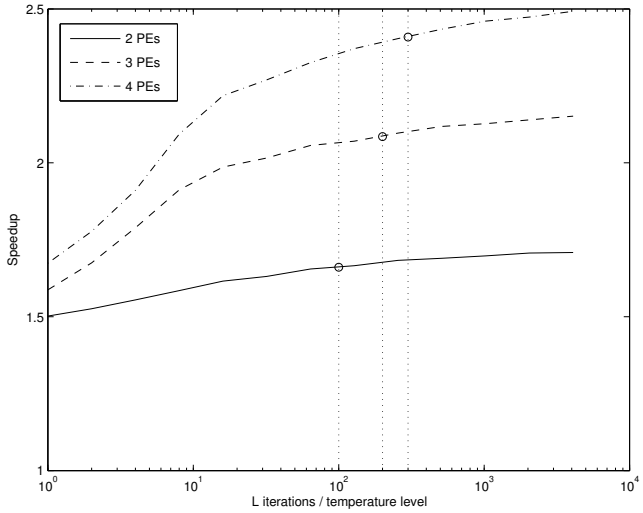


Figure 21 Effect of L parameter (log scale) for time optimizing 100 node graphs.

The circles mark $L = N(M-1)$ case for each PE number.

10. Conclusions

This paper presents an optimization method that carries out memory buffer and execution time optimization simultaneously. Results are compared with three task mapping algorithms, which are simulated annealing (SA), group migration and random mapping. The mapping algorithms presented are applicable to a wide range of task distribution problems, for both static and dynamic task graphs. The SA algorithm is shown to be the best algorithm in terms of optimized cost function value and convergence rate. Simultaneous time and memory optimization method with SA is shown to speed up execution by 63% without memory buffer size increase. As a comparison, optimizing the execution time only speeds up the application by 112% but also increases memory buffer sizes by 49%. Therefore a trade-off between our method and the pure time optimization case is 33% of saved on-chip memory but 23% loss in execution speed.

This paper also presents a unique method to automatically select SA parameters based on the problem complexity which consists of hardware and application factors. Therefore, the error-prone manual tuning of optimization parameters can be avoided by using our method, and optimization results can be improved by better fitting optimization parameters to

the complex problem. It is experimentally shown that the number of iterations for SA must be at least linear to the number of application graph nodes and the number of PEs to reach good results.

Future work should study the task distribution problem of minimizing iterations to reach near optimum results with SA, instead of just focusing on the final cost function value. Also, the next logical step is to evaluate the method on dynamic task graphs. In addition, more mapping algorithms, such as genetic algorithms and Tabu Search, should be tested with our memory optimization method, and automatic selection of free parameters should be devised for those algorithms as well. Also, adding more factors into the cost function motivates research on multi-objective optimization to fulfill additional design restrictions of the system.

11. References

[1] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors", MIT Press, 1989.

[2] T. Wild, W. Brunnbauer, J. Foag, N. Pazos, "Mapping and Scheduling for Architecture Exploration of Networking SoCs", Proceedings of the 16th International Conference on VLSI Design, pp. 376-381, 2003.

[3] O. Sinnen, L. Sousa, "Communication Contention in Task Scheduling", IEEE Transactions on Parallel and Distributed Systems, Vol. 16, Issue 6, pp. 503-515, 2005.

[4] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi, "Optimization by Simulated Annealing", Science, Vol. 200, No. 4598, pp. 671-680, 1983.

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

[6] A.S. Wu, H. Yu, S. Jin, K.-C. Lin, G.A. Schiavone, "An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling", IEEE Transactions on Parallel and Distributed Systems, Vol. 15, Issue 9, pp. 824-834, 2004.

[7] T. Lei, S. Kumar, "A Two-step Genetic Algorithm for Mapping Task Graphs to a Network on Chip Architecture", Proceedings of the Euromicro Symposium on Digital System Design (DSD'03), pp. 180-187, 2003.

[8] C. Coroyer, Z. Liu, "Effectiveness of Heuristics and Simulated Annealing for the Scheduling of Concurrent Tasks – An Empirical Comparison", Rapport de recherché de l'INRIA – Sophia Antipolis, No. 1379, 1991.

[9] F. Glover, E. Taillard, D. de Werra, "A User's Guide to Tabu Search", Annals of Operations Research, Vol. 21, pp. 3-28, 1993.

[10] T.D. Braun, H.J. Siegel, N. Beck, "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.

[11] D. Spinellis, "Large Production Line Optimization Using Simulated Annealing", Journal of Production Research, Vol. 38, No. 3, pp. 509-541, 2000.

[12] B.W. Kernighan, S. Lin, "An Efficient Heuristics Procedure for Partitioning Graphs", The Bell System Technical Journal, Vol. 49, No. 2, pp. 291-307, 1970.

[13] Y.-K. Kwok, I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms", Journal of Parallel and Distributed Computing, Vol. 59, No. 3, pp. 381-422, 1999.

[14] D. Gajski, F. Vahid, S. Narayan, J. Gong, "Specification and Design of Embedded Systems", Prentice Hall, ISBN: 0131507311, 1994.

[15] H. Orsila, T. Kangas, T.D. Hämäläinen, "Hybrid Algorithm for Mapping Static Task Graphs on Multiprocessor SoCs", International Symposium on System-on-Chip (SoC 2005), 2005.

[16] O. Sinnen, L. Sousa, "Task Scheduling: Considering the Processor Involvement in Communication", Proceedings of the 3rd International Workshop on Algorithms, Models, and Tools for Parallel Computing on Heterogeneous Networks, ISPDC'04/HetroPar'04, pp. 328-335, 2004.

[17] R. Szymanek, K. Kuchcinski, "A Constructive Algorithm for Memory-aware Task Assignment and Scheduling", International Conference on Hardware Software Codesign, Proceedings of the 9th International Symposium on Hardware/Software Codesign, pp. 147-152, 2001.

[18] Y.-K. Kwok, I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors", ACM Computing Survey, Vol. 31, No. 4, pp. 406-471, 1999.

[19] L. Ingber, "Very Fast Simulated Re-Annealing", Mathematical and Computer Modelling, Vol. 12, No. 8, pp. 967-973, 1989.

[20] Standard Task Graph Set, http://www.kasahara.elec.waseda.ac.jp/schedule/, 2005-12-20.

[21] O. Lehtoranta, E. Salminen, A. Kulmala, M. Hännikäinen, T. D. Hämäläinen, "A Parallel MPEG-4 Encoder for FPGA Based Multiprocessor SoC", Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL2005), pp. 380-385, 2005.

[22] T. Bui, C. Heighman, C. Jones, T. Leighton, "Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms", Proceedings of the 26th ACM / IEEE Conference on Design Automation, pp. 775-778, 1989.

[23] G. Ascia, V. Catania, M. Palesi, "An Evolutionary Approach to Network-on-Chip Mapping Problem", The 2005 IEEE Congress on Evolutionary Computation, Vol. 1, pp. 112-119, 2005.

[24] P.R. Panda, N.D. Dutt, A. Nicolau, "Local Memory Exploration and Optimization in Embedded Systems", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, Issue 1, pp. 3-13, 1999.

[25] J. Hou, W. Wolf, “Process Partitioning for Distributed Embedded Systems”, Proceedings of the Fourth International Workshop on Hardware/Software Co-Design (Codes/CASHE ’96), pp. 70-76, 1996.
[26] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, J. Riihimäki, K. Kuusilinna, "UML-based Multi-Processor SoC Design Framework", Transactions on Embedded Computing Systems, 2006, ACM, Accepted and to appear.
[27] H. Orsila, T. Kangas, E. Salminen, T. D. Hämäläinen, "Parameterizing Simulated Annealing for Distributing Task Graphs on Multiprocessor SoCs", International Symposium on System-on-Chip 2006, Tampere, Finland, November 14-16, 2006.

AUTHOR BIOGRAPHY:

Heikki Orsila, M.Sc. 2004, Tampere University of Technology (TUT), Finland. He is currently pursuing Doctor of Technology degree and working as a research scientist in the DACI research group in the Institute of Digital and Computer Systems at TUT. His research interests include parallel computation, optimization, operating systems and emulation.

Tero Kangas, M.Sc. 2001, Tampere University of Technology (TUT). Since 1999 he has been working as a research scientist in the Institute of Digital and Computer Systems (DCS) at TUT. Currently he is working towards his PhD degree and his main research topics are SoC architectures and design methodologies in multimedia applications.

Erno Salminen, M.Sc. 2001, TUT. Currently he is working towards his PhD degree in the Institute of Digital and Computer Systems (DCS) at TUT. His main research interests are digital systems design and communication issues in SoCs.

Timo D. Hämäläinen (M.Sc. 1993, Ph.D. 1997, TUT) acted as a senior research scientist and project manager at TUT in 1997-2001. He was nominated to full professor at TUT/Institute of Digital and Computer Systems in 2001. He heads the DACI research group that focuses on three main lines: wireless local area networking and wireless sensor networks, high-performance DSP/HW based video encoding, and interconnection networks with design flow tools for heterogeneous SoC platforms.

Marko Hännikäinen (M.Sc. 1998, Ph.D. 2002, TUT) acts as a senior research scientist in the Institute of Digital and Computer Systems at TUT, and a project manager in the DACI research group. His research interests include wireless local and personal area networking, wireless sensor and ac-hoc networks, and novel web services.