Heikki Orsila

Optimizing Algorithms for Task Graph Mapping on Multiprocessor System on Chip

Thesis for the degree of Doctor of Science in Technology
Tampere University of Technology
April 2011

Contact Information:

Heikki Orsila

Tampere University of Technology
Department of Computer Systems
P.O.Box 553
FIN-33101 Tampere
Finland
tel: +358407325989
email: heikki.orsila@iki.fi

ABSTRACT

The objective of this Thesis is to analyze and improve MPSoC design space exploration, specifically the task mapping using Simulated Annealing (SA) with fully automatic optimization. The work concentrates mostly on application execution time optimization. However, a trade-off between memory buffer and time optimization and a trade-off between power and time optimization is also analyzed. Applications are represented as public Standard Task Graph sets and Kahn Process Networks (KPNs).

Main focus is on SA as the optimization algorithm for task mapping. A state of the art survey is presented on using SA for task mapping. Thesis analyzes the impact of SA parameters on convergence. A systematic method is presented for automatically selecting parameters. The method scales up with respect to increasing HW and SW complexity. It is called Simulated Annealing with Automatic Temperature (SA+AT).

Optimal subset mapping (OSM) algorithm is presented as a rapidly converging task mapping algorithm. SA+AT is compared with OSM, Group Migration, Random Mapping and a Genetic Algorithm. SA+AT gives the best results. OSM is the most efficient algorithm.

Thesis presents new data on the convergence of annealing. Answers are given to global optimum convergence properties and the convergence speed in terms of mapping iterations. This covers optimization of the run-time of mapping algorithms so that a trade-off can be made between solution quality and algorithm's execution time. SA+AT saves up to half the optimization time without significantly decreasing solution quality.

Recommendations are presented for using SA in task mapping. The work is intended to help other works to use SA. *DCS task mapper*, an open source simulator and optimization program, is presented and published to help development and evaluation of mapping algorithms. The data to reproduce Thesis experiments with the program is also published.

PREFACE

The work presented in this Thesis has been carried out in the Department of Computer Systems at Tampere University of Technology during the years 2003-2011.

I thank my supervisor Prof. *Timo D. Hämäläinen* for guiding and encouraging me towards doctoral degree. Grateful acknowledgments go also to Prof. *Andy Pimentel* and Adj. Prof. *Juha Plosila* for thorough reviews and constructive comments on the manuscript.

It has been my pleasure to work in the Department of Computer Systems. Many thanks to all my colleagues for the discussions and good atmosphere. Especially Dr. *Erno Salminen*, Dr. *Kimmo Kuusilinna*, Prof. *Marko Hännikäinen* and Dr. *Tero Kangas*, deserve a big hand for helping me in several matters. In addition, I would like to thank all the other co-authors and colleagues, *Tero Arpinen*, M.Sc. *Kalle Holma*, M.Sc., *Tero Huttunen*, M.Sc., Dr. *Ari Kulmala*, *Janne Kulmala*, *Petri Pesonen*, M.Sc., and *Riku Uusikartano*, M.Sc., with whom I have had pleasant and stimulating discussions on and off-topic.

I want to thank my parents for the support they have given me. I also want to thank *Susanna* for love and understanding.

Tampere, April 2011
Heikki Orsila

TABLE OF CONTENTS

LIST OF PUBLICATIONS

This Thesis is based on the following publications: [P1]-[P7]. The Thesis also contains some unpublished material.

[P1] 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), pp. 146-150, 2005.

[P2] 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, pp. 73-76, Tampere, Finland, Nov 14-16, 2006.

[P3] H. Orsila, T. Kangas, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, *Automated Memory-Aware Application Distribution for Multi-Processor System-On-Chips*, Journal of Systems Architecture, Volume 53, Issue 11, ISSN 1383-7621, pp. 795-815, 2007.

[P4] H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, *Optimal Subset Mapping And Convergence Evaluation of Mapping Algorithms for Distributing Task Graphs on Multiprocessor SoC*, International Symposium on System-on-Chip, Tampere, Finland, 2007, pp. 52-57.

[P5] H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, *Evaluation of Heterogeneous Multiprocessor Architectures by Energy and Performance Optimization*, International Symposium on System-on-Chip 2008, pp. 157-162, Tampere, Finland, Nov 4-6, 2008.

[P6] H. Orsila, E. Salminen, T. D. Hämäläinen, *Best Practices for Simulated Annealing in Multiprocessor Task Distribution Problems*, Chapter 16 of the

Book “Simulated Annealing”, ISBN 978-953-7619-07-7, I-Tech Education and Publishing KG, pp. 321-342, 2008.

[P7] H. Orsila, E. Salminen, T. D. Hämäläinen, *Parameterizing Simulated Annealing for Distributing Kahn Process Networks on Multiprocessor SoCs*, International Symposium on System-on-Chip 2009, Tampere, Finland, Oct 5-7, 2009.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACO	Ant Colony Optimization
CP	Critical Path
CPU	Central Processing Unit
DA	Decomposition Approach (constraint programming)
DAG	Directed Acyclic Graph
DLS	Dynamic Level Scheduling
DMA	Direct memory access
DSE	Design Space Exploration
DSP	Digital Signal Processing
FIFO	First In First Out
FP	Fast pre-mapping
FSM	Finite State Machine
GA	Genetic Algorithm
GM	Group Migration
GPU	Graphics Processing Unit
GSA	Genetic Simulated Annealing
HA	Hybrid Algorithm
HTG	Hierarchical Task Graph

HW	Hardware
iff	If and only if
ISS	Instruction Set Simulator
KPN	Kahn Process Network
LIFO	Last In First Out (stack)
LP	Linear Programming
LS	Local Search
MoC	Model of Computation
MPSoC	Multiprocessor System-on-Chip
OSM	Optimal Subset Mapping
PE	Processing Element (CPU, acceleration unit, etc.)
PIO	Programmed IO; Processor is directly involved with IO rather than DMA controller.
ReCA	Reference Constructive Algorithm
RM	Random Mapping
SA	Simulated Annealing
SA+AT	Simulated Annealing with Automatic Temperature
SoC	System-on-Chip
ST	Single task move heuristics
STG	Static Task Graph
SW	Software
TS	Tabu Search

1. INTRODUCTION

A multiprocessor system-on-chip (*MPSoC*) consists of processors, memories, accelerators and interconnects. Examples include the *Cell*, *TI OMAP*, *ST Nomadik SA* [87], *HIBI* [47] [54] [66]. MPSoC design requires *design space exploration* (DSE) [30] to find an appropriate system meeting several requirements that may be mutually conflicting, e.g. energy-efficiency, cost and performance.

An MPSoC may contain tens of processing elements (PEs) and thousands of software components. Implementing the SoC and applications requires many person-years of work. The complexity of the system requires high-level planning, feasibility estimation and design to meet the requirements and reduce the risk of the project.

1.1 MPSoC design flow

High-level SoC design flows are used to decrease the investment risk in development. Figure 1 shows the Koski [39] design flow for MPSoCs at a high level as a representative design flow.

The first design phase defines the abstractions and functional components of the system. UML is used to model the system components. Y-model [41] [42] is applied on the system being designed. Architecture and applications are modeled separately but combined together into a system by *mapping*.

A functional model of the system is created in the second phase. Application code is written, or generated if possible, and functionally verified. Performance of application components are estimated by using benchmarks, test vectors and profiling. Parts of the application may later be implemented in HW to optimize performance.

The third and fourth design phase contains the design space exploration part. *Problem space* defines the system parameters being searched for in exploration. This includes

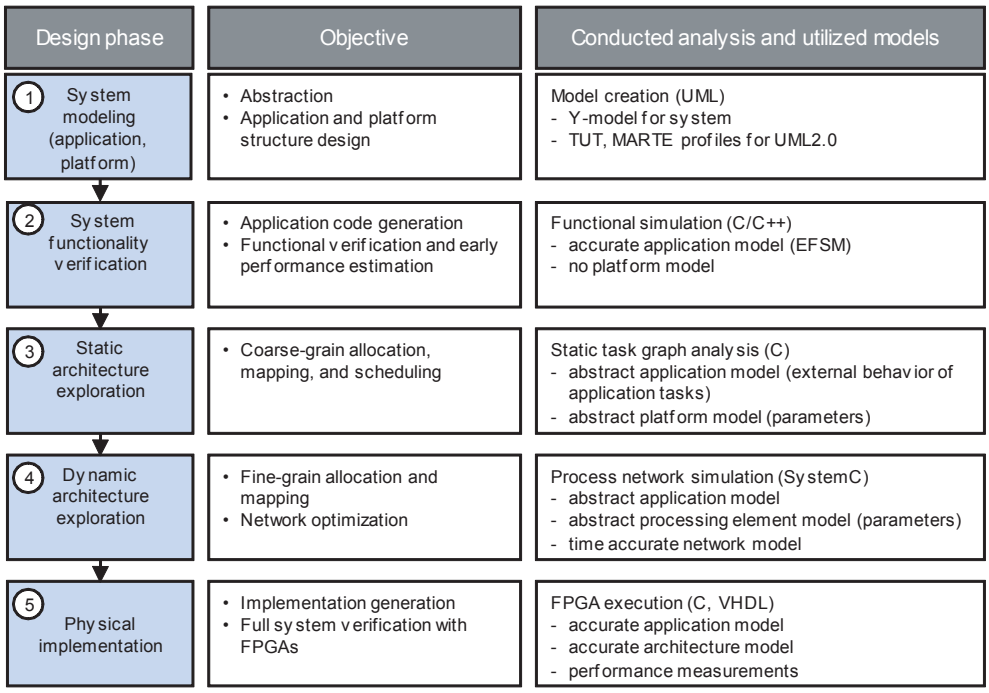


Fig. 1. *Koski design flow for multiprocessor SoCs [38]*

parameters such as the number of PEs, memory hierarchy, clock frequency, timing events and placing application tasks to PEs. *Objective space* determines the properties that are the effects of design choices in the problem space. This includes properties such as execution time, communication time, power and silicon area. Problem space properties do not directly affect the properties in the objective space. Exploration searches for a combination of problem space properties to optimize *objective space* properties. The third and fourth phases differ by the accuracy of objective space measures in the simulation model, the fourth is more accurate.

The system is implemented in finer grain with a SystemC model in the fourth design phase. The allocation, schedule and mapping is taken as input from the third phase. The system is optimized again. A significantly fewer number of systems is evaluated compare to third phase, because system evaluation is much slower now.

The fifth phase is physical implementation. The system is implemented and verified for FPGA. The application is finalized in C and hardware in VHDL. Accurate objective space measures are finally obtained.

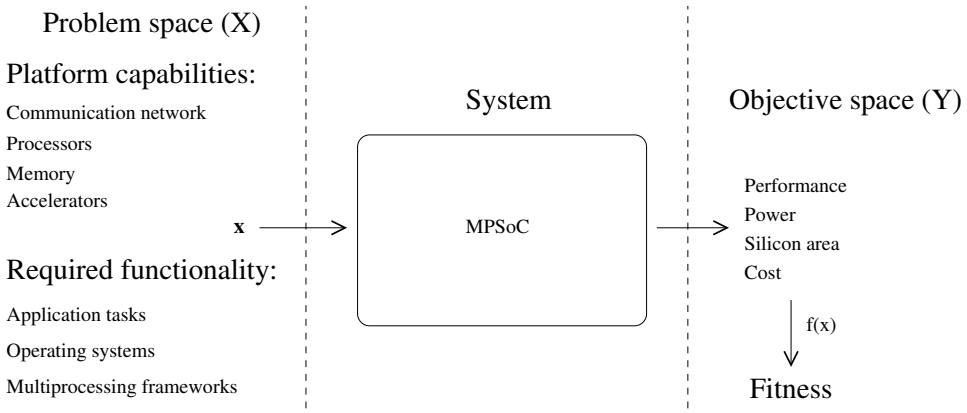


Fig. 2. *MPSoC implementation starts from the problem space that has platform capabilities and required functionality. A point x is selected from the problem space X. The point x defines a system which is implemented and evaluated with respect to measures in objective space Y. These measures include performance, power, silicon area and cost. Objective space measures define the objective function value $f(x)$, which is the fitness of the system.*

1.2 The mapping problem

The *mapping problem*, a part of the exploration process, means finding $x \in X$ to optimize the solution of an *objective function* $f(x) \in R^n$, where X is the problem space and x is the choice of free design parameters. Objective function uses simulation and estimation techniques to compute objective space (Y) measures from x. The resulting value is a multi-objective optimization problem if $n > 1$. Figure 2 shows the relation between problem and objective space.

Figure 3 shows the mapping process in higher detail. The first stage is system allocation where the system is implemented by combining existing components and/or creating new ones for evaluation. The system is then implemented by mapping selected components to HW resources. Figure 4 shows tasks being mapped to PEs and PEs connected with communication network. This yields a system candidate that can be evaluated. Evaluation part determines *objective space* measures of the system. Note, the evaluated system may or may not be feasible according to given objective space constraints such as execution time, power and area. Pareto optimal feasible systems are selected as the output of the process. In single objective case there is only one such system.

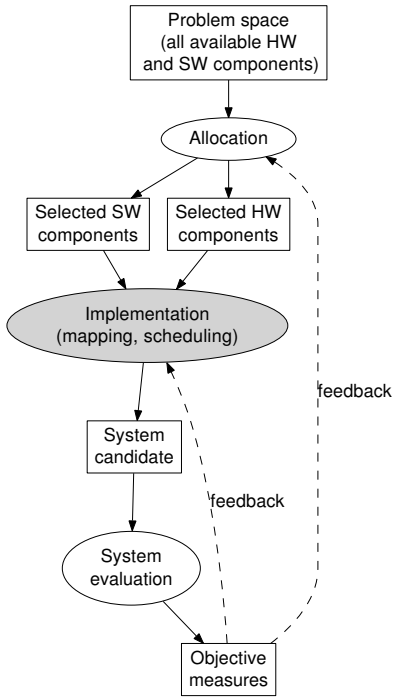


Fig. 3. *The mapping process. Boxes indicate data and ellipses are operations.*

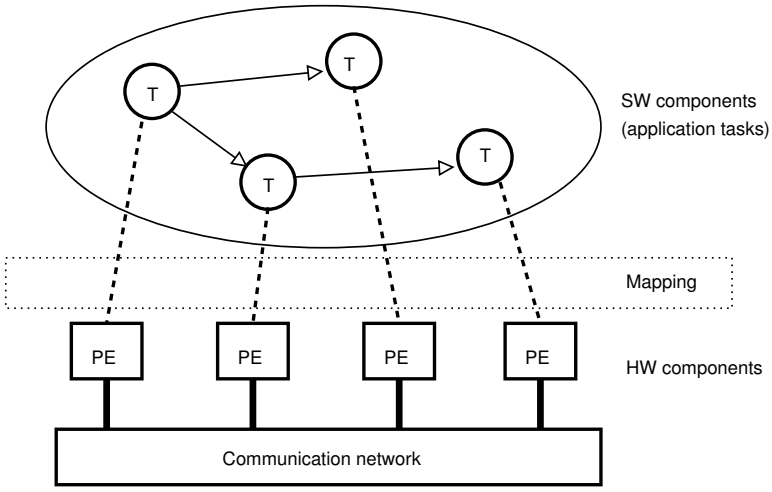


Fig. 4. *SW components (application tasks) are mapped to HW components. T denotes an application task. PE denotes a processing element.*

The application consists of *tasks* that are mapped to PEs. A *task* is defined to be any executable entity or a specific job that needs to be processed so that the application can serve its purpose. This includes operations such as computing output from input, copying data from one PE to another, mandatory time delay, waiting for an event to happen. A task has a mapping target, such as PE, task execution priority, execution time slot, communication channel to use, or any discrete value that represents a design choice for a particular task. The mapping target can be determined during system creation or execution time. A number of mappings is evaluated for each system candidate. The number of mappings depends on the properties of the system candidate, including the number of tasks and PEs.

The problem is finding a mapping from N objects to M resources with possible constraints. This means M^N points in the problem space. For example, the problem space for 32 tasks mapped to just 2 PEs has 4.3×10^9 points. It takes 136 years to explore the whole problem space if evaluating one mapping takes one second. Therefore, it is not possible to explore the whole problem space for anything but small problems. This means the space must be pruned or reduced. A fast optimization procedure is desired in order to cover a reasonable number of points in the problem space. This is the reason for separate third and fourth stages in Koski. However, a fast procedure comes with the expense of accuracy in objective space measurements, e.g. estimated execution time and power.

Several algorithmic approaches have been proposed for task mapping [9] [20]. The main approaches are non-iterative and iterative heuristics. Non-iterative heuristics compute a single solution which is not further refined. Simulation is only used for estimating execution times of application tasks and performance of PEs. Tasks are mapped to PEs based on execution time estimates and the structure of the task graph. List scheduling heuristics [51] are often used. Good results from these methods usually require that the critical path of the application is known.

The iterative approaches are often stochastic methods that do random walk in the problem space. Iterative approaches give better results than non-iterative approaches because they can try multiple solutions and use simulations instead of estimates for evaluation. Several iterative algorithms have been proposed for task mapping. Most of them are based on heuristics such as Simulated Annealing (SA) [12] [44], Genetic Algorithms (GA) [31], Tabu Search (TS) [28] [7], local search heuristics, load balancing techniques, and Monte Carlo. These methods have also been combined

together. For example, both SA and GA are popular approaches that have been combined together into a Genetic Simulated Annealing (GSA) [17] [77].

Parameters of the published algorithms have often not been reported, the effect of different parameter choices is not analyzed, or the algorithms are too slow. Proper parameters vary from task to task, but manual parameter selection is laborious and error-prone. Therefore, systematic methods and analysis on parameter selection is preferable in exploration. Development of these methods and analysis is the central research problem in the Thesis. Automatic parameter selection methods are presented for SA, and the effects of parameters are analyzed. SA was selected for the Thesis because it has been successfully applied on many fields including task mapping, but there is a lack of information on the effects of SA parameters on task mapping.

1.3 Objectives and scope of research

The objective of this Thesis is to analyze and improve MPSoC design space exploration, specifically the task mapping using Simulated Annealing with fully automatic optimization. The work concentrates mostly on application execution time optimization. However, [P3] considers a trade-off between memory buffer and time optimization and [P5] between power and time optimization.

MPSoCs used in this Thesis are modeled on external behavior level. Timings and resource contention is simulated, but actual data is not computed. This allows rapid scheduling, and hence, rapid evaluation of different mappings during the development work. Applications are represented as public Standard Task Graph sets [81] and Kahn Process Networks (KPNs) [37] generated with *kpn-generator* [46] with varying level of optimizing difficulty. The graphs are selected to avoid bias for being application specific, since optimal mapping algorithm depends on the graph topology and weights for a given application. This Thesis tries to overcome this and find general purpose mapping methods that are suitable to many types of applications.

Main focus is on Simulated Annealing as the optimization algorithm for task mapping. First, this Thesis analyzes the impact of SA parameters. Many publications leave parameter selection unexplained, and often not documented. This harms attempts of automatic exploration as manual tuning of parameters is required, which is also error-prone due to humans. This motivates finding a systematic method for

automatically selecting parameters.

Second, Thesis gives answers to global optimum convergence properties and the convergence speed in terms of mapping iterations. Optimal solution for the task mapping problem requires exponential time with respect to number of nodes in the task graph. Spending more iterations to a solve a problem gives diminishing returns. The effectiveness of an algorithm usually starts to dampen exponentially after some number of iterations. These properties have not been examined carefully in task mapping problems previously.

Third, the Thesis covers optimization of the run-time of mapping algorithms so that a trade-off can be made between solution quality and algorithm's execution time.

Following research questions are investigated:

1. How to select optimization parameters for a given point in problem space?
2. The convergence rate: How many iterations are needed to reach a given solution quality?
3. How does each parameter of the algorithm affect the convergence rate and quality of solutions?
4. How to speedup the convergence rate? That is, how to decrease the run-time of mapping algorithm?
5. How often does the algorithm converge to a given solution quality?

1.4 Summary of contributions

Task mapping in design space exploration often lacks systematic method that scales with respect to the application and hardware complexity, i.e. the number of task nodes and PEs. This Thesis presents a series of methods how to select parameters for a given MPSoC. Developed methods are part of the DCS task mapper tool that was used in Koski design flow [3] [38] [39]. Convergence rate is analyzed for our modified Simulated Annealing algorithm. Also, a comparison is made between automatically selected parameters and global optimum solutions. We are not aware of a global optimum comparison that presents such detailed comparison. Presented methods are

compared to other studies, that specifically use SA or other optimization method. Their advantages and disadvantages are analyzed.

Summary of contributions of this Thesis are:

- Review of existing work on task mapping
- Development of task mapping methods including an automatic parameter selection method for Simulated Annealing that scales up with respect to HW and SW complexity. This is called *Simulated Annealing with Automatic Temperature* (SA+AT).
- Optimal subset mapping (OSM) algorithm
- Comparisons and analyses of developed methods including:
  - Comparison of SA, OSM, Group Migration (GM) and Random Mapping (RM) and GA
  - Convergence analysis of SA+AT with respect to global optimum solution
  - Comparison of SA+AT to published mapping techniques
- Tool development: DCS task mapper tool that is used to explore mapping algorithms themselves efficiently

1.4.1 Author's contribution to published work

The Author is the main author, contributor in all the Publications and the author of DCS task mapper.

Tero Kangas helped developing ideas in [P1]-[P3].

Erno Salminen helped developing ideas in [P2]-[P7]. In particular, many important experiments were proposed by him.

Professor Timo D. Hämäläinen provided highly valued insights and comments for the research and helped to improve text in all the Publications.

Professor Marko Hännikäinen helped to improve text in [P3]-[P5].

1.5 Outline of Thesis

Thesis is outlined as follows. Chapter 2 introduces design space exploration and concepts relevant to task mapping. Chapter 3 presents related work for task mapping. Chapter 4 presents the research questions, research method and results of Publications of the Thesis. Chapter 5 presents the DCS task mapper tool that was used to implement experiments for the Thesis and publications. Chapter 6 presents new results on SA convergence. SA is also compared with GA. Chapter 7 presents recommendations for publishing about SA and using it for task mapping. Chapter 8 shows the relevance of the Thesis. Chapter 9 presents conclusions and analyzes impact of the Publications and Thesis.

2. DESIGN SPACE EXPLORATION

2.1 Overview

Design space exploration (DSE) concerns the problem of selecting a design from the problem space to reach desired goals in the objective space within a given time and budget. Finding an efficient MPSoC implementation is a DSE problem where a combination of HW and SW components are selected. HW/SW systems are moving to more complex designs in which heterogeneous processors are needed for low power, high performance, and high volume markets [86]. This complexity and heterogeneity complicates the DSE problem. Gries [30] presents a survey of DSE methods. Bacivarov [4] presents an overview of DSE methods for KPNs on MPSoCs.

The problem space is a set of solutions from which the designer can select the design. The selected design and its implementation manifests in measures of the objective space such as power, area, throughput, latency, form factor, reliability, etc. Hence, it is a multi-objective optimization problem. *Pareto optimal* solutions are sought from the problem space [27]. A solution is pareto optimal if improving any objective measures requires worsening some other. The best solution is selected from pareto optimal solutions. An automatic deterministic selection method for the best solution transforms the problem into a single objective problem. A manual selection requires designer experience.

The problem space is often too large to test all the alternatives in the given time or budget. For example, the task mapping problem has time complexity $O(M^N)$, where M is the number of PEs and N is the number of tasks. Selection from problem space may require compromises in objective space measures, e.g. power vs. throughput. Objective space measures should be balanced to maximize the success of the final product.

Graphics is often used to visualize alternative DSE methods and parameters. An

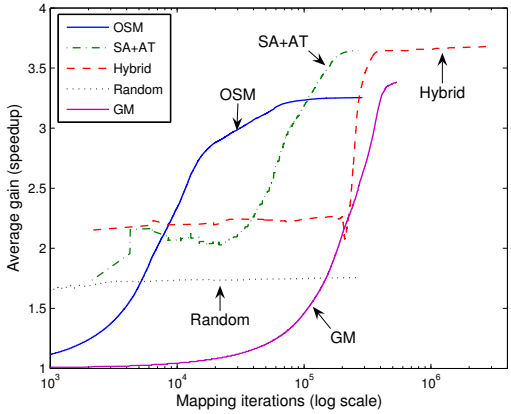


Fig. 5. *Evaluation of several mapping algorithms. Number of iterations and the resulting application speedup are compared for each algorithm.*

expert or heuristics selects a particular choice from the alternatives. For example, Figure 5 shows a comparison of mapping algorithms from [P4] where the number of mapping iterations is plotted against application speedup produced by each of the algorithms. The expert may choose between fast and slow exploration depending on whether coarse exploration or high performance is needed.

2.2 Design evaluation

There are at least four relevant factors for design evaluation of the problem space [30]. First, the time to implement the evaluation system is important. The evaluation system itself can be difficult to implement. This may require implementing simulators or emulators on various HW/SW design levels. This can be zero work by using an existing evaluation system, or years of work to create a new one.

Second, the time to evaluate a single design point with a given evaluation system places the limit on how many designs can be tested in the problem space. The method for evaluating a single design point in the problem space is the most effortful part of the DSE process. A single point can be evaluated with an actual implementation or a model that approximates an implementation. A good overview of computation and communication models is presented in [34]. Evaluation can be effortful which means only a small subset of all possible solutions are evaluated. This often means that incremental changes are made on an existing system; not creating a completely

unique system. Finding a new design point can be as fast as changing a single integer, such as changing a mapping of a single task. This could take a microsecond, where as simulating the resulting circuit can take days. Therefore, testing a single design point can be 10^{11} times the work of generating a new design point. Finding a new design point can also be an expensive operation, such as finding a solution for a hard constraint programming problem.

Third, the accuracy of the evaluation in objective space measures sets the confidence level for exploration. Usually there is a trade-off between evaluation speed and accuracy. Evaluation accuracy is often unknown for anything but the physical implementation which is the reference by definition.

Fourth, automating the exploration process is important, but it may not be possible or good enough. Implementation to a physical system is probably not possible without manual engineering, unless heavy penalties are taken. For example, automatic testing and verification of real-time properties and timing is not possible. Also, algorithms and heuristics often miss factors that are visible to an experienced designer. The grand challenge is to decrease manual engineering efforts as engineers are expensive and prone to errors.

2.3 Design flow

Traditional SoC design flow starts by writing requirements and specifications. The functionality of the specification is first prototyped in C or some other general purpose programming language. A software implementation allows faster development and testing of functional features than a hardware implementation. The prototype is analyzed with respect to qualities and parameters of the program to estimate speed, memory and various other characteristics of the system. For example, a parameterizable ISS could be implemented in C to test functionality and estimate characteristics of a processor microarchitecture being planned. When it is clear that correct functionality and parameters have been found, the system is implemented with hardware and software description languages. Parts of the original prototype may be used in the implementation.

However, transforming the reference SW code into a complete HW/SW system is far from trivial and delays in product development are very expensive. The biggest de-

lays happen when design team discover faults or infeasible choices late in the design. Automatic exploration reduces that risk by evaluating various choices early with less manual effort. Although human workers are creative and can explore the design in new ways, automated flow is many orders of magnitude faster in certain tasks. For example, trying out a large set of different task mappings, frequencies, and buffer sizes can be rather easily automated.

Automated SoC design tools does a series of transformations and analysis on the system being designed. This covers both application and HW model. Application and HW models must adhere to specific formalisms to support automated exploration. Therefore, it's necessary to raise the level of abstraction in design for the sake of analysis and automated transformations, because traditional software and hardware description languages are nearly impossible to analyze by automation. This has lead to estimation methods based on simpler models of computation (MoC). The models must capture characteristics of the system that are optimized and analyzed. Exploration tools are limited to what they are given possibilities to change and see. This can be very little or a lot. Minimally this means only a single bit of information. For the sake of optimization, it would be useful to have a choice of multitude of application and HW implementations.

2.3.1 Koski design flow

This Thesis presents DCS task mapper which is the mapping and scheduling part of the static exploration method in Koski framework [39]. The overview and details of the Koski DSE process is described in Kangas [38]. Koski framework uses a high-level UML model and C code for the application, and a high-level UML model and a library of HW components for the system architecture.

Application consists of finite state machines (FSMs) that are modeled with high-level UML and C code. The HW platform is specified with the UML architecture model and a library of HW components. The UML model, C code and HW components are synthesized into an FPGA system [38] where the application is benchmarked and profiled to determine its external behavior. The external behavior is defined by timings of sequential operations inside processes and communication between processes. An XSM model is then generated that contains the architecture, initial mapping and the profiled application KPN. The code generation, profiling and physical implementa-

tion is shown in Figure 6.

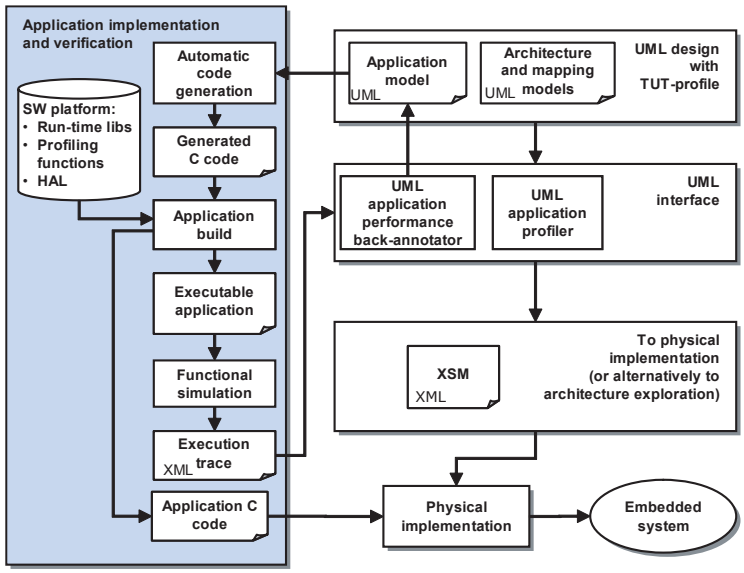


Fig. 6. *Koski design flow from code generation to physical implementation [38]*

Figure 7 shows an example video encoder modeled as a KPN from [65]. A number of slaves is run in parallel coordinated by a master process that orders slaves to encode parts of input picture. Each slave executes an identical KPN consisting of 21 computation tasks and 4 memory tasks. The system is simulated with Transaction Generator [40] which is a part of the dynamic exploration method. Data transfers between tasks that are mapped to separate PEs are forwarded to the communication network. Execution times of processes and the size of inter-process messages are determined by profiling the execution times on an instruction set simulator. Transaction Generator is used to explore mappings and HW parameters of the system.

The XSM model is explored with static and dynamic exploration methods that are used to optimize and estimate the system performance. The static and dynamic exploration methods are used as a two phase optimization method shown in Figure 8. Static exploration does coarse grain optimization, it tries a large number of design points in the problem space. The dynamic exploration method does fine-grain optimization.

The extracted KPN is passed to static architecture exploration method for performance estimation and analysis for potential parallelism for an MPSoC implementa-

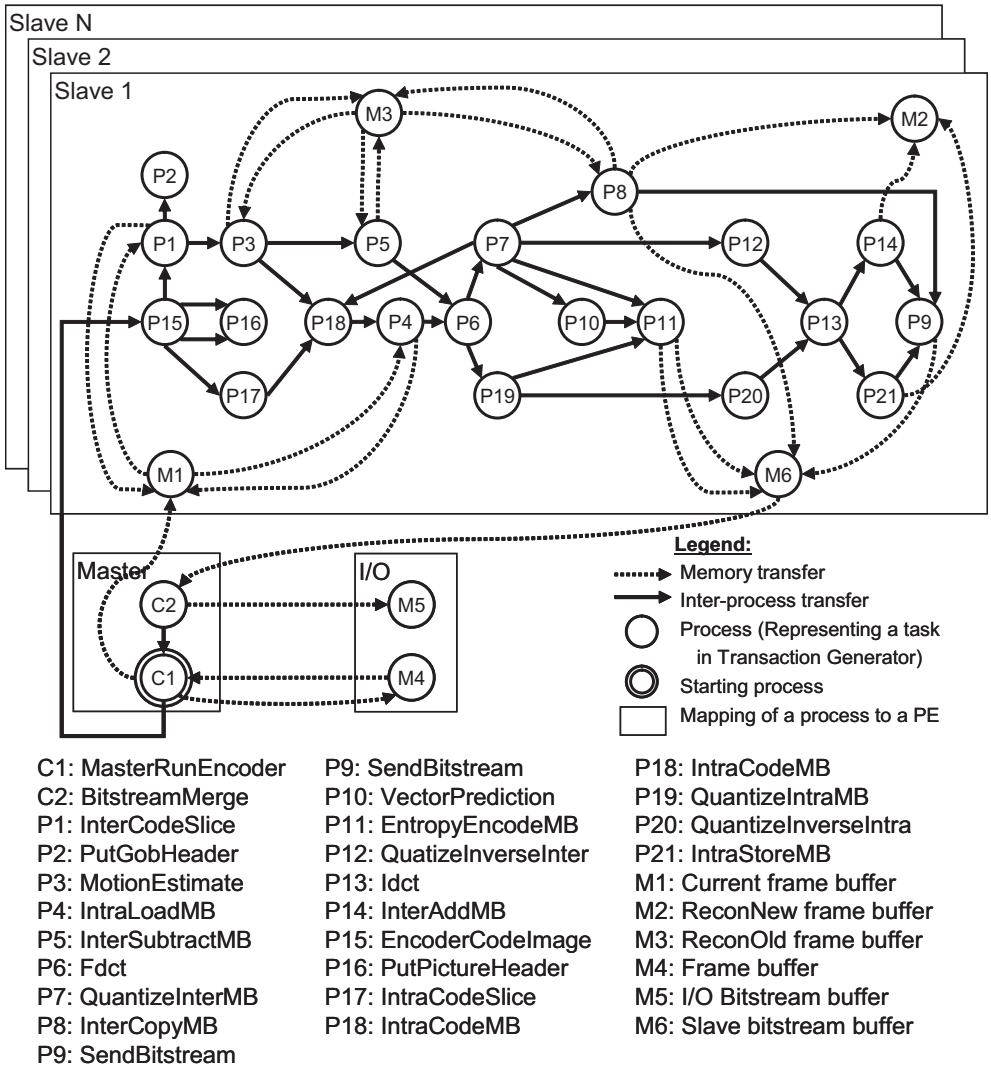


Fig. 7. *Video encoder modelled as a KPN [65]*

tion. Figure 9 shows the static exploration method as a diagram. The KPN, achitecture and initial mapping are embedded in the XSM model. The KPN model is composed of processes which consist of operations. Each process executes operations whose timings were profiled with an FPGA/HW implementation. Three operations are required for behavioral level simulation: read, compute and write operations. Read operation reads an incoming message from another process. Compute operations keeps a PE busy for a given number of cycles. Write operation writes a message of given size to a given process. Timing and data sizes of these operations capture the

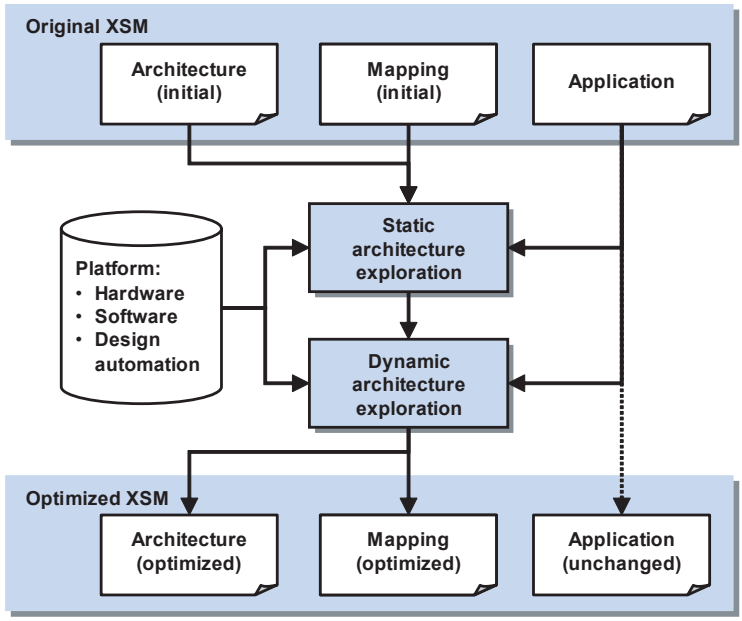


Fig. 8. *Two phase exploration method [38]*

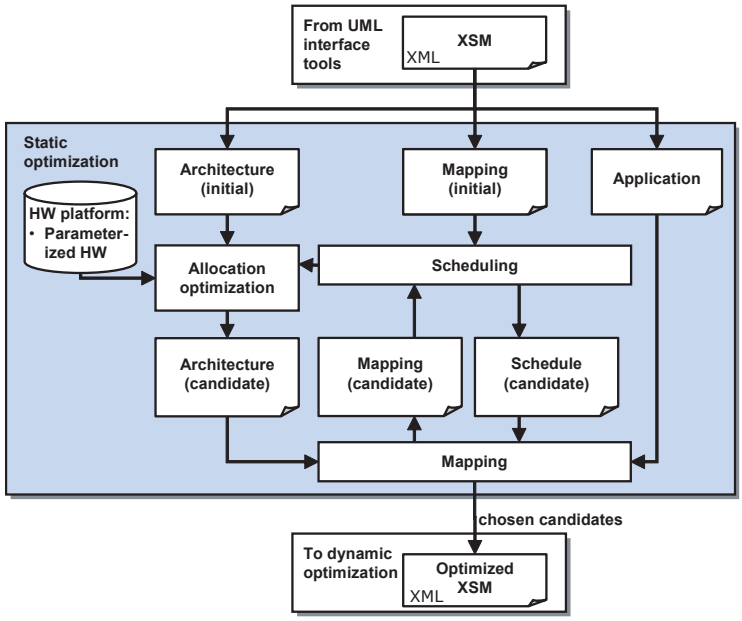


Fig. 9. *Static exploration method [38]*

parallel and distributed behavior of the application. This application model allows orders of magnitude faster exploration than a cycle-accurate ISA simulation or a circuit

model. For example, in our work evaluating a single design point in a 4 PE and 16 node KPN design took approximately 0.1ms as a behavioral model that schedules all computation and communication events on the time line. A circuit level model of the system could take days or weeks to simulate. Behavior level simulation is explained in Chapter 5.

The model is then further optimized with more accuracy in the dynamic architecture exploration phase. Optimization results from the static method are used as the initial point for the dynamic method. Specific parameters of HW system are optimized in this phase. The communication network (HIBI) and the KPN is modeled and simulated as a cycle accurate SystemC model. A transaction generator is used to generate events for the simulator based on the application embedded inside the XSM model. The mapping algorithms of DCS task mapper are also re-used in the dynamic exploration phase.

Finally, the solutions from exploration are analyzed, and the best solution is selected. This usually requires engineering experience. The best system is delivered for the back-end process to be implemented.

The inaccuracy of the static and dynamic exploration methods needs to be evaluated. A risk margin is a multiplier that is defined as the upper bound for the ratio of implemented and estimated cost of the system. A risk margin of 1 would be desirable, but implementation complexity raises the margin to a higher value. Parameters that make up the accuracy of estimation model are initially based on system specifications, designer experience and crude testing and benchmarking on existing HW/SW implementations. Therefore, the risk margin must be determined experimentally. Final estimates of the parameters can be determined by fine-tuning with some manual effort. Manual effort was to be avoided, but it allows exploration tool to be more efficient. Exploration tool is useful as long as it saves total engineering time.

Kangas [38] (p. 102) estimated the error of static and dynamic exploration methods for a video encoder application. The static method gave an estimation error of less than 15% and the dynamic method gave an error less than 10%. Timing profile was obtained with by profiling a synthesized FPGA system, and these timings were inserted into the application model.

3. RELATED WORK

This chapter investigates the use of Simulated Annealing in task mapping. We analyze the research question of how to select SA optimization parameters for a given point in DSE problem space. We present a survey of existing state-of-the-art in SA task mapping.

3.1 Simulated Annealing algorithm

SA is a widely used metaheuristic for complex optimization problems. It is a probabilistic non-greedy algorithm that explores the search space of a problem by annealing from a high to a low *temperature*. Temperature is a historic term originating from annealing in metallurgy where material is heated and cooled to increase the size of its crystals and reduce their effects. Temperature indicates the algorithm's willingness to accept moves to a worse state. Probabilistic behavior means that SA can find solutions of different goodness between 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.

Figure 10 shows the SA pseudocode. The algorithm takes initial temperature T_0 and initial state S as a parameters. **Cost** function evaluates the objective function to be optimized. **Temperature** function returns the annealing temperature as a function of T_0 and loop iteration number i. **Move** functions generates a new state from a given state. **Random** function returns a random real number in range $[0,1)$. **Accept** function returns *True* iff a state change with cost function difference $\Delta C > 0$ should be accepted. **End-Condition** function returns *True* iff optimization should be termi-

nated. Parameters of the end conditions are not shown in the pseudocode. These may include measures such as the number of consecutive rejected moves, current and a given final temperature, current and accepted final cost. Finally the algorithm returns the best state S_{best} in terms of the **Cost** function.

```
SIMULATED_ANNEALING(S, T_0)
 1  C ← COST(S)
 2  S_best ← S
 3  C_best ← C
 4  for i ← 0 to ∞
 5  do T ← TEMPERATURE(T_0, i)
 6     S_new ← MOVE(S, T)
 7     C_new ← COST(S_new)
 8     ΔC ← C_new − C
 9     if ΔC < 0 or RANDOM() < ACCEPT(ΔC, T)
10        then if C_new < C_best
11                then S_best ← S_new
12                     C_best ← C_new
13             S ← S_new
14             C ← C_new
15     if END-CONDITION()
16        then break
17  return S_best
```

Fig. 10. *Pseudocode of the Simulated Annealing (SA) algorithm*

3.2 SA in task mapping

Table 1 shows SA usage for 14 publications, each of which are summarized below. All publications apply SA for task mapping, five publications use SA for task scheduling, and one for communication routing. None uses SA simultaneously for all purposes. There are many more publications that use different methods for task mapping, but they are outside the scope of this Thesis. Synthetic task mapping means the paper uses task graphs that are not directed toward any particular application, but benchmarking the mapping algorithms.

Table 1. *Table of publications and problems where Simulated Annealing is applied. Task mapping with SA is applied in a publication when "Mapping sel." column is "Y". Task scheduling and communication routing with SA is indicated similarly with "Sched. sel." and "Comm. sel." columns, respectively. All* 14 *publications apply SA to task mapping in this survey, five to task scheduling, and one to communication routing.*

Paper	Mapping sel.	Sched. sel.	Com. sel.	Application field
Ali [2]	Y	N	N	QoS in multisensor shipboard computer
Bollinger [8]	Y	N	Y	Synthetic task mapping
Braun [9]	Y	N	N	Batch processing tasks
Coroyer [20]	Y	Y	N	Synthetic task mapping
Ercal [24]	Y	N	N	Synthetic task mapping
Ferrandi [25]	Y	Y	N	C applications partitioned with OpenMP: Crypto, FFT, Image decompression, audio codec
Kim [43]	Y	Y	N	Synthetic task mapping
Koch [45]	Y	Y	N	DSP algorithms
Lin [56]	Y	N	N	Synthetic task mapping
Nanda [58]	Y	N	N	Synthetic task mapping
Orsila [P7]	Y	N	N	Synthetic task mapping
Ravindran [61]	Y	Y	N	Network processing: Routing, NAT, QoS
Wild [85]	Y	N	N	Synthetic task mapping
Xu [88]	Y	N	N	Artificial intelligence: rule-based expert system
# of publications	14	5	1	

Ali [2] optimizes performance by mapping continuously executing applications for heterogeneous PEs and interconnects while preserving two quality of service constraints: maximum end-to-latency and minimum throughput. Mapping is optimized statically to increase QoS safety margin in a multisensor shipboard computer. Tasks are initially mapped by a fast greedy heuristics, after which SA further optimizes the placement. SA is compared with 9 other heuristics. SA and GA were the best heuristics in comparison. SA was slightly faster than GA, with 10% less running time.

Bollinger [8] optimizes performance by mapping a set of processes onto a multiprocessor system and assigning interprocessor communication to multiple communication links to avoid traffic conflicts. The purpose of the paper is to investigate task mapping in general.

Braun [9] optimizes performance by mapping independent (non-communicating) general purpose computing tasks onto distributed heterogeneous PEs. The goal is to execute a large set of tasks in a given time period. An example task given was analyzing data from a space probe, and send instructions back to the probe before communication black-out. SA is compared with 10 heuristics, including GA and TS. GA, SA and TS execution times were made approximately equal to compare effectiveness of these heuristics. GA mapped tasks were 20 to 50 percent faster compared to SA. Tabu mapped tasks were 50 percent slower to 5 percent faster compared to SA. SA was run with only one mapping per temperature level, but repeating the annealing process 8 times for two different temperature coefficients. One mapping per temperature level means insufficient number of mappings for good exploration. However, GA is better with the same number of mappings. GA gave the fastest solutions.

Coroyer [20] optimizes performance excluding communication costs by mapping and scheduling DAGs to homogeneous PEs. 7 SA heuristics are compared with 27 list scheduling heuristics. SA results were the best compared to other heuristics, but SA's running time was two to four orders of magnitude higher than other heuristics. The purpose of the paper is to investigate task mapping and scheduling in general.

Ercal [24] optimizes performance by mapping DAGs onto homogeneous PEs and a network of homogeneous communication links. Load balancing constraints are maintained by adding a penalty term into the objective function. Performance is estimated with a statistical measure that depends on task mapping, communication

profile of tasks and the distance of communicating tasks on the interconnect network. Simulation is not used. The model assumes communication is Programmed IO (PIO) rather than DMA. SA is compared with a proposed heuristics called Task Allocation by Recursive Mincut. SA's *best* result is better than the *mean* result in 4 out of 7 cases. Running time of SA is two orders of magnitude higher than the proposed heuristics.

Ferrandi [25] optimizes performance by mapping and scheduling a Hierarchical Task Graph (HTG) [26] onto a reconfigurable MPSoC of heterogeneous PEs. HTGs were generated from C programs parallelized with OpenMP [70]. SA is compared with Ant Colony Optimization (ACO) [23], TS and a FIFO scheduling heuristics combined with first available PE mapping. SA running time was 28% larger than ACO and 12% less than TS. FIFO scheduling heuristics happens during run-time. SA gives 11% worse results (performance of the solution) than ACO and comparable results with TS.

Kim [43] optimizes performance by mapping and scheduling independent tasks that can arrive at any time to heterogeneous PEs. Tasks have priorities and soft deadlines, both of which are used to define the performance metric for the system. Dynamic mapping is compared to static mapping where arrival times of tasks are known ahead in time. SA and GA were used as a static mapping heuristics. Several dynamic mapping heuristics were evaluated. Dynamic mapping run-time was not given, but they were very probably many orders of magnitude lower than static methods because dynamic methods are executed during the application run-time. Static heuristics gave noticeably better results than dynamic methods. SA gave 12.5% better performance than dynamic methods, and did slightly better than GA. SA run-time was only 4% of the GA run-time.

Koch [45] optimizes performance by mapping and scheduling DAGs presenting DSP algorithms to homogeneous PEs. SA is benchmarked against list scheduling heuristics. SA is found superior against other heuristics, such as *Dynamic Level Scheduling* (DLS) [78]. SA does better when the proportion of communication time increases over the computation time, and the number of PEs is low.

Lin [56] optimizes performance while satisfying real-time and memory constraints by mapping general purpose synthetic tasks to heterogeneous PEs. SA reaches a global optimum with 12 node graphs.

Nanda [58] optimizes performance by mapping synthetic random DAGs to homoge-

neous PEs on hierarchical buses. The performance measure that is optimized is an estimate of expected communication costs and loss of parallelism with respect to critical path (CP) on each PE. Schedule is obtained with list scheduling heuristics. Two SA methods are presented where the second one is faster in run-time but gives worse solutions. The two algorithms reach within 2.7% and 2.9% of the global optimum for 11 node graphs.

Ravindran [61] optimizes performance by mapping and scheduling DAGs with resource constraints on heterogeneous PEs. SA is compared with DLS and a decomposition based constraint programming approach (DA). DLS is the computationally most efficient approach, but it loses to SA and DA in solution quality. DLS runs in less than a second, while DA takes up to 300 seconds and SA takes up to 5 seconds. DA is an exact method based on constraint programming that wins SA in solution quality in most cases, but is found to be most viable for larger graphs where constraint programming fails due to complexity of the problem space.

Wild [85] optimizes performance by mapping and scheduling DAGs to heterogeneous PEs. PEs are processors and accelerators. SA is compared with TS, FAST [49] [50] and a proposed Reference Constructive Algorithm (ReCA). TS gives 6 to 13 percent, FAST 4 to 7 percent, and ReCA 1 to 6 percent better application execution time than SA. FAST is the fastest optimization algorithm. FAST is 3 times as fast than TS for 100 node graphs and 7 PEs, and 35 times as fast than SA. TS is 10 as fast as SA.

Xu [88] optimizes performance of a rule-based expert system by mapping dependent production rules (tasks) onto homogeneous PEs. A global optimum is solved for the estimate by using linear programming (LP) [57]. SA is compared with the global optimum. SA reaches within 2% of the global optimum in 1% of the optimization time compared to LP. The cost function is a linear estimate of the real cost. In this sense the global optimum is not real.

3.3 SA parameters

Table 2 shows parameter choices for the previous publications. Move and acceptance functions, and annealing schedule, and the number of iterations per temperature level was investigated, where N is the number of tasks and M is the number of PEs. “N/A” indicates the information is not available. “DNA” indicates that the value does not

Table 2. *Simulated Annealing move heuristics and acceptance functions. "Move func." indicates the move function used for SA. "Acc. func" indicates the acceptance probability function. "Ann sch." indicates the annealing schedule. "q" is the temperature scaling co-efficient for geometric annealing schedules. "T_0 ada." means adaptive initial temperature selection. "Stop ada." means adaptive stopping criteria for optimization. "L" is the number of iterations per temperature level, where N is the number of tasks and M is the number of PEs.*

Paper	Move func.	Acc. func.	Ann. Sch.	q	T_0 ada.	Stop ada.	L
Ali [2]	ST, SW1	E	G	0.99	N	N	1
Bollinger [8]	MBOL	N/A	B	DNA	Y	N/A	$N(N-1)/2$
Braun [9]	ST	IE	G	0.8, 0.9	Y	N	1
Coroyer [20]	ST, P1	E	G, F	0.95	Y	Y	$N(M-1)$
Ercal [24]	ST	E	G	0.95	Y	N	$5N(M-1)$
Ferrandi [25]	ST, P4	E	G	0.99	N	N	LFE
Kim [43]	ST, SW1, P3	E	G	0.99	N	N	1
Koch [45]	Koch	E	K	DNA	Y	Y	N
Lin [56]	MLI	E	G	0.8	Y	Y	LLI
Nanda [58]	ST, SW2	E	G	0.9	N	N	5000
Orsila [P7]	ST	NIE	G	0.95	Y	Y	$N(M-1)$
Ravindran [61]	H1	E	K	DNA	N	Y	N/A
Wild [85]	ST, EM	N/A	G	N/A	N/A	Y	N/A
Xu [88]	SW1	E	G	0.5	Y	Y	N
Most common	ST	E	G	0.95	Y	N	-

apply to a particular publication. Detailed explanation of parameters is presented in Sections 3.3.1, 3.3.2 and 3.3.3.

3.3.1 Move functions

Table 2 shows 11 move functions applied in the publications.

Single Task (*ST*) move takes one random task and moves it to a random PE. It is the most common move heuristics, 9 out of 14 publications use it. It is not known how many publications exclude the current PE from solutions so that always a different PE

is selected by randomization. Excluding the current PE is useful because evaluating the same mapping again on consecutive iterations is counterproductive.

EM [85] is a variation of ST that limits task randomization to nodes that have an effect on critical path length in a DAG. The move heuristics is evaluated with SA and TS. SA solutions are improved by 2 to 6 percent, TS solutions are not improved. Using EM multiplies the SA optimization time by 50 to 100 times, which indicates it is dubious by efficiency standards. However, using EM for TS approximately halves the optimization time!

Swap 1 (*SW1*) move is used in 3 publications. It chooses 2 random tasks and swaps their PE assignments. These tasks should preferably be mapped on different PEs. *MBOL* is a variant of *SW1* where task randomization is altered with respect to annealing temperature. At low temperatures tasks that are close in system architecture are considered for swapping. At high temperatures more distant tasks are considered.

Priority move 4 (*P4*) is a scheduling move that swaps the priorities of two random tasks. This can be viewed as swapping positions of two random tasks in a task permutation list that defines the relative priorities of tasks. *P1* is a variant of P4 that considers only tasks that are located on the same PE. A random PE is selected at first, and then priorities of two random tasks on that PE are swapped. *P3* scheduling move selects a random task, and moves it to a random position in a task permutation list.

Hybrid 1 (*H1*) is a combination of both task assignment and scheduling simultaneously. First ST move is applied, and then *P3* is applied to the same task to set a random position on a permutation list of the target PE. *Koch* [45] is a variant of H1 that preserves precedence constraints of the moved task in selecting the random position in the permutation of the target PE. That is, schedulable order is preserved in the permutation list.

MLI is a combination of ST and SW1 that tries three mapping alterations. First, tries ST greedily. The move heuristics terminates if the ST move improves the solution, otherwise the move is rejected. Then SW1 is tried with SA acceptance criterion. The move heuristics terminates if the acceptance criterion is satisfied. Otherwise, ST is tried again with SA acceptance criterion.

ST is the most favored mapping move. Heuristics based on swapping or moving priorities in the task priority list are the most favored scheduling moves. The choice of mapping and scheduling moves has not been studied thoroughly.

3.3.2 Acceptance functions

Table 2 shows acceptance functions for the publications. Acceptance function takes the change in cost ΔC and temperature T as parameters. There are 3 relevant cases to decide whether to accept or reject a move. The first case $\Delta C < 0$ is trivially accepted. The second case $\Delta C = 0$ is probabilistically often accepted. The probability is usually 0.5 or 1.0. The third case $\Delta C > 0$ is accepted with a probability that decreases when T decreases or ΔC grows.

Exponential acceptor function (*E*)

$$Accept(\Delta C, T) = exp(\frac{-\Delta C}{T}). \quad (1)$$

is the most common choice. Orsila [P7] uses the normalized inverse exponential function (*NIE*)

$$Accept(\Delta C, T) = \frac{1}{1 + exp(\frac{\Delta C}{0.5 C_0 T})}. \quad (2)$$

where T is in normalized range $(0, 1]$ and C_0 is the initial cost. Inverse exponential function is also known as the negative side of a *logistic function* $f(x) = \frac{1}{1+exp(-x)}$. Braun [9] uses an inverse exponential function (*IE*)

$$Accept(\Delta C, T) = \frac{1}{1 + exp(\frac{\Delta C}{T})}. \quad (3)$$

where T_0 is set to C_0. Temperature normalization is not used, but the same effect is achieved by setting the initial temperature properly.

Using an exponential acceptor with normalization and a proper initial temperature is the most common choice. This choice is supported by the experiment in Section 6.2.1.

3.3.3 Annealing schedule

Table 2 shows annealing schedules for the publications. The annealing schedule is a trade-off between solution quality and optimization time. The annealing schedule defines the temperature levels and the number of iterations for each temperature level. Optimization starts at the initial temperature T_0 and ends at final temperature T_f. These may not be constants, in which case initial temperature selection and/or stopping criteria are adaptive. Stopping criteria defines when optimization ends.

Geometric temperature scale (G) is the most common annealing schedule. Temperature T is multiplied by a factor $q \in (0,1)$ to get the temperature for the next level. That is, $T_{next} = qT$. Usually several iterations are spent on a given temperature level before switching to the next level. q is most often 0.95 or 0.99 in literature. Some works use $q = 0.9$ that is also used by Kirkpatrick [44] that presented SA. They used SA for partitioning circuits to two chips. This is analogous to mapping tasks onto two PEs.

Bollinger [8], denoted B, computes a ratio R of minimum cost to average cost on a temperature level, and then applies a geometric temperature multiplier $T_{next} = min(R, q_L)T$, where q_L is an experimentally chosen lower bound for the temperature multiplier. q_L varied in range $[0.9, 0.98]$.

Ravindran [61] and Koch [45], denoted K, use a dynamic q factor that depends on the statistical variance of cost at each temperature level. The temperature is calculated with

$$T_{next} = \frac{T}{1 + \frac{Tln(1+\delta)}{3\sigma_T}} \tag{4}$$

where δ is an experimental parameter used to control the temperature step size and σ_T is the standard deviation of cost function on temperature level T. Higher δ or lower σ_T means a more rapid decrease in temperature. Koch suggests δ in range $[0.25, 0.50]$.

Coroyer [20] experimented with a fractional temperature scale (F). Temperature T is calculated as $T_i = \frac{T_0}{i}$ where T_0 is the initial temperature and i is the iteration number starting from 1. Fractional temperature was found to be worse than geometric.

The normalized exponential acceptor

$$Accept(\Delta C, T) = exp(\frac{-\Delta C}{0.5 C_0 T}) \tag{5}$$

and the normalized inverse exponential acceptor (2) put T_0 to a range $(0,1]$ where initial acceptance probabilities are adjusted to the initial cost.

The choice of initial temperature T_0 can be chosen experimentally or methodically. [P2] [P6] [P7] [9] [8] [20] [24] [45] [56] [88] present methods to set T_0 based on given problem space characteristics and desired acceptance criteria for moves. [P2] [P6] [P7] determine T_0 from the task graph and the system architecture. T_0 can be set purely based on simulation so that T_0 is raised high enough so that average move

acceptance probability p is high enough for aggressive statistical sampling of the problem space [20] [45]. Often $p \sim 0.9$.

There are several common stopping criteria. Optimization ends when a given T_f has been reached [20] [45], a given number of consecutive rejections happen, a given number of consecutive moves has not improved the solution, a given number of consecutive solutions are identical [20], or a solution (cost) with given quality is reached [45]. These criteria can also be combined. Constants associated with these criteria are often decided experimentally, but adaptive solutions exist too. [P2] [P6] [P7] use a T_f that is computed from problem space.

The number of iterations per temperature level L is defined as an experimental constant or a function of the problem space. A function of the problem space is more widely applicable, but a constant can be more finely tuned to a specific problem. L is often dependent on N and M. Ercal [24] proposes L that is proportional to $N(M-1)$, the number of neighboring solutions in the mapping space. Publications in this Thesis use the same principle.

Lin [56] uses an adaptive number of iterations per temperature level (*LLI*). Their approach starts with $L_0 = N * (N+M)$ for T_0. The number of iterations for the next temperature level is calculated by $L_{next} = min(1.1L, N(N+M)^2)$. However, on each temperature level they compute an extra $X - L$ iterations iff $X > L$, where $X = \frac{1}{exp((C_{min}-C_{max})/T)}$ and C_{min} and C_{max} are the minimum and maximum costs on the temperature level T. They compute initial temperature as $T_0 = a + b$, where a is the maximum execution cost of any task at any PE, and b is the maximum communication cost between any two tasks. Then they compute T_0 average and standard deviation of cost at T_0 by sampling a number of moves. The temperature is doubled (once) if the minimum and maximum cost is not within two standard deviations from the average.

Ferrandi [25] has adaptive number of iterations per temperature level (*LFE*). Temperature level is switched on the first accepted on the temperature level. However, there can be arbitrarily many rejected moves.

4. PUBLICATIONS

This chapter presents the problem space, research questions and contributions for Publications [P1-P7].

4.1 Problem space for Publications

Publications for this Thesis focus on mapping N tasks to M PEs, where tasks are interdependent general purpose computational tasks which can be executed in a finite time, i.e. the tasks are guaranteed to terminate within a specific time.

Applications in [P1-P6] are directed acyclic task graphs (DAGs), which are a special case of KPNs. The graphs were obtained from Standard Task Graph (STG) collection [81]. Scheduling is done with the *longest path* heuristics, which defines task priority to be the longest path from a node to its exit nodes. The longest path is also called the critical path. Nodes on the critical path are scheduled first. Applications are KPNs in [P7]. FIFO scheduling is used for KPN tasks. Tasks get scheduled in the order they become ready. Cycle costs for operations and communication message sizes are embedded directly into DAG and KPN models.

PEs are general purpose processors. PEs are homogeneous in [P1-P4, P6-P7], and heterogeneous in [P5]. PEs are connected with one or more communication links that each have a given throughput, latency and arbitration latency.

Execution time is a part of the objective space in all publications. [P3] also considers memory consumption due to communications buffering. [P5] considers energy consumption with given silicon area constraints in the problem space.

Table 3. *Scope of publications based on optimization criteria and research questions*

	P1	P2	P3	P4	P5	P6	P7
Optimized variables							
Exec. time	×	×	×	×	×	×	×
Memory			×				
Area/energy					×		
Multi-objective					×		
Mapping algorithm							
Convergence rate analysis		×	×		×	×	×
Run time optimized		×		×		×	×
Global optimum comparison							×
Research questions analyzed							
1. Parameter selection in a DSE point	×	×	×	×		×	×
2. Number of mappings and convergence							×
3. Parameters and convergence		×	×	×		×	×
4. How to speedup convergence		×		×		×	×
5. Converge rate							×

4.2 Research questions

The contribution of the Thesis is systematic analysis of the five research questions presented in Section 1.3.

Table 3 shows the scope of publications based on optimization criteria and research questions.

First research question How to select optimization parameters for a given point in DSE problem space? Parameter selection should factor in complexity which is defined by a given design point in the problem space. E.g. more PEs or tasks should imply more iterations for the algorithm to explore a sufficient part of the implementation space. It may also be that successful optimization is not a result of sufficient number of iterations, but entirely a different strategy is needed for different points in the problem space. A single set of parameters should not be used for multiple points in the problem space that have different complexities. Optimization effort is then wasted for simpler problems, but too little effort is used for more complex problems.

To find results we analyzed the effect of parameter selection for varying problem complexity. The benefit of mapping iterations versus problem complexity is weighed. Reliability of the methods was tested with small graphs, and scalability with larger graphs.

Second research question How many mapping iterations are needed to reach a given solution quality? The problem is NP-complete [59], and the difficulty of problem varies with respect to application and HW complexity. The global optimum is usually not known, and therefore, is not known either how far a given solution is from the optimum. This makes termination criteria for optimization hard to decide. A very fast evaluation cycle is needed to find global optimum solutions for even a simple and very constrained HW/SW system.

We examined the mapping iterations versus solution quality by obtaining global optimum solutions for small graphs and simple HW systems by brute force search.

Third research question How does each parameter of the algorithm affect the convergence rate? Each parameter may have a different effect depending on the point in problem space. Parameters can be compared with each other by altering them in the same point in problem space.

To find results we analyzed optimization convergence rate based on the effect of initial and final temperature, the number of mapping iterations per temperature level, different SA acceptance functions and the zero transition probability of the acceptance function.

Fourth research question How to speedup convergence rate? That is, how to decrease the run-time of mapping algorithm? Lot of the optimization can be ineffective. Effect of past optimization can be undone by later stages of optimization by changing the same or related task mappings obtained before.

We found out a way to compute initial and final temperatures to speed up optimization. Also, Optimal Subset Mapping algorithm is presented that converges rapidly.

Fifth research question How often does the algorithm converge to a given solution quality? Many combinatorial optimization algorithm, such as SA, rely on probabilis-

tic chance. Some estimates are needed how many times a probabilistic algorithm needs to be run to obtain a solution of given quality.

To increase reliability of our results we applied the methods on both small and large graphs. Small graphs and small number of PEs was exploited to obtain a global optimum comparison. Global optimum results give us an absolute reference on the convergence rate of the algorithms. For example, a method may converge within 5% of the global optimum solution in 10% of the optimization runs.

4.3 Hybrid Algorithm for Mapping Static Task Graphs on Multiprocessor SoCs

[P1] proposes a Hybrid Algorithm (HA) to automatically distribute task graphs onto a multiprocessor system to speedup execution. Research question 1 is analyzed. The multiprocessor system had 2, 3 or 4 PEs connected with a shared bus, but no shared memory. Message passing was used to carry out computation in parallel.

The Hybrid Algorithm is a combination of three algorithms: Fast Pre-mapping (FP), GM and SA. FP is used initially to place tasks tasks rapidly on separate processors. FP automatically places children of a task graph node to separate PEs to rapidly start with a parallel solution. This is not intended to be a good solution. The idea is to exploit trivial parallelism in the beginning of optimization to save mapping iterations. SA and GM are used after FP. SA and GM are used for global search, and GM is used for local search. The algorithm start with a full annealing temperature range, and iteratively run SA by halving the initial temperature but keep the final temperature fixed. The SA would become finer grained on each iteration, because its initial temperature would be greedier on each iteration. GM is run after each SA iteration to optimize easy task placements with a greedy local search method. Iterating the process would repeatedly find better and better solutions. The best solution seen during the process would be returned.

HA is not efficient because the SA's final temperature is set too low so that greedy optimization has already happened before GM is used. Therefore, GM is not needed. GM could be useful if it were the case that SA stopped before it got greedy. That is, the final temperature of SA would have to be higher.

The only real practical approach in this algorithm was to use SA by running it itera-

tively, and return the best result seen from all iterations. This could have been done without initial temperature adjustments, although halving the initial temperature at each iteration could be a heuristics for determining how many times to run the SA before stopping. This algorithm was shown to be wasteful in terms of optimization iterations in [P4].

It was later discovered that the experiment failed because there was insufficient number of mapping iterations per SA optimization. There was too few iterations per temperature level, and the final temperature was too high. SA would not converge on a good solution for these reasons. Also, there was a possibility of premature termination before the final temperature. This showed us that manual parameter selection is error-prone, and the failure motivated us to devise methods for automatic parameter selection. SA+AT was created for this reason.

4.4 Parameterizing Simulated Annealing for Distributing Task Graphs on multiprocessor SoCs

[P2] proposes SA+AT method that automatically parameterizes SA. Research questions 1, 3 and 4 are analyzed. The presented method decreases optimization time and increases application speed. A trade-off is made between optimization time and optimized execution time. The goal is to find a reliable optimization method that tries to search broad regions in the problem space, but also focus on the attractive parts of the space with more effort. This requires a careful annealing schedule, and possibly several independent runs of the algorithm. The method saved 50% of optimization time in an experiment by only marginally slowing down the application.

The method takes the basic philosophy that the number of iterations and the temperature range is determined by the structure of the given application and hardware architecture. The more complex the application or hardware the more iterations are needed to converge to a good mapping reliably. Exponential time algorithms are directly rejected so that the method can be applied for large problems.

The method parameterizes SA by looking at the number of PEs and tasks, and task execution times on PEs. The method scales up with problem space complexity. The number of mappings per temperature level is set with respect to the number of PEs and tasks. The number of temperature levels is based on task execution times on PEs.

The PEs can be heterogeneous which affects the number of temperature levels.

The annealing schedule is defined by the number of iterations per temperature level L, initial and final temperatures, and the coefficient q that defines a geometric temperature schedule.

Transition probability is defined so that transition probabilities occur within an effective range, and the temperature range is normalized into $(0,1]$. The method needs a value for the k variable, which determines a safety margin for initial and final temperature. Using $k = 2$ has been found sufficient, if not overly paranoid, in our papers.

A method is proposed to determined initial and final temperatures based on task execution times relative to application execution time on fastest and slowest processor. Initial temperature selection is not crucial in the sense that overestimation leads to excessive optimization time, but does not make the final solution bad. The proposed method sets initial temperature high enough so that optimization effort is not wasted for slowly converging chaotic random moves. Final temperature T_f selection is crucial, however. Too high T_f means greedy optimization opportunities are lost so that possible local minima are not found. Too low T_f means optimization time is wasted in a local minimum that does not converge anywhere else.

L is chosen to be $N(M-1)$ and compared to different values with respect to application speedup. It is found that the given choice is appropriate for 300 task graphs with 2 to 8 processors.

The method is tested with 10 random graphs (DAGs) with 50 nodes and 10 random graph with 300 nodes. The number of PEs is 2 to 8. The method is able to predict an efficient annealing schedule for the problem. The number of optimization iterations is halved but optimized application's execution time is increased only marginally.

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

[P3] proposes a method for optimizing memory consumption and execution time of a task graph running on an MPSoC. Research questions 1 and 3 are analyzed. Parallelism is needed to speedup execution, but parallelism also increases memory consumption. The optimization algorithm uses an objective function that penalizes memory consumption and execution time.

Memory consumption is optimized for three kinds of buffers: First, temporary results that are stored on a PE waiting to be re-used on the same PE and then discarded. Second, temporary results that are stored on a PE waiting to be sent to another PE and then discarded. Third, input data from other PEs that is still being used by the PE.

SA, GM and RM algorithms are used to optimize two separate objective functions. The time-objective objective is used to optimize execution time. The memory-time objective is used to optimize both memory consumption and execution time.

The time-objective yields 2.12 speedup on average for SA, but also increases memory consumption by 49%. The memory-time objective yields 1.63 speedup without increase in memory consumption. The trade-off is 23% slower but 33% less memory.

The memory consumption trade-off is achieved by using the memory-time objective and adjusting the mapping by SA. Further memory savings would be possible by adjusting the schedule of the executed task graph to minimize the life-span of each intermediate result. But it is beyond the scope of the paper.

4.6 Optimal Subset Mapping And Convergence Evaluation of Mapping Algorithms for Distributing Task Graphs on Multiprocessor SoC

[P4] proposes a fast mapping method called *Optimal Subset Mapping* (OSM). Research questions 1, 3 and 4 are analyzed. OSM is a *divide and conquer* [21] method that assumes that different parts of the problem can be optimized separately that would make the solution converge towards a global optimum. This assumption is not valid for task mapping, but it can be a good strategy for other problems.

OSM takes a random subset of tasks and finds the optimum mapping in that subset by trying all possible mappings (brute force search). This process is repeated by decreasing and increasing the subset size within reasonable complexity limits. OSM is probabilistic because it chooses the subset randomly. It is also greedy because it only accepts an improvement to the best known solution at each round.

OSM is compared with GM, HA, SA, SA+AT and RM. OSM was the most efficient algorithm with respect to speedup divided by the number of iterations, but the absolute speedup was lower than with GM, HA, SA and SA+AT. OSM can be used as a

starting point for a large optimization task because it converges rapidly. SA+AT is the best algorithm overall. RM sets the baseline for all comparisons. Any algorithm should do better than random mapping when systematic optimization opportunities exist in the problem space. GM, OSM, SA and SA+AT do much better than RM.

The algorithms were evaluated with a mapping that has 300 node random task graphs and 2, 4 and 8 PEs. OSM converges very rapidly initially. Convergence characteristics of OSM and GM are quite the opposite to each other, while the SA+AT method [P2] is in the middle. OSM loses against GM, SA and SA+AT in application speedup. On possible strategy to use OSM would be to start with OSM and switch to SA+AT when OSM does not increase the speedup anymore.

4.7 *Evaluating of Heterogeneous Multiprocessor Architectures by Energy and Performance Optimization*

[P5] analyzes a trade-off between energy consumption and performance among heterogeneous MPSoCs. Pareto optimums are selected from a cloud of MPSoC candidates whose number of PEs and types of PEs varies with respect to performance, area and power.

Task graphs were distributed onto a set of architecture, each architecture consisting of heterogeneous PEs. 141 architectures were tested with 10 graphs that each have 300 nodes.

Energy consumption was estimated with static and dynamic energy parameters for 3 cases: without dynamic energy (constant power), with little dynamic power, and with more dynamic power. Well performing architectures were mostly the same in all cases. From low, medium and high speed PEs, the medium speed PEs were most power-efficient in the experiment.

Results from the experiment showed that SA method presented in [P2] shows good convergence properties for heterogeneous architectures too. The earlier paper used homogeneous PEs.

It was found that both energy-efficient and well performing solutions exist. However, most solutions were bad in all respects. The results indicate that good evaluation of heterogeneous architectures requires good mapping when the mapping problem com-

plexity grows. Therefore, automated exploration tools are needed. SA+AT method is presented and evaluated as a partial solution for this problem.

4.8 Best Practices for Simulated Annealing in Multiprocessor Task Distribution Problems

[P6] is a survey of SA methods for task mapping. Research questions 1, 3 and 4 are analyzed. SA parameterization is examined from existing publications. Common annealing schedules, acceptance functions and termination conditions are presented. 4 case studies are examined. Their relative properties, weaknesses and advantages are compared.

The survey presents analytical insight into selection of initial and final temperatures and the number of mappings L per temperature level. Optimization should be stopped at final temperature T_f so that optimization effort is not wasted for greedy optimization that is stuck in a local optimum. SA+AT method presented in [P2] is analyzed mathematically. It is shown that SA+AT coefficient k is inverse exponentially proportional to the probability p_f of accepting a worsening move on the final temperature level.

The survey also presents 9 best practices for using SA with task mapping problems. It is argued that more attention should be put on reflecting problem space dimensions in the SA parameters. A variety of SA parameters is recommended, including geometric temperature scaling coefficients, use of systematic methods for determination of initial and final temperatures, and a suitable number of iterations per temperature level. It is recommended that ST move heuristics should be used when in doubt, the same problem should be optimized multiple times to decrease the chance of bad random chains, a normalized temperature range should be used.

The survey presents an idea that SA could be used as a main program for an application that continuously optimizes itself provided that the application tolerates variance in parameters and solutions. For example, run-time speed of the application could be optimized on-the-fly to adapt to changing conditions. This discards real-time applications, but can be adapted for throughput seeking applications.

Open research questions in SA task mapping are also presented. The question of optimal zero transition probability ($\Delta C = 0$) is answered in the Thesis.

4.9 Parameterizing Simulated Annealing for Distributing Kahn Process Networks on Multiprocessor SoCs

[P7] extends and improves the SA+AT method presented in [P2]. Research questions 1 to 5 are analyzed. SA+AT is extended to support KPNs, which requires changes to parameter selection that determines an efficient annealing schedule. Optimization time is decreased with sensitivity analysis of tasks. Tasks with lowest execution times are ignored in the temperature range calculation to save optimization effort. Throwing the least time consuming set of processes, e.g. 5%, allows us to skip many temperature levels for the final temperature, which saves optimization time. Quality of results stayed approximately the same, but over half the optimization time was saved. The method can be applied to general process networks, it is not limited to KPNs.

Convergence properties of the SA+AT method were analyzed by extensive simulations using graphs with varying difficulty. SA+AT convergence is compared to global optimum solutions that are found with a brute force search for 16 node graphs with 2 and 3 PEs. Global optimum convergence rate varied from 0.2 percent to 35.8 percent. Numeric estimate is presented on the number of iterations to reach a global optimum by repeating SA+AT algorithm many times on the same problem. The number is based on the global optimum convergence probability. We are not aware of a global optimum convergence experiments that are this extensive.

4.9.1 Corrected and extended results

Unfortunately, the simulation results were later found to be inaccurate due to a bug in the simulator code. The bug was that two tasks mapped on the same PE would together communicate over the shared interconnect instead of doing faster memory operations on the local memory bus that is exclusively dedicated to that PE. The simulator bug changed the optimization process so that the interconnect became an unnecessary bottleneck for all comunication. This was a performance bug, and thus, the applications ran much slower than they could. We fixed the communication bug, and we present the corrected results. Also, we extend the global optimum convergence results for 4 PEs.

The experiment was divided into setups A and B. We explain the effect of simulator

bug in detail for setups A and B. The cost model for task communication is modelled as $f(n)+g(n)$, where $f(n)$ is the PE cost in time to handle communication layers for n bytes of data, and $g(n)$ is the respective interconnect cost. For setup A, $f(n)=0$. For setup B, $f(n)$ is in $O(n)$ for tasks communicating on different PEs and 0 for tasks communicating on the same PE. Thus, the cost model in setup B favors that two communicating tasks are located on the same PE. The cost model in setup A does not favor placing communicating tasks on the same PE, because the $f(n)$ is always 0. The mapping process in setup A would merely place tasks so that schedule would maximize interconnect utilization. Thus, the mapping process does not have the desired effect on performance. However, setup B convergence figures in the publication are indicative of a properly working system because of the PE cost model that favors placing communicating tasks on the same PE.

The effectiveness of the automatic temperature method was evaluated again with a fixed simulator. The test was the same as in the original publication, using 100 graphs for each setup. The number of nodes varies from 16 to 256.

Table 4 shows speedups for setup A. This is the corrected version of Table III in [P7]. Table shows the minimum, mean, median and maximum speedups for SA+AT and SA+ST for 2 to 4 PEs. SA+ST is the same as SA+AT, but it doesn't use the temperature range computation method presented.

Table 5 shows the number of mappings for setup A. This is the corrected version of Table IV in [P7]. SA+AT is less than 0.7 percent slower than SA+ST on average, but uses less than 38 percent of mapping iterations. The result tables for setup B are omitted; the numbers are similar. In short, SA+AT is at most 1.4 percent slower than SA+ST on average, but uses less than 37 percent of mapping iterations.

Tables 6, 7, 8 and 9 are the corrected versions of Tables VII to X in [P7]. t is the optimized application execution time, t_0 is the global optimum. Table 6 and 7 show the convergence rate to within p percent of global optimum for setups A and B, respectively. Results are also extended with 4 PE results. Table 8 and 7 show the expected the number of mappings it takes to repeat SA+AT to reach withing p percent of global optimum. Convergence results follow the same pattern with the fixed simulator as in the original publication.

Global optimum was found in 0.3 to 38.3 percent of optimization runs with SA+AT. It was found that near global optimum are frequently obtained. Results within 1%

Table 4. *Automatic and static temperature compared: SA+AT vs SA+ST speedup values with setup A. Higher value is better.*

	2 PEs SA+AT	2 PEs SA+ST	3 PEs SA+AT	3 PEs SA+ST	4 PEs SA+AT	4 PEs SA+ST
Min	1.781	1.817	2.296	2.347	2.612	2.585
Mean	**1.969**	**1.973**	**2.777**	**2.788**	**3.310**	**3.333**
Std	0.038	0.034	0.153	0.145	0.264	0.254
Med	1.985	1.987	2.821	2.836	3.362	3.390
Max	2.000	2.000	2.980	2.966	3.767	3.808

Table 5. *Automatic and static temperature compared: SA+AT vs SA+ST number of mappings with setup A. Lower value is better.*

	2 PEs		3 PEs		4 PEs	
	SA+AT	SA+ST	SA+AT	SA+ST	SA+AT	SA+ST
Min	473	2 900	1 190	5 800	2 170	8 690
Mean	6 570	17 990	13 400	36 080	20 590	54 250
Std	6 570	15 830	13 190	31 760	19 830	47 760
Med	3 620	11 590	7 450	23 240	11 780	34 890
Max	25 150	50 610	50 610	95 440	79 080	147 270

Table 6. *Brute force vs. SA+AT with setup A for* 16 *node graphs. Proportion of SA+AT runs that converged within p from global optimum. Higher convergence proportion value is better. Lower number of mappings is better.*

	Proportion of runs within limit p					
	2 PEs		3 PEs		4 PEs	
$p = \frac{t}{t_o} - 1$	acyclic	cyclic	acyclic	cyclic	acyclic	cyclic
+0%	0.030	0.045	0.003	0.007	0.001	0.005
+1%	0.323	0.124	0.006	0.019	0.004	0.012
+2%	0.617	0.302	0.017	0.033	0.011	0.028
+3%	0.819	0.529	0.040	0.061	0.021	0.050
+4%	0.951	0.717	0.094	0.103	0.047	0.084
+5%	0.990	0.853	0.177	0.158	0.089	0.135
+6%	0.999	0.939	0.288	0.235	0.140	0.206
+7%	1.000	0.977	0.447	0.330	0.210	0.285
+8%		0.991	0.603	0.438	0.311	0.374
+9%		0.999	0.745	0.555	0.427	0.459
+10%		1.000	0.851	0.665	0.542	0.554
+11%			0.927	0.753	0.665	0.641
+12%			0.969	0.830	0.770	0.719
+13%			0.988	0.890	0.862	0.791
+14%			0.995	0.931	0.924	0.850
+15%			0.999	0.960	0.963	0.897
+16%			1.000	0.978	0.985	0.930
+17%				0.990	0.994	0.956
+18%				0.995	0.998	0.973
+19%				0.998	0.999	0.985
+20%				0.999	1.000	0.992
+21%				1.000		0.996
+22%						0.998
+23%						1.000
mean mappings	765	800	1743	1754	2960	2943
median mappings	770	761	1737	1721	2943	2956

are obtained in 0.6 to 46.0 percent of runs. Results within 2% are obtained in 1.7 to 61.7 percent of runs. Results within 5% are obtained in 15.8 to 99.0 percent of runs. The results indicate that it is possible to save significant proportion of optimization iterations by sacrificing a few percent of solution quality from the global optimum.

Table 7. *Brute force vs. SA+AT with setup B for* 16 *node graphs. Proportion of SA+AT runs that converged within p from global optimum. Higher convergence proportion value is better. Lower number of mappings is better.*

	Proportion of runs within limit p					
	2 PEs		3 PEs		4 PEs	
$p = \frac{t}{t_o} - 1$	acyclic	cyclic	acyclic	cyclic	acyclic	cyclic
+0%	0.383	0.326	0.045	0.126	0.042	0.066
+1%	0.460	0.388	0.088	0.160	0.067	0.076
+2%	0.580	0.516	0.134	0.250	0.097	0.091
+3%	0.647	0.560	0.188	0.323	0.140	0.116
+4%	0.734	0.632	0.236	0.372	0.202	0.176
+5%	0.783	0.661	0.337	0.422	0.267	0.210
+6%	0.830	0.697	0.463	0.470	0.338	0.342
+7%	0.883	0.712	0.564	0.497	0.427	0.395
+8%	0.914	0.735	0.659	0.533	0.535	0.488
+9%	0.931	0.748	0.747	0.595	0.620	0.536
+10%	0.951	0.809	0.807	0.632	0.709	0.585
+11%	0.960	0.853	0.858	0.665	0.781	0.627
+12%	0.966	0.875	0.897	0.728	0.833	0.676
+13%	0.972	0.901	0.925	0.761	0.874	0.721
+14%	0.991	0.906	0.947	0.790	0.907	0.750
+15%	0.994	0.911	0.969	0.820	0.932	0.777
+16%	0.996	0.918	0.983	0.848	0.954	0.811
+17%	0.996	0.923	0.990	0.868	0.968	0.837
+18%	0.996	0.935	0.996	0.895	0.977	0.865
+19%	0.997	0.941	0.998	0.915	0.984	0.877
+20%	0.997	0.953	0.999	0.924	0.989	0.892
+21%	0.998	0.959	1.000	0.943	0.995	0.900
...	...	...		...	...	...
+24%	1.000	0.965		0.972	0.999	0.924
+25%		0.988		0.978	1.000	0.930
+26%		0.994		0.981		0.935
...		...		...		...
+30%		1.000		0.993		0.966
...				...		...
+37%				1.000		0.985
...						...
+48%						1.000
mean mappings	799	829	1649	1735	2661	2879
median mappings	784	788	1627	1623	2647	2704

Table 8. *Brute force vs. SA+AT with setup A for* 16 *node graphs: Approximate number of mappings to reach within p percent of global optimum. As a comparison to SA+AT, Brute force takes* 32770 *mappings for* 2 *PEs,* 14.3*M mappings for* 3 *PEs and* 1.1*G mappings for* 4 *PEs. Lower value is better.*

	Number of mappings					
	2 PEs		3 PEs		4 PEs	
	acyclic	cyclic	acyclic	cyclic	acyclic	cyclic
p	SA+AT	SA+AT	SA+AT	SA+AT	SA+AT	SA+AT
+0%	25 070	17 670	581 030	261 760	2 466 800	639 890
+1%	2 370	6 470	290 520	89 940	740 040	247 350
+2%	1 240	2 650	100 760	52 990	276 650	106 650
+3%	930	1 510	44 130	28 700	143 700	58 990
+4%	800	1 120	18 580	16 980	62 850	35 040
+5%	770	940	9 840	11 080	33 300	21 850
+6%	$\leq$ 770	850	6 060	7 460	21 130	14 260
+7%	...	820	3 900	5 310	14 100	10 320
+8%		810	2 890	4 000	9 510	7 870
+9%		800	2 340	3 160	6 940	6 410
+10%		$\leq$ 800	2 050	2 640	5 470	5 320
+11%		...	1 880	2 330	4 450	4 590
+12%			1 800	2 110	3 840	4 100
+13%			1 770	1 970	3 430	3 720
+14%			1 750	1 880	3 200	3 470
+15%			1 750	1 830	3 070	3 280
+16%			1 740	1 790	3 010	3 160
+17%			...	1 770	2 980	3 080
+18%				1 760	2 970	3 030
+19%				1 760	2 960	2 990
+20%				1 760	...	2 970
+21%				1 750		2 960
...				...		...
+25%						2 950

Table 9. *Brute force vs. SA+AT with setup B for* 16 *node graphs: Approximate number of mappings to reach within p percent of global optimum. As a comparison to SA+AT, Brute force takes* 32770 *mappings for* 2 *PEs,* 14.3*M mappings for* 3 *PEs and* 1.1*G mappings for* 4 *PEs. Lower value is better.*

	Number of mappings					
	2 PEs		3 PEs		4 PEs	
	acyclic	cyclic	acyclic	cyclic	acyclic	cyclic
p	SA+AT	SA+AT	SA+AT	SA+AT	SA+AT	SA+AT
+0%	2 090	2 540	33 930	13 790	62 910	43 620
+1%	1 740	2 140	18 630	10 840	39 600	37 730
+2%	1 380	1 610	12 290	6 950	27 580	31 570
+3%	1 240	1 480	8 750	5 370	19 060	24 860
+4%	1 090	1 310	6 990	4 660	13 190	16 350
+5%	1 020	1 250	4 900	4 110	9 990	13 740
+6%	960	1 190	3 570	3 690	7 890	8 820
+7%	910	1 160	2 920	3 490	6 240	7 290
+8%	880	1 130	2 500	3 250	4 970	5 900
+9%	860	1 110	2 210	2 920	4 290	5 370
+10%	840	1 030	2 040	2 740	3 750	4 930
+11%	830	970	1 920	2 610	3 410	4 590
+12%	830	950	1 840	2 380	3 190	4 260
+13%	820	920	1 780	2 280	3 040	3 990
+14%	810	920	1 740	2 200	2 940	3 840
+15%	800	910	1 700	2 120	2 850	3 710
+16%	$\leq$ 800	900	1 680	2 050	2 790	3 550
+17%	...	900	1 670	2 000	2 750	3 440
+18%		890	1 660	1 940	2 720	3 330
+19%		880	1 650	1 900	2 700	3 280
+20%		870	$\leq$ 1 650	1 880	2 690	3 230
+21%		870	...	1 840	2 680	3 200
+22%		860		1 830	2 670	3 160
+23%		860		1 800	2 670	3 130
+24%		860		1 780	2 660	3 120
+25%		840		1 780	...	...
+26%		830		1 770		3 080
+27%		...		1 760		3 040
...				...		...
+37%				1 740		2 920
...						...
+45%						2 880

5. DCS TASK MAPPER

5.1 DCS task mapper

DCS task mapper is a tool that does task mapping (placement) and scheduling for MPSoCs. Applications are modeled as DAGs or KPNs. Tasks are processes that execute on a PE, such as a processor or a HW accelerator. The tool tries to place each task to a PE so that a given objective function is minimized. The program supports SA, GA, RM, GM, OSM, brute force and FP mapping algorithms.

The program is available at DACI research group web site [22] under an Open Source license: LGPL 2.1 [55]. The program was written in C with performance in mind to carry out experiments for Publications [P2-P7]. [P1] used a Python prototype of the DCS task mapper. The tool, including the Python prototype, was written from scratch by the Author of the Thesis.

The program simulates a virtual system based on PEs and communication channels. Figure 11 shows the data flow and state transitions of the tool on a high level. Events, operations of tasks and communication messages are scheduled onto PEs and communication channels. Tasks compete for computation and communication time, but only one task at a time may execute on a PE or communicate on a communication channel. The program reads a mapping-scheduling problem as input. The input consists of a virtual system architecture, task graph, optimization algorithm and its parameters. The program maps task graph onto the system and schedules it to obtain objective space values, e.g. execution time, power and area. Execution time, power and area are annotated into architecture and task graph description. Time-level behavior is simulated according to annotated values. The system does not produce actual data.

Time behavior and resource contention is simulated for PEs and communication channels onto which task computation and communication is mapped. The simulator

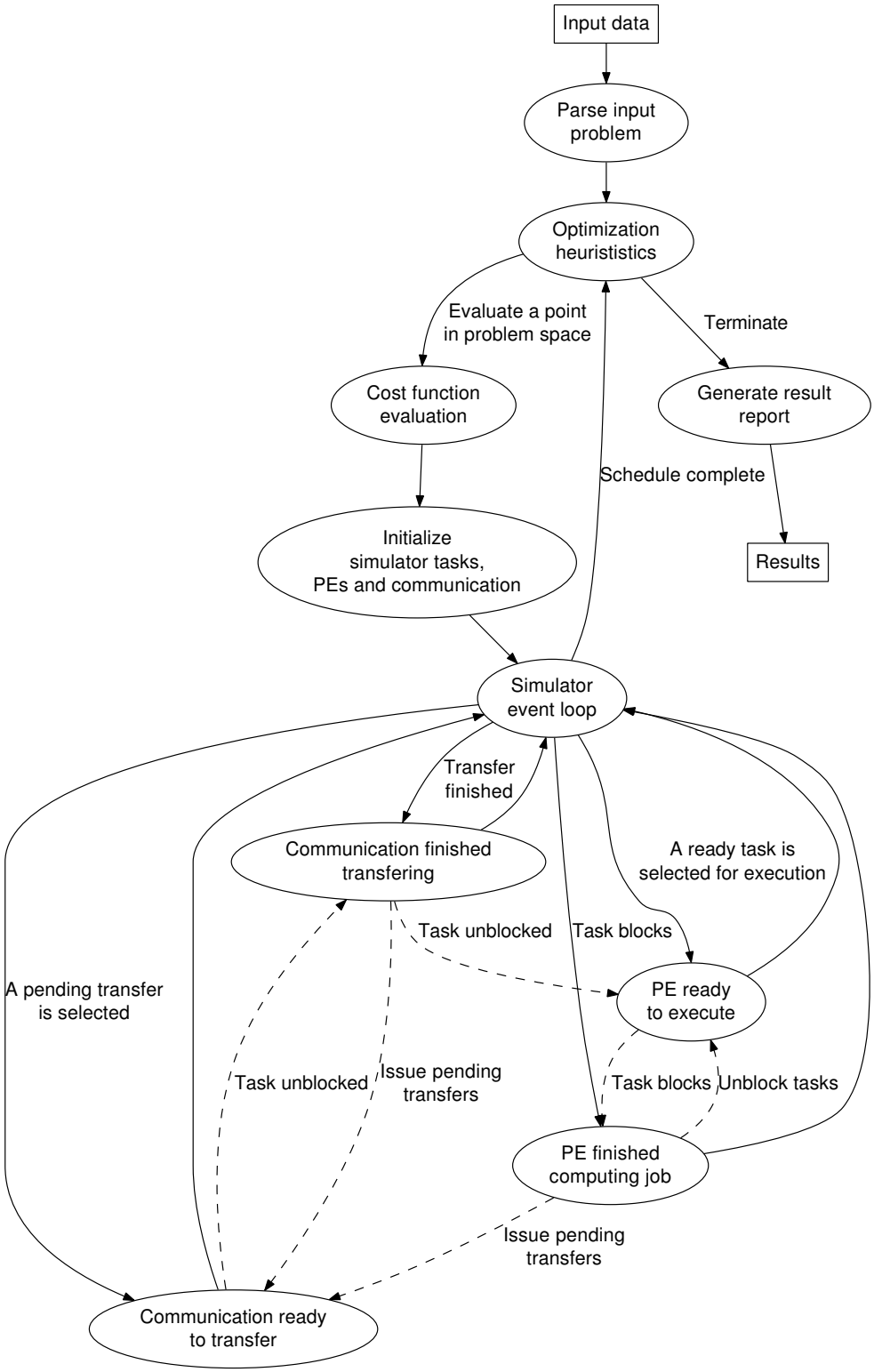


Fig. 11. *DCS task mapper data flow and state transitions. Solid line indicates data flow. Dashed line indicates causality between events in the state machine.*

event loop handles four events: transfer finished, computation finished, communication channel ready to transfer and PE ready to compute.

Tasks consist of 3 kinds of operations: compute X cycles, read Y or write Z bytes of data onto a communication channel. This simple operation set makes fast scheduling and simulation possible. PE executes operations from a task until a blocking read operation takes place, then a task switch takes place on the PE. When a task receives data that it needs to continue execution, it is unblocked. Tasks are waken up based on task priorities. Task priorities are supported for scheduling optimization. DAGs use critical path scheduling to set task priorities. KPNs use FIFO scheduling, but priority based scheduling is supported too. An efficient heap data structure is used to determine the task that has the most highest priority.

Communication scheduling can be optimized with different scheduling policies. FIFO, LIFO, priority and random order policies are supported. Critical path scheduling is used with DAGs. FIFO scheduling is used with KPNs. Adding new scheduling policies is modular.

The program contains approximately 8200 lines of C [62] written for GNU/Linux and UNIX-like systems. The program only depends on C library and a few system calls. The program was originally prototyped with a Python [63] implementation that was two orders of magnitude slower than the C implementation. Python was found to be an excellent prototyping language for algorithms and simulation, but it was too slow. Months of simulation time was reduced to days with a C implementation. The C implementation took several computation months to complete the experiments of the publications.

The execution time of the task mapper is effectively dependent on the simulation part, and thus, the number of mappings tried by the optimization heuristics. Optimization heuristics used consume less than one percent of the execution time. Table 10 shows the number of mappings explored, the mapping speed and the time computed with DCS task mapper for publications [P2] [P3] [P4] [P5] [P7].

5.2 *Job clustering with* jobqueue

Publications [P1]-[P4] use shell scripts to distribute task mapping jobs into a cluster of 10 Gentoo Linux machines with 2.8GHz Pentium 4 processor. A shell script would

Table 10. *Number of mappings explored, the mapping speed and the time computed with DCS task mapper for publications*

	P2	P3	P4	P5	P7
Mappings explored (1E9)	2.03		2.00	0.85	0.52
Mappings per second	620	2100-12000	297	358	90
Computation time (days)	38		78	27	67
PEs	2-8	2-4	2,4,8	2-5	2-4
Tasks	50, 300	50, 100	300	300	16-256

upload executable image and data files with OpenSSH [60] to machines in the cluster, and start the processing jobs on background with *GNU screen* [72]. This approach turned out to be tedious, since machines were often rebooted by users, and had to be restarted manually. Result data files were collected with SSH from each machine. Results were analyzed with custom Python scripts and standard shell tools.

The failing machine problem was mitigated by creating a new clustering tool called *jobqueue* [35]. jobqueue is spelled with non-capital initial letter, that is, the command line syntax form. The tool automatically re-issues jobs from failed machines to others. Publications [P5], [P7] and this Thesis use jobqueue to distribute tasks onto a cluster of machines, or a set of cores on a single machine.

jobqueue is command line tool that tracks a number of parallel jobs. jobqueue reads one line per computation job from a file (or standard input), and issues these jobs to a number of execution places. Execution place is usually a machine in a cluster, or a single core on the machine that executes jobqueue. Jobqueue supports job migration for failed machines. If a machine is rebooted, the job comes back to work queue and is later submitted to another machine. jobqueue terminates when all jobs have been processed, or all execution places have failed.

Each issued job is a line in a job file. The line is interpreted as a shell command with arguments. jobqueue appends the execution place id to be the last argument of the executed shell command. It is up to the shell command to upload data files and issue RPC. Typically ssh and rsync are used inside the script. Remote command issuing is usually as simple as the following shell script:

```
#!/bin/sh
input="$1"
results="$2"
host="$3"
cat "$input" |ssh "$host" "nice -n 19 bin/taskmapper" \
 > "$results"
```

`bin/taskmapper` is the DCS task mapper executable on the remote machine. The input is read from a data file that is piped through OpenSSH to the remote taskmapper. The remote taskmapper writes results to its standard output which is piped through OpenSSH to a local result file. Minimal engineering and resources are needed to setup this kind of system. No commercial software is needed.

Jobqueue has 2280 lines of C code, out of which 970 lines are dead code under prototype development. It was written by the Author of this Thesis. The source code is available under Public Domain at [35].

6. SIMULATED ANNEALING CONVERGENCE

6.1 On global optimum results

This section analyzes research questions 2, 3 and 5. How many mapping iterations are needed to reach a given solution quality? SA convergence rate and the number of mapping iterations is compared with respect to global optimum solutions. How does each parameter of the algorithm affect the convergence rate and quality of solutions? The effect of L value and acceptance function is analyzed with respect to the number of mapping iterations and solution quality. How often does the algorithm converge to a given solution quality? We present convergence results with reference to global optimum solutions.

A global optimum is hard to find for large task graphs since the mapping problem is in NP-complexity class. Effectiveness of heuristics may decrease rapidly as the number of nodes grows. The total number of mappings X for N nodes and M PEs is $X = N^M$. The number of mappings grows exponentially with respect to nodes, and polynomially with respect to PEs. Adding PEs is preferable to adding nodes from this perspective.

[P7] analyzed global optimum convergence for 2 and 3 PE architectures and 16 node KPNs by using a brute force algorithm. The experiment is extended to 32 node graphs on 2 PEs. More than 2 PEs would be too costly to compute in brute force for 32 or more nodes. PE mapping was fixed for one node, which yields $X = 2^{31} \sim 2.1 \times 10^9$ mappings for each graph. Fixing one node is a valid choice because PEs are homogeneous. The technical details of the experiment are explained in [P7].

Ten task graphs were generated with setup B in [P7]. Graphs were generated with kpn-generator [46] with following parameters: target distribution parameter 10%, cyclic graphs (default) and b-model parameter for both communication size and computation time as 0.7 (default). Target distribution defines the relative number of tasks

that can act as a target for each task. For example, 10% target distribution for a 32 node graph means each task can communicate with at most 3 other tasks.

Global optimum results were compared to SA+AT by running SA+AT 1000 times independently for each graph. The proportion of SA+AT that come within p percent of global optimum was calculated. Table 11 shows the proportion of SA+AT runs that got execution time $t \leq (1+p)t_o$, where p is the execution time overhead compared to global optimum execution time t_o. $p = 0\%$ means the global optimum (execution time). $p = 10\%$ means execution time no more than 10% over the global optimum. The last two rows show mean and median values for the number of mappings tried in a single SA+AT run. Figure 12 shows the same values. SA uses 0.4 to 9 millionths of the brute force iterations, but does not always converge to a global optimum.

Table 12 shows the expected number of mappings needed to obtain a result within p percent of global optimum by repeatedly running SA+AT. The same values are plotted in Figure 13. These values are based on the mean iterations and probabilities in Table 11. Note, convergence is probabilistic, which means the actual global optimum might never be reached. SA uses on average 43 to 64 millionths of the brute force iterations to reach global optimum. Optimization time is reduced by 5 orders of magnitude.

Brute force uses 2.0×10^4 times the iterations compared to SA with $L = 32$, the choice of SA+AT, and 2.3×10^4 times iterations compared to SA with $L = 64$. Both values show 4 orders of improvement over brute force. Results within 3% of global optimum are reached with 5 orders of magnitude improvement in iterations by using the SA+AT. Results within 16% are reached with 6 orders of magnitude improvement.

Figure 14 shows the task graph execution time for one graph plotted against the number of mappings. All possible mapping combinations for 32 nodes and 2 PEs were computed with brute force search. One node mapping was fixed. 31 node mappings were varied resulting into $2^{31} = 2147483648$ mappings. The initial execution time is 875 μs, mean 1033 μs and standard deviation 113 μs. Table 13 shows the proportion of SA runs that converged within p percent of global optimum and the associated number of mappings within that range computed from the brute force search. There is only one mapping that is the global optimum: 535 μs. SA reaches the global optimum in 2.1% of the runs. Repeating SA yields the global optimum in approximately $1E5$ iterations. RM would find global optimum in $1E9$ iterations in approximately

Table 11. *SA+AT convergence with respect to global optimum with* $L = 16, 32, 64, 128$ *and* 256 *for* 32 *nodes. Automatic parameter selection method in SA+AT chooses* $L =$ 32*. Values in table show proportion of SA+AT runs that converged within p from global optimum. Higher value is better.*

	Proportion of runs within limit p				
	L value				
$p = \frac{t}{t_o} - 1$	16	32	64	128	256
+0%	0.006	0.016	0.038	0.080	0.152
+1%	0.010	0.026	0.061	0.129	0.232
+2%	0.023	0.051	0.117	0.224	0.365
+3%	0.039	0.084	0.173	0.311	0.477
+4%	0.057	0.119	0.230	0.391	0.558
+5%	0.076	0.154	0.287	0.468	0.649
+6%	0.101	0.201	0.355	0.553	0.732
+7%	0.137	0.264	0.439	0.653	0.834
+8%	0.178	0.337	0.536	0.757	0.919
+9%	0.220	0.400	0.606	0.819	0.949
+10%	0.267	0.461	0.675	0.870	0.969
+11%	0.319	0.531	0.747	0.917	0.985
+12%	0.378	0.605	0.812	0.952	0.994
+13%	0.449	0.664	0.864	0.972	0.998
+14%	0.509	0.726	0.906	0.983	0.999
+15%	0.568	0.780	0.935	0.992	1.000
+16%	0.627	0.831	0.958	0.996	
+17%	0.681	0.868	0.974	0.998	
+18%	0.731	0.903	0.984	0.999	
+19%	0.778	0.930	0.990	0.999	
+20%	0.820	0.953	0.994	1.000	
+21%	0.853	0.968	0.997		
+22%	0.884	0.979	0.998		
+23%	0.908	0.985	0.999		
+24%	0.931	0.991	1.000		
+25%	0.947	0.995			
+26%	0.961	0.996			
+27%	0.971	0.998			
+28%	0.980	0.999			
+29%	0.987	1.000			
+30%	0.992				
...	...				
+37%	1.000				
mean mappings	840	1 704	3 516	7 588	19 353
median mappings	836	1 683	3 437	7 428	18 217

Table 12. *Approximate expected number of mappings for SA+AT to obtain global optimum within p percent for $L = 16, 32, 64, 128$ and 256 for 32 nodes. SA+AT chooses $L = 32$. Best values are in boldface on each row. Lower value is better.*

	Estimated number of mappings				
	L value				
p	16	32	64	128	256
+0%	137741	108507	**92766**	94966	127744
+1%	84022	65021	**57921**	58865	83347
+2%	36531	33208	**29947**	33919	53023
+3%	21767	20402	**20334**	24429	40556
+4%	14793	**14340**	15300	19406	34683
+5%	11099	**11084**	12242	16206	29820
+6%	**8311**	8471	9907	13721	26439
+7%	**6155**	6458	8001	11627	23208
+8%	**4723**	5060	6561	10027	21059
+9%	**3812**	4255	5804	9264	20402
+10%	**3150**	3695	5212	8724	19970
+11%	**2635**	3208	4708	8276	19654
+12%	**2221**	2817	4328	7972	19472
+13%	**1873**	2564	4069	7810	19394
+14%	**1651**	2348	3881	7717	19367
+15%	**1480**	2183	3761	7647	19357
+16%	**1340**	2051	3670	7618	
+17%	**1234**	1963	3609	7604	
+18%	**1149**	1886	3572	7595	
+19%	**1080**	1832	3550	7593	
+20%	**1025**	1787	3537	7589	
+21%	**985**	1760	3526		
+22%	**950**	1741	3524		
+23%	**926**	1730	3520		
+24%	**903**	1719	3518		
+25%	**888**	1713			
+26%	**874**	1710			
+27%	**865**	1706			
+28%	**857**	1705			
+29%	**851**	1704			
+30%	**847**				
+31%	**845**				
...	...				
+37%	**841**				

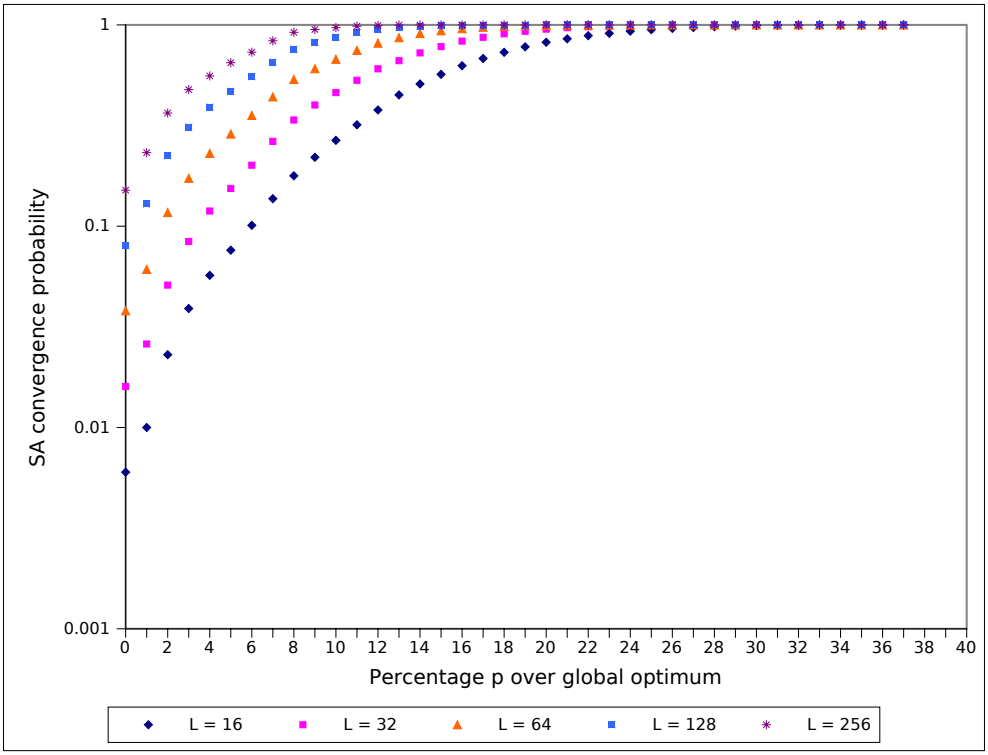


Fig. 12. *SA+AT convergence with respect to global optimum with* $L = 16, 32, 64, 128$ *and* 256 *for* 32 *nodes. SA+AT chooses* $L = 32$*. Lines show proportion of SA+AT runs that converged within p from global optimum for a given value of L. Lower p value on X-axis is better. Higher probability on Y-axis is better.*

half the runs. RM only converges within 13% of the global optimum in a million iterations. Also, there is only one mapping that is the worst solution: 1708 μs. Reversing the cost difference comparison in SA yields the worst solution in 0.9% of the runs. The SA algorithm can therefore also find the global maximum.

Table 14 shows global optimum convergence rate variance between graphs for L values. Doubling L approximately doubles the mean convergence rate. Minimum and maximum convergence rates vary by a factor of 50, approximately.

With $L = 256$ the hardest graph converged with probability 0.4%, but the same graph converged within 1% with 4.4% probability, within 2% with 8.8% probability, and within 3% with 22.3% probability. Sacrificing just 3% from optimum more than doubles the convergence rate on average, as shown by Table 11. The easiest graph converged to optimum with 27.8% probability, but within 3% with just 32.6% prob-

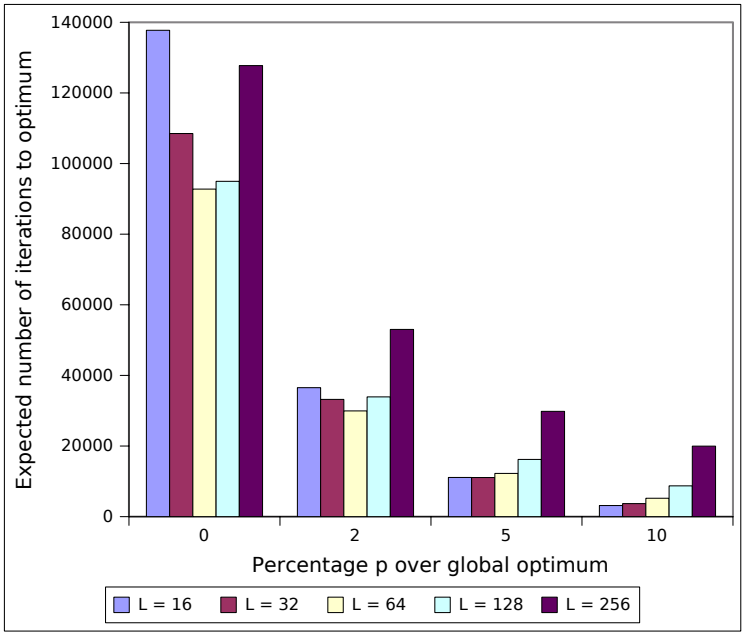


Fig. 13. *Expected number of iterations to reach global optimum within $p = 0\%, 2\%, 5\%$ and 10% for $L = 16, 32, 64, 128$ and 256 with 32 nodes. SA+AT chooses $L = 32$. Lower p value on X-axis is better. Lower number of iterations on Y-axis is better.*

Table 13. *Proportion of SA runs that converged within p percent of global optimum and the associated number of mappings within that range computed from the brute force search. Higher mapping number and SA run proportion is better.*

$p = \frac{t}{t_o} - 1$	Number of mappings	SA run proportion
+0%	1	2.1%
+1%	1	2.1%
+2%	4	5.8%
+3%	8	11.2%
+4%	16	17.1%
+5%	30	19.8%
+6%	45	21.2%
+7%	74	28.2%
+8%	127	34.7%
+9%	226	40.5%
+10%	355	49.0%

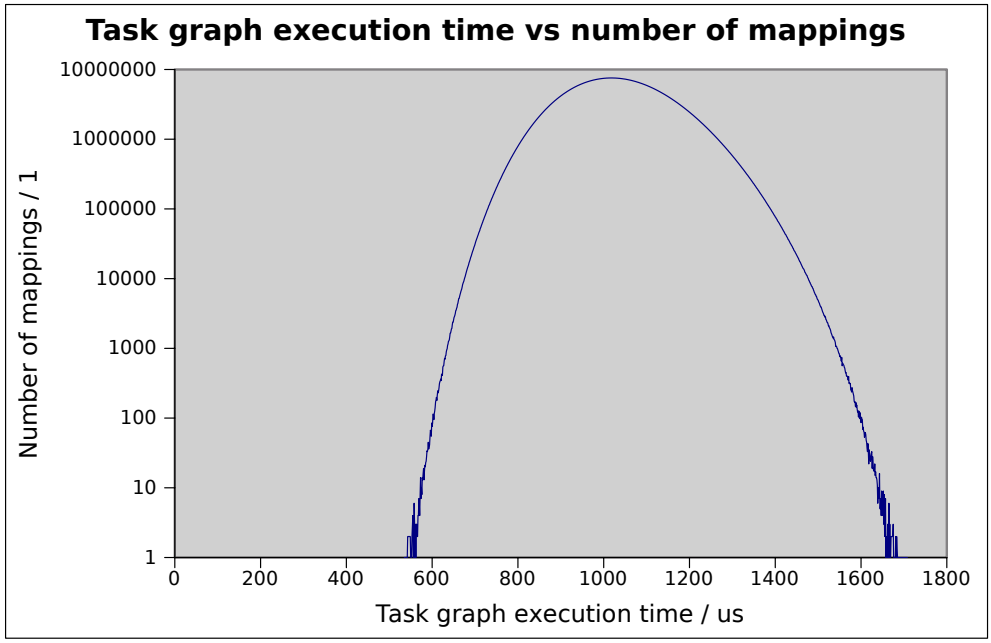


Fig. 14. *Task graph execution time for one graph plotted against the number of mappings. All possible mapping combinations for* 32 *nodes and* 2 *PEs were computed with brute force search. One node mapping was fixed.* 31 *node mappings were varied resulting into* $2^{31} = 2147483648$ *mappings. The initial execution time is* 875 *µs, mean* 1033 *µs and standard deviation* 113 *µs. There is only one mapping that reaches the global optimum* 535 *µs.*

ability. This indicates there are not many solutions near the global optimum, but hitting the global optimum is rather probable (27.8%). This is opposite to the hardest graph where hitting the global optimum is hard (probability 0.4%), but there are many solutions within 3% that hit at 22.3% probability.

Experiment data, scripts and instructions how to repeat and verify the experiment are available at DCS task mapper web page [22] under the section “Experimental data”. A reference C implementation of the SA+AT algorithm is also available at [64]. The experiment can be repeated with a GNU/Linux system by using DCS task mapper and jobqueue [35]. Experiment scripts probe for available processors and use multiprocessing to produce the results.

Table 14. *Global optimum convergence rate varies between graphs and L values. The sample is* 10 *graphs. Columns show the minimum, mean, median and maximum probability of convergence to global optimum, respectively. Value* 0% *in Min column means there was a graph for which global optimum was not found in* 1000 *SA+AT runs. For* $L \geq 32$ *global optimum was found for each graph. Note, the Mean column has same values as the* $p = 0\%$ *row in Table 11.*

L	Min (%)	Mean (%)	Median (%)	Max (%)
16	0.0	0.6	0.4	1.6
32	0.1	1.6	0.9	4.7
64	0.3	3.8	2.9	8.6
128	0.3	8.0	6.9	15.7
256	0.4	15.2	15.6	27.8

6.2 On SA acceptance probability

6.2.1 On acceptor functions

We analyze research question 3 in this section: How normalized exponential and inverse exponential acceptor functions compare in solution quality and convergence? We ran an experiment to compare the normalized inverse exponential (2) and normalized exponential (5) acceptor functions. Parameter selection method from [P7] was applied to (5). The experiment was re-run for both acceptor functions, 100 graphs, 2-4 PEs, 10 times each case, totaling 6000 optimization runs.

Table 15 shows gain results for both acceptor functions. Table 16 shows the number of iterations. Exponential acceptor is not more than half a percent better in terms of gain.

However, exponential acceptor loses up to 7% in the number of iterations. Both SA algorithms have an equal number of iterations per temperature from T_0 to T_f. Optimization terminates when $T \leq T_f$ and L consecutive rejections happen. Acceptance function affects the number of consecutive rejections which makes the difference in the number of iterations. Inverse exponential acceptor has higher rejection probability, which makes it stop earlier than an exponential acceptor.

Very few papers try different acceptor functions. [P1-7] use inverse exponential function. In retrospect it seems that exponential function would not have affected results

Table 15. *Comparing average and median gain values for the inverse exponential and exponential acceptors. A is the mean gain for experiments run with inverse exponential acceptor. B is the same for exponential acceptor. C and D are median gain values for inverse exponential acceptor and exponential acceptor, respectively. Higher value is better in columns A, B, C and D.*

	Mean gain			Median gain		
PEs	A	B	$\frac{A}{B}-1$	C	D	$\frac{C}{D}-1$
2	1.329	1.334	-0.3%	1.314	1.316	-0.2%
3	1.566	1.576	-0.6%	1.531	1.538	-0.5%
4	1.794	1.798	-0.2%	1.757	1.761	-0.2%

Table 16. *Comparing average and median iterations for the inverse exponential and exponential acceptors. E is the mean iterations for experiments run with inverse exponential acceptor. F is the same for exponential acceptor. G and H are the median iterations for inverse exponential and exponential acceptors, respectively. Lower value is better in columns E, F, G and H.*

	Mean iterations			Median iterations		
PEs	E	F	$\frac{E}{F}-1$	G	H	$\frac{G}{H}-1$
2	6758	6882	-1.8%	4228	4272	-1.0%
3	13577	14120	-3.8%	8456	8536	-1.0%
4	20734	22398	-7.4%	12850	13806	-6.9%

significantly. It is possible that a better acceptor function exists, but these two are used most widely.

Furthermore, we compared brute force results with 32 nodes and 2 PEs between normalized exponential and normalized inverse exponential acceptors. Table 17 shows the difference between global optimum converge rates for the two acceptors for SA+AT. Results indicate normalized exponential acceptor converges slightly more frequently than normalized inverse exponential acceptor. The SA+AT choice $L = 32$ displays difference of at most 2.4% in absolute convergence rate.

We recommend using normalized exponential acceptor for task mapping.

Table 17. *The table shows difference between SA+AT convergence rate of normalized exponential acceptor and normalized inverse exponential acceptor results. Inverse exponential acceptor results are shown in Table 11. The experiment is identical to that in Table 11. Automatic parameter selection method in SA+AT chooses $L = 32$. p percentage shows the convergence within global optimum. A positive value indicates normalized exponential acceptor is better.*

Exec. time overhead	Convergence rate difference				
	L value				
$p = \frac{t}{t_o} - 1$	16	32	64	128	256
+0%	0.001	0.003	0.006	0.007	0.009
+1%	0.003	0.006	0.008	0.003	0.011
+2%	0.001	0.009	0.014	0.009	0.006
+3%	0.003	0.010	0.016	0.015	0.015
+4%	0.004	0.011	0.016	0.012	0.015
+5%	0.006	0.014	0.021	0.011	0.010
+6%	0.011	0.014	0.025	0.012	0.013
+7%	0.011	0.016	0.028	0.010	0.003
+8%	0.011	0.015	0.030	0.006	−0.004
+9%	0.014	0.016	0.031	0.004	−0.004
+10%	0.016	0.019	0.035	0.002	−0.002
+11%	0.015	0.019	0.023	0.000	−0.002
+12%	0.013	0.014	0.015	−0.004	−0.002
+13%	0.003	0.024	0.012	−0.002	−0.001
+14%	0.002	0.020	0.007	0.001	0.000
+15%	0.001	0.019	0.006	0.000	
+16%	−0.001	0.015	0.004	0.000	
+17%	0.002	0.013	0.002	0.000	
+18%	0.001	0.009	0.002	0.000	
+19%	−0.002	0.006	0.002	0.001	
+20%	−0.004	0.001	0.002	0.000	
+21%	−0.004	−0.001	0.001		
+22%	−0.005	0.000	0.000		
+23%	−0.002	0.000	0.000		
+24%	−0.002	0.000	−0.001		
+25%	−0.002	0.000	0.000		
+26%	−0.002	0.001			
+27%	−0.001	0.000			
+28%	0.000				
+29%	−0.001				
...	...				
+35%	0.000				

6.2.2 On zero transition probability

SA acceptance function defines the probability to accept a move for a given objective change and the temperature. Zero transition probability is the probability of accepting a move that does not change the cost. We studied the effect of zero transition probability to the solution quality. This belongs to the category of the third research question.

The normalized inverse exponential acceptance function (2) gives 0.5 probability for zero transition, that is, when $\Delta C = 0$. The acceptance function was modified (6) to test the effect of zero transition probability:

$$Accept(\Delta C, T) = \frac{2a}{1 + exp(\frac{\Delta C}{0.5 C_0 T})}, \quad (6)$$

a is the probability for $\Delta C = 0$. SA+AT was re-run for several graphs with the setup specific in detail in [P7] with distinct probabilities $a \in [0.1, 0.2, \ldots, 1.0]$. Two 16 node cyclic graphs, one 128 node cyclic graph, two 16 node acyclic graphs, and one 128 acyclic graph, all with target distribution 10%, were tested with a 3 PE system. No causality was found between solution quality and the zero transition probability.

6.3 On Comparing SA+AT to GA

We present a comparison of SA+AT and a GA heuristics with respect to global optimum results. GA is used to optimize the same 10 graphs that were optimized with SA+AT in Section 6.1. Graphs have 32 nodes and the architecture has 2 PEs. The converge rate and the number of iterations to reach a global optimum solution is presented. This sets a direct comparison with SA+AT results. First, we present the GA heuristics in Section 6.3.1. Second, we analyze the problem of finding free parameters of the heuristics in Section 6.3.2. Finally, we describe the experiment and its results in Section 6.3.3.

6.3.1 GA heuristics

GA heuristics shown in Figure 15 is used to optimize a population of mapping solutions. GA heuristics is implemented in DCS task mapper. Line 1 creates a population

```
GENETIC_ALGORITHM(S_0)
 1  P ← CREATE_POPULATION_SET(S_0)
 2  i ← 0
 3  S_best ← COMPUTE_FITNESS_FOR_EACH_MEMBER(P ∪ S_0)
 4  while i < i_max
 5  do P_new ← SELECT_ELITE_FROM_POPULATION(P)
 6     while |P_new| < |P|
 7     do S_a ← SELECT_RANDOM_MEMBER(P)
 8        S_b ← SELECT_RANDOM_MEMBER(P)
 9        P_new ← P_new ∪ REPRODUCE(S_a, S_b)
10     P ← P_new
11     S_best ← COMPUTE_FITNESS_FOR_EACH_MEMBER(P ∪ S_best)
12  return S_best
```

Fig. 15. *Pseudocode of a Genetic Algorithm (GA) heuristics*

P of members based on an initial task mapping S_0. Each member is a mapping of a task graph. The first member of the population is S_0. Following members are generated by applying *point mutation* with 100% probability for each member. Point mutation applies on the first task for the first mutated member, on the second task for the second mutated member, and so on. This approach initializes the population so that most members are initially mutated with respect to one task mapping. Line 2 initializes the number of evaluated mappings i to 0. Lines 3 and 11 compute a *fitness* value for each member of the population. The most fittest member is returned as S_{best}. Variable i is incremented by the number of members whose fitness values are computed, as a side effect. Fitness value is used to determine a reproduction probability for a member. Fitness value F is computed as $F = \frac{1}{C}$ where C is the objective value (cost) of a member. Reproduction probability for the member is determined by probability

$$p_a = \frac{F_a}{\sum_{b \in P} F_b} \tag{7}$$

where p_a is the reproduction probability of population member a, F_a is the fitness value of member a and b is a member of population P. p_a is the share of the member a's fitness in the population. Line 4 loops the optimization algorithm until i_{max} mappings have been evaluated in total. The execution of the algorithm is very accurately

determined by the scheduling and simulation of a mapping. The experiment varies i_{max} to compare SA and GA efficiency with respect to number of mappings. Line 5 initializes a new population P_{new} that consists of the elite of population P. The elite is a given number of most fittest members in the population, as determined by the fitness value F. This is called *elitism*. Line 6 starts a loop that generates new members into the population by reproducing randomly selected members from population P. The loop continues until population P_{new} has as many members as population P. Line 7 and 8 choose a member from population by random. These members will reproduce a child. Member a of the population is selected with probability p_a that is determined by member's fitness value. The random selection also disqualifies a given number of least fittest members of the population from random selection. This is called *discrimination*. Line 9 combines the two selected members with a **Reproduce** function into a new member that is inserted into the new population. Line 10 copies population P_{new} to P. Line 12 returns the most fittest member, that is, the lowest cost member of the population after i_{max} mapping evaluations.

Reproduce function has two steps: crossover and mutation. Both steps take place with a given probability. These are called *crossover probability* and *chromosome mutation probability*. DCS task mapper has an optimization for equivalent functionality of line 11 which avoids recomputing new member's fitness if neither crossover nor mutation has affected the member.

Crossover If crossover takes place, a new member is constructed with *uniform* randomization. Uniform randomization selects a task mapping for each task by randomly assigning the mapping from either parent.

Chromosome Mutation If mutation takes place, mapping of each task is determined with *point mutation*. Point mutation takes an individual task, and alters its mapping with a probability determined by a given *gene mutation probability*. The mapping is altered by selecting a random PE. The alteration function is the **Move** function from SA+AT.

6.3.2 Free parameters of GA

Crossover, gene and chromosome mutation probabilities, the amount of elitism and discrimination, the population size and the total number of evaluated mappings i_{max} are the free parameters for the experiment. Crossover and chromosome mutation probability was randomly selected from range $[0.5, 1.0)$. Gene mutation probability was selected from range $[0.0, 0.1)$.

Population size was randomized from 4 to 256 members. Elitism was at most 20% of population size. Discrimination was at most 1 member. Discrimination was limited to 1 as it was found in earlier experiments that it is not helpful with values over 1 with our problem cases.

The experiment was repeated for three values of $i_{max} = 2048$, 4096 and 8192. For each i_{max} value, other free parameters were determined with a parameter exploration process that randomized values from the given ranges. The parameter exploration process evaluated 7641 random parameters for $i_{max} = 2048$ and 4096, and 9372 random parameters for $i_{max} = 8192$. Each parameter presents a selection of all free variables.

A cost value was computed for each parameter by evaluating each of the 10 graphs 10 times. That is, GA was run 100 times for each parameter. For each GA run, the optimized objective value was normalized with a global optimum value obtained in the experiment in Section 6.1.

For each graph, a *graph mean* value was computed from the 10 normalized objective values (each ≥ 1). The graph mean value is at least 1. The cost value of the parameter is the mean of graph means, that is, at least 1. A lower cost means a parameter is better. The parameter exploration process totalled $(2 \times 7641 + 9372) \times 100 = 2465400$ GA runs. This yielded approximately 1.2×10^{10} evaluated mappings with a cluster of machines and simulations parallelized with jobqueue.

Figure 16 shows the distribution of cost values for GA parameters in the parameter exploration process. 1.00 would mean the algorithm converges to global optimum on average. 1.10 means that algorithm converges within 10% of the global optimum. The minimum and maximum costs for different i_{max} series in the Figure are 1.124 and 1.532 for $i_{max} = 2048$, 1.093 and 1.515 for $i_{max} = 4096$, and 1.077 and 1.509 for $i_{max} = 8192$. The maximum cost is 36% to 40% higher than the minimum cost

obtained. This distribution means the parameter exploration process is necessary.

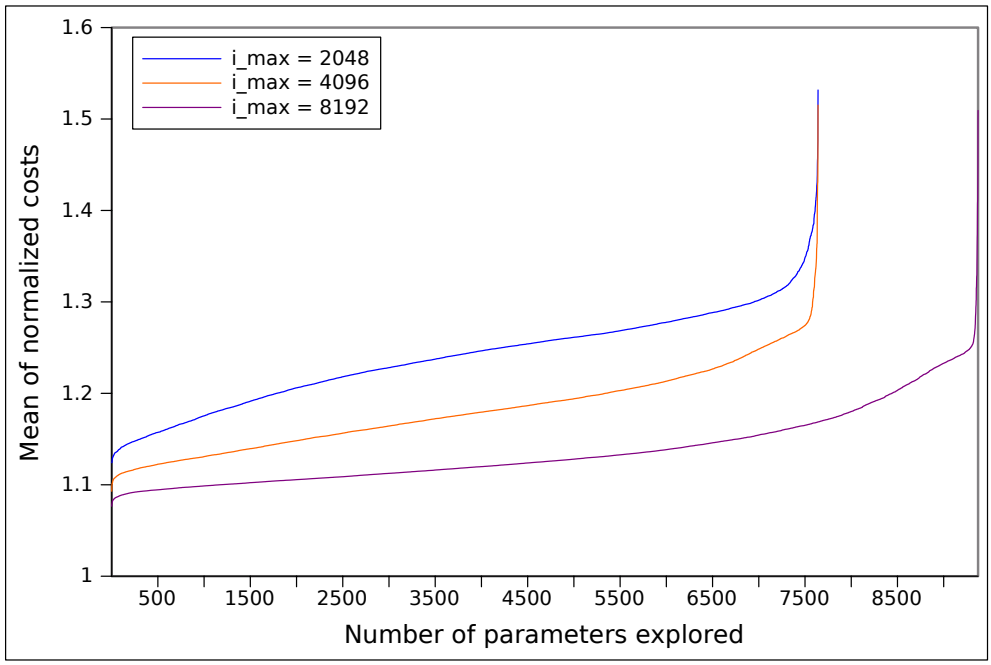


Fig. 16. *Distribution of parameter costs in the exploration process. Y-axis is the mean of normalized costs,* 1.00 *being the global optimum. Value* 1.1 *would indicate that a parameter was found that converges on average within* 10% *of the global optimum. Lower value is better.*

6.3.3 GA experiment

3 parameters for each value of i_{max} was selected from the parameter exploration process for detailed evaluation, that is the experiment. 3 paramaters for 3 values of i_{max} totals 9 parameters. GA is run 1000 times for each graph and each parameter. With ten graphs, this means $1000 \times 10 \times 9 = 90000$ GA runs. In total, this yields evaluating approximately 430 million mappings with GA. We present the results of the best of three parameters for each i_{max}. The best found parameters are shown in Table 18. It is not surprising that low population size was preferred because finding a good mapping by mutation requires a large number of sequential generations even with a good crossover algorithm.

Table 19 shows the global optimum convergence rate with p percent for SA+AT and

Table 18. *The best found GA parameters for different max iteration counts i_{max}. Discrimination and elitism presented as the number of members in a population.*

		i_{max}		
Parameter	Range	2048	4096	8192
Chromosome mutation probability	$[0.5, 1.0)$	0.995	0.906	0.691
Crossover probability	$[0.5, 1.0)$	0.500	0.678	0.861
Gene mutation probability	$[0, 0.1)$	0.068	0.043	0.088
Population size ($\|P\|$)	$[4, 256]$	4	5	7
Discrimination	$[0, 1]$	1	1	1
Elitism	$[0, 0.2\|P\|]$	1	1	1

GA. SA+AT results are copied directly from the earlier experiment. Columns in the table are ordered with respect to mean number of evaluated mappings per run. For example, the second column is SA with 1704 iterations ($L = 32$), the least iterations per run. The third column is GA with $i_{max} = 2048$.

Table 20 shows the expected number of iteration to reach within p percent of global optimum by repeating the GA or SA algorithm over and over again. SA+AT results are copied directly from the earlier experiment. Figure 17 shows the same information for $p = 0\%$, 2%, 5% and 10%.

SA+AT by default ($L = 32$) does mean of 1704 mappings per run, and converges to global optimum with probability $1,6\%$. This yields the mean of 106500 mapping iterations to reach global optimum by repeating the SA algorithm over and over again. GA does mean of 2048 mappings per run and converges to global optimum with probability $0,9\%$. This yields the mean of 227600 iterations to reach global optimum by repeating the GA algorithm. In this case, SA needs 53% less iterations than GA to reach global optimum.

The most efficient SA action takes place with $L = 64$, where SA needs approximately 92500 iterations to reach global optimum. The most efficient GA case is $i_{max} = 4096$, where 215600 iterations is needed. SA needs 57% less iterations.

GA is less efficient in this comparison, but there are biasing issues that are worth considering in the experiment.

The first issues is that the free parameters of GA were separately chosen for each i_{max} value. This should affect GA results positively, because the best GA parameters were

Table 19. *SA+AT and GA convergence compared with respect to global optimum. Automatic parameter selection method in SA+AT chooses $L = 32$. Mean mappings per run for GA is the i_{max} value. Values in table show proportion of optimization runs that converged within p from global optimum. Higher value is better.*

	Proportion of runs within limit p					
Algorithm	SA+AT	GA	SA+AT	GA	SA+AT	GA
Mean mappings / run	1 704	2 048	3 516	4096	7 588	8192
$p = \frac{t}{t_o} - 1$	$L = 32$		$L = 64$		$L = 128$	
+0%	0.016	0.009	0.038	0.019	0.080	0.036
+1%	0.026	0.013	0.061	0.028	0.129	0.052
+2%	0.051	0.029	0.117	0.058	0.224	0.101
+3%	0.084	0.051	0.173	0.100	0.311	0.155
+4%	0.119	0.071	0.230	0.124	0.391	0.187
+5%	0.154	0.091	0.287	0.149	0.468	0.226
+6%	0.201	0.128	0.355	0.208	0.553	0.298
+7%	0.264	0.173	0.439	0.274	0.653	0.384
+8%	0.337	0.244	0.536	0.376	0.757	0.495
+9%	0.400	0.296	0.606	0.432	0.819	0.560
+10%	0.461	0.354	0.675	0.486	0.870	0.620
+11%	0.531	0.428	0.747	0.569	0.917	0.700
+12%	0.605	0.500	0.812	0.630	0.952	0.761
+13%	0.664	0.562	0.864	0.689	0.972	0.807
+14%	0.726	0.616	0.906	0.728	0.983	0.841
+15%	0.780	0.666	0.935	0.760	0.992	0.867
+16%	0.831	0.712	0.958	0.796	0.996	0.897
+17%	0.868	0.751	0.974	0.820	0.998	0.916
+18%	0.903	0.787	0.984	0.848	0.999	0.934
+19%	0.930	0.822	0.990	0.885	0.999	0.952
+20%	0.953	0.854	0.994	0.919	1.000	0.967
+21%	0.968	0.885	0.997	0.943		0.978
+22%	0.979	0.913	0.998	0.961		0.987
+23%	0.985	0.936	0.999	0.973		0.991
+24%	0.991	0.954	1.000	0.984		0.996
+25%	0.995	0.968		0.991		0.998
+26%	0.996	0.978		0.994		0.999
+27%	0.998	0.985		0.997		0.999
+28%	0.999	0.990		0.998		0.999
+29%	1.000	0.993		0.999		1.000
+30%		0.995		0.999		
+31%		0.997		1.000		
...		...				
+34%		1.000				

Table 20. *Approximate expected number of mappings for SA+AT and GA to obtain objective value within p percent of the global optimum. SA+AT chooses $L = 32$ by default. Best values are in boldface on each row. Lower value is better.*

	Estimated number of mappings					
Algorithm	SA+AT	GA	SA+AT	GA	SA+AT	GA
Mean mappings / run	1 704	2 048	3 516	4096	7 588	8192
p	$L = 32$		$L = 64$		$L = 128$	
0%	106500	227600	**92500**	215600	94900	227600
1%	65500	157500	**57600**	146300	58800	157500
2%	33400	70600	**30100**	70600	33900	81100
3%	**20300**	40200	**20300**	41000	24400	52900
4%	**14300**	28800	15300	33000	19400	43800
5%	**11100**	22500	12300	27500	16200	36200
6%	**8500**	16000	9900	19700	13700	27500
7%	**6500**	11800	8000	14900	11600	21300
8%	**5100**	8400	6600	10900	10000	16500
9%	**4300**	6900	5800	9500	9300	14600
10%	**3700**	5800	5200	8400	8700	13200
11%	**3200**	4800	4700	7200	8300	11700
12%	**2800**	4100	4300	6500	8000	10800
13%	**2600**	3600	4100	5900	7800	10200
14%	**2300**	3300	3900	5600	7700	9700
15%	**2200**	3100	3800	5400	7600	9400
16%	**2100**	2900	3700	5100	...	9100
17%	**2000**	2700	3600	5000		8900
18%	**1900**	2600	3600	4800		8800
19%	**1800**	2500	3600	4600		8600
20%	**1800**	2400	3500	4500		8500
21%	**1800**	2300	...	4300		8400
22%	**1700**	2200		4300		8300
23%	...	2200		4200		8300
24%		2100		4200		8200
25%		2100		4100		...
26%		2100		...		
27%		2100				
28%		2100				
29%		2100				
30%		2100				
31%		2100				
32%		2000				

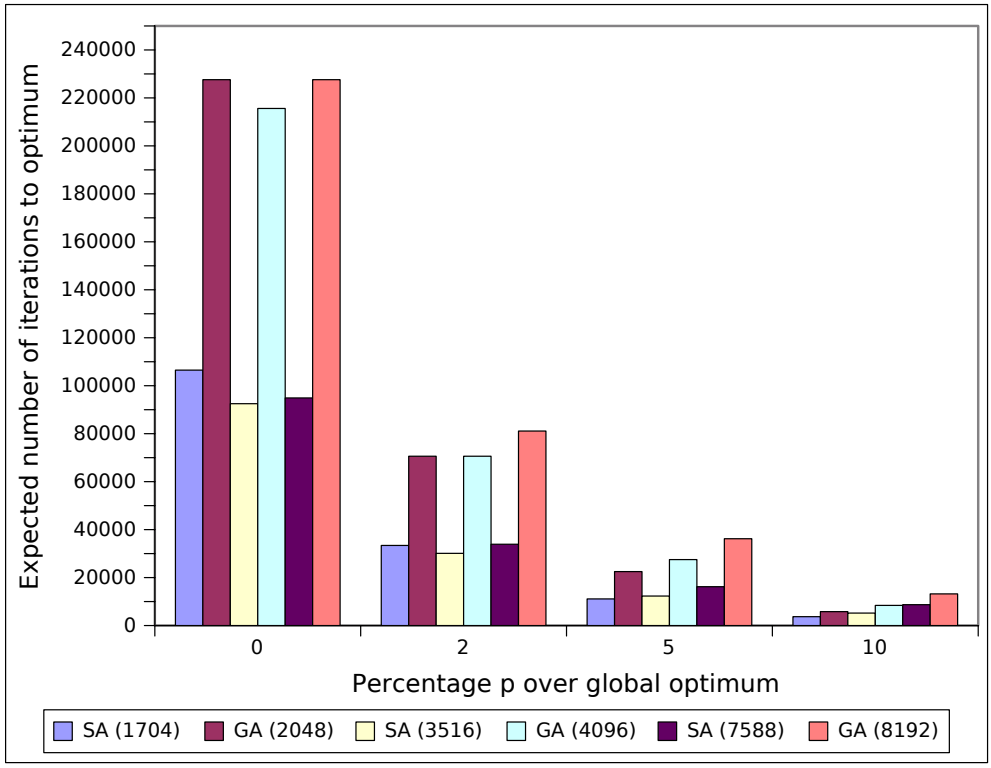


Fig. 17. *Expected number of iterations to reach global optimum within $p = 0\%, 2\%, 5\%$ and 10% for GA and SA. Lower p value on X-axis is better. Lower number of iterations on Y-axis is better.*

chosen for each case separately. In SA cases, only the L variable is varied.

The second bias issue comes from the effectiveness of GA crossover that should combine solutions to create new ones. The crossover algorithm is not specifically tuned for this problem. We use uniform crossover because it is unbiased to any particular problem. For this reason it is also not aware task mapping details. Many task mapping papers use either the one-point or two-point crossover with GA. These heuristics are affected by the ordering, i.e. the numbering, of tasks themselves. The optimization algorithm should ideally not depend on the permutation of numberings in which the tasks are coded into chromosome. However, an intelligent heuristics could determine a good permutation where one- or two-point crossovers are efficient. Detailed application analysis could reveal dependencies of tasks that could be exploited to create an application specific crossover heuristics. This is, however, a research interest we are not pursuing. Wider applicability of a mapping heuristics requires that intrinsic

details of specific problems are not assumed.

Experiment data, scripts and instructions how to repeat and verify the GA experiment is available at DCS task mapper web page [22] under the section “Experimental data”.

7. RECOMMENDATIONS FOR USING SIMULATED ANNEALING

This chapter lists recommendations for publishing about SA and using it for task mapping.

7.1 On publishing results for task mapping with Simulated Annealing

There are several guidelines that apply for publishing data on parallel computation [5] and heuristics in general [6]. We present recommendations for publishing results on task mapping with SA:

1. Report pseudocode for the algorithm. A textual description of the algorithm is often too ambiguous for reproducing the experiment. Specify temperature scale, cost, move, and acceptance functions.

2. Report numerical values for constants for the algorithm. Specify temperature scaling factor, initial and final temperatures and the number of iterations per temperature level. These are needed to re-produce the same results.

3. Report the rate of convergence. Plot the number of mapping iterations against optimized values (objective space). Report mean and median number of iterations.

4. Compare the heuristics with a global optimum for a trivial problem that can be solved by brute force. Report the proportion of optimization runs, and the number of iterations, that reach within p percent of the global optimum.

5. Report the optimization time per iteration. This is more or less the simulation time per mapping.

7.2 Recommended practices for task mapping with Simulated Annealing

We recommend following practices for task mapping with SA:

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. α is the number of neighboring mapping solutions. The number of mapping iterations should grow as a function of problem complexity parameters N and M unless experimental results indicate good results for a specific L value. Publications of the Thesis show convergence results for using α and its multipliers. [20] suggests that the number of mapping iterations per temperature level should be a multiple of α.

2. Use geometric temperature schedule with $0.90 \leq q \leq 0.99$. Most known results use values in this range. Our experiments have found $q = 0.95$ to be a suitable value. The L value has to be adjusted with the q variable. Dropping q from 0.95 to 0.90 implies that the number of iterations in a given temperature range halves unless L is doubled.

3. Use a systematic method for choosing the initial and final temperatures, e.g. one published in [P2] [P6] [P7] [9] [8] [20] [24] [45] [56] [88]. A systematic method decreases manual work and the risk of careless parameter selection. A systematic method may also decrease optimization time.

4. Use a normalized exponential acceptance function (5). This implies a normalized temperature range $T \in (0,1]$ which makes annealing schedules more comparable between problems. It is easy to select a safe but wasteful range when temperature is normalized, e.g. $T \in (0.0001,1]$. It is also easier to add new influencing factors into the cost function since the initial cost does not directly affect the selection of initial temperature when a normalized acceptance function is used. Instead, the relative change in cost function value in moves is a factor in the selection of initial temperature.

5. Use the ST (single task) move function 3.3.1, if in doubt. It is the most common heuristics which makes also the comparison easier to other works.

6. Run the heuristics several times for a given problem. The solution quality variance can be significant. As an example, Table 11 shows the probability

of convergence. The values can be used to compute a risk for not reaching a solution within p percent of the optimum in a given number of repetitions.

7. 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 stopped earlier without sacrificing reliability.

8. ON RELEVANCE OF THE THESIS

Table 8 displays citations to the Publications of the Thesis. We found 26 publications that refer to these by using Google's normal and scholar search [71], IEEE Xplore [33] and ACM digital library [1]. There are 14 publications that use or apply a method presented in publications of this Thesis. Some of these use a part of the method, and possibly modify it to their needs. [P2] is the most applied paper with 6 publications using or adopting a method from it. [P3] is the most cited paper, with 14 citations.

Table 21. *Table of citations to the publications in this Thesis. "Citations" column indicates publications that refer to the given paper on the "Publication" column. Third column indicates which of these papers use or apply a method presented in the paper. The method may be used in a modified form.*

Publication	Citations	Uses or applies a method proposed in the paper
P1	[3] [38] [39]	[3] [38] [39]
P2	[10] [11] [29] [32] [61] [67] [68] [69]	[10] [11] [16] [29] [61] [67] [69]
P3	[13] [14] [15] [18] [19] [36] [48] [53] [73] [74] [75] [76] [79] [80] [84]	[53] [73] [75] [76]
P4	[3] [65] [83]	[3]
P5	None	None
P6	[32] [65] [67]	[32] [67]
P7	None	None

SA method proposed in [P1] was intended for Koski [39] design framework for MP-SoCs. The method was used in several Koski related publications [3] [38] [39]. The method was slow due to broad annealing temperature range. This lead to development

of automatic parameter selection method proposed in [P2]. Optimal subset mapping (OSM) from [P4] is also included in the Koski design framework.

SA parameter selection method proposed in [P2] was applied in [10] [11] [16] [29] [61] [67] [69].

Caffarena [10] [11] binds operations to FPGA resources to minimize area. They use the normalized exponential acceptor (5) to normalize the temperature range which eases the temperature selection.

Chang [16] mapped applications onto a NoC where physical defects were detoured by rewiring. Rewriting required remapping the application to new physical tiles on the NoC by using the parameter selection method proposed in [P2].

Greenfield [29] mapped 250 random task graphs to a NoC with 8x8 array of tiles by using SA and the proposed parameter selection method. The effect of the method was not reported.

Ravindran [61] and Satish [67] followed techniques from the parameter selection method of Koch [45] and [P2], but they are vague in details. The algorithm they propose mostly resembles the Koch method. Satish also reports using the initial and final temperature calculations from [P6]. Ravindran and Satish report that SA is found to be better than DLS and DA for large task graphs with irregular architectures, but DA does better for smaller task graphs.

Satish [69] used the parameter selection method from [P2] to optimize the scheduling for tasks and CPU-GPU data transfers. Their problem space is scheduling rather than mapping, but they only had to replace the cost and move functions to use the proposed method. They applied the method to Edge detection and convolutional neural networks on large data sets. The edge detection problem had 8 tasks and 13 data items. Two convolutional neural networks had 740 tasks and 1134 data items, and 1600 tasks and 2434 data items. The SA approach won the heuristic depth first search approach published in [82] by a performance factor 2. SA decreased data transfers by a factor of 30 compared to unoptimized implementation. The result was within 16% of the global optimum. Jitter was a problematic side effect of stochastic SA algorithm. The standard deviation of the SA results was 4% of the mean value.

SA method in [P3] was applied in Lee [53] by using the normalized inverse exponential acceptance function (2) and the L value. The SA algorithm was compared with

a proposed task mapping algorithm. The proposed algorithm decreased communication costs by 2% on average compared to SA. The proposed algorithm was specifically tuned to the problem by adding detailed knowledge of the problem space, such as the topology of the NoC. The SA algorithm was used as a generic task mapping algorithm without the details.

Shafik [73] used the SA method in [P3] to map task graphs to optimize performance, power and soft-errors. When performance and/or power was optimized separately or simultaneously, the SA method gave better results than Shafik's proposed method. Shafik's method factored soft-errors into the cost function, but left them out of SA cost function. Consequently, Shafik's method was better at solving soft-errors. It was not reported how SA would perform with soft-errors included in the cost function. It was reported that SA gave 5% better power, or 10% better performance than the proposed method, when power and performance were optimized separately. Simultaneous SA optimization of power and performance showed 2% better power and 3% better performance compared to the proposed method. Shafik also applies the SA method in [75] and [76].

Houshmand [32] uses the parameterization method from [P2] and initial temperature calculation from [P6]. They propose an improvement to these methods by applying parallelism to SA exploration. However, results are inconclusive. Proposed method is presented as a broken pseudocode, which makes reproducing the benchmark uncertain and hard. The comparison between the original and the improved method did not show relative efficiency. The improved method used 5 to 10 times the mapping iterations compared to the original method. Results indicate that the original method has better relative efficiency. It was not shown how the original method would have performed with 5 to 10 times the iterations. This could be done by repeatedly running the algorithm 5 to 10 times, or possibly scaling the L value by a factor from 5 to 10.

9. CONCLUSIONS

9.1 Main results

The Thesis collects together a state of the art survey on SA for task mapping, new data on annealing convergence and recommendations for using SA in task mapping. The results presented in the Thesis are still a small step towards understanding the parameter selection and the convergence process.

The first research question in Chapter 4 was how to select optimization parameters for a given point in DSE problem space. This research question was answered in the recommendations in Chapter 7 based on the survey in Chapter 3, publications [P1-7] and experiments in Chapter 6.

The second research question was how many mapping iterations are needed to reach a given solution quality. Chapter 6 reported new information about converge rates with respect to global optimum by estimating the number of iterations for 32 node random task graphs. This information helps task mappers decide on the SA iterations to reach a given level of solution quality. As far as we know, 32 nodes in this work is the largest random task graph for which global optimum has been determined. This bound is not likely to grow fast as the mapping problem is NP.

SA was also found to be over 50% more efficient than GA in a task mapping experiment. SA and GA are both popular heuristics for task mapping, but neither has yet won over the other.

The third research question was how does each parameter of the algorithm affect the convergence rate. Table 12 displayed the effect of L value to the expected number of iterations to reach a given solution quality. The proposed $L = N(M-1)$ in SA+AT was most efficient for a quality target within 4 to 5 percent of the global optimum for 32 nodes and 2 PEs. However, doubling the proposed value to $L = 2N(M-1)$ was the most efficient value to reach within 0 to 3 percent of the global optimum.

Section 6.2.1 presented experiments where exponential acceptors were found better than inverse exponential acceptors. Therefore, we recommend exponential function instead of the inverse exponential function.

Section 6.2.2 presented an experiment where no significant effect was found between the solution quality and the zero transition probability in the acceptance function.

The fourth research question was how to speedup convergence rate. SA optimization process resembles Monte Carlo optimization in high temperatures that converges very slowly. The Monte Carlo effect is compared with RM in publications [P4] [P7]. Any optimization algorithm should do better than RM unless the problem lacks tractability for systematic solutions. One intractable case is presented in [P7].

[P2] proposes an initial temperature computation method that avoids most of the Monte Carlo phase. The paper also proposes a final temperature computation method that stops optimization when optimization is mostly greedy and the algorithm may get stuck into a local optimum. The method is further extended to KPNs in [P7]. The final temperature computation is also refined to save optimization time by filtering out tasks with insignificant amount of computation.

[P4] presents a rapidly converging mapping algorithm. The algorithm does not obtain good solution quality, but it could be used to obtain an initial solution for more expensive algorithms.

Section 6.1 presented data on L value effect on trade-off between run-time and solution quality. This information can be used to estimate the optimizing effort for graphs with up to 32 nodes. The number of required mapping iterations are given to justify expenses in optimization time to reach a given level of quality.

The fifth research question was how often does the algorithm converge to a given solution quality. Section 6.1 helps task mapper to target a specific distance from the global optimum and gives the probability for annealing within that distance.

There are several aspects that have not been addresses in the Thesis. Future work needs to be done on following aspects:

- It lacks experiments on real-world applications modeled as task graphs
- The resemblance between random task graphs and real-world applications is not shown

- Iterations per temperature level should not be constant. Less iterations should be used in the early Monte Carlo phase of annealing, as random mapping converges very slowly.
- Architectures and interconnects used in simulations were mostly homogeneous. Heterogeneous architectures make the problem space more complex, and thus might reveal convergence problems in proposed optimization algorithms.
- The proposed methods have not been tested on real systems which raises doubt on the validity of testing and simulation models presented

This Work on task mapping could be applied on other problem fields, such as a job shop scheduling problem [52]. Task mapping can be generalized to any problem where several objects are placed on several resources. For example, a series of tasks can be mapped and scheduled to workers and robots.

BIBLIOGRAPHY

[1] ACM digital library, 2011. [online] http://portal.acm.org

[2] Ali, S., Kim, J.-K., Siegel, H. J. and Maciejewski, A. A., "Static heuristics for robust resource allocation of continuously executing applications", Journal of Parallel and Distributed Computing, Vol. 68, Issue 8, August 2008, pp. 1070-1080, ISSN 0743-7315, DOI: 10.1016/j.jpdc.2007.12.007.

[3] Arpinen, T., Salminen, E. and Hämäläinen, T. D., "Performance Evaluation of UML2-Modeled Embedded Streaming Applications with System-Level Simulation", EURASIP Journal on Embedded Systems, 2009. [online] http://www.hindawi.com/journals/es/2009/826296.html

[4] Bacivarov, I., Haid, W., Huang, K. and Thiele, L., "Methods and Toold for Mapping Process Networks onto Multi-Processor Systems-On-Chip", Handbook of Signal Processing Systems, ISBN 978-1-4419-6345-1, Springer, pp. 1007-1040, 2010.

[5] Bailey, D. H., "Twelve ways to fool the masses when giving performance results on parallel computers", Supercomputer Review, Vol. 4, No. 8, pp. 54-55, 1991. [online] http://crd.lbl.gov/~dhbailey/dhbpapers/twelve-ways.pdf

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

[7] Beaty, S. J., "Genetic Algorithms versus Tabu Search for Instruction Scheduling", International conference on Neural Network and Genetic Algorithms, pp. 496-501, Springer, 1993.

[8] Bollinger, S. W. and Midkiff, S. F., "Heuristic Technique for Processor and Link Assignment in Multicomputers", IEEE Transactions on Computers, Vol. 40, pp. 325-333, 1991.

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

[10] Caffarena, G., Lopez, J., Leyva, G., Carreras, C. and Nieto-Taladriz, O., "Architectural Synthesis of Fixed-Point DSP Datapaths using FPGAs", International journal of reconfigurable computing, Article 8, January 2009. [online] http://downloads.hindawi.com/journals/ijrc/2009/703267.pdf

[11] Caffarena, G., "Combined Word-Length Allocation and High-Level Synthesis of Digital Signal Processing Circuits", Universidad politecnica de Madrid, Doctoral thesis, 2008. [online] http://oa.upm.es/1822/1/GABRIEL_CAFFARENA_FERNANDEZ.pdf

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

[13] Carvalho, E. and Moraes, F., "Congestion-aware Task Mapping in Heterogeneous MPSoCs", International Symposium on System-on-Chip 2008, Tampere, Finland, 2008.

[14] Carvalho, E., Marcon, C., Calazans, N. and Moraes, F., "Evaluation of Static and Dynamic Task Mapping Algorithms in NoC-Based MPSoCs", International Symposium on System-on-Chip 2009, Tampere, Finland, pp. 87-90 2009. [online] http://www.inf.pucrs.br/~moraes/my_pubs/papers/2009/soc09_carvalho.pdf

[15] Carvalho, E., Calazans, N. and Moraes, F., "Investigating Runtime Task Mapping for NoC-based Multiprocessor SoCs", Proceedings of the 17th IFIP/IEEE International Conference on Very Large Scale Integration, 2009. [online] http://www.inf.pucrs.br/~moraes/my_pubs/papers/2009/ifip_vlsi_soc_2009_id47_carvalho.pdf

[16] Chang, Y.-C., Chiu, C.-T., Lin, S.-Y. and Liu, C.-K., "On the design and analysis of fault tolerant NoC architecture using spare routers", Proceedings of the 16th Asia and South Pacific Design Automation Conference (ASPDAC '11), pp. 431-436, 2011. [online] http://portal.acm.org/citation.cfm?id=1950906

[17] Chen, H., Flann, N. S. and Watson, D. W., "Parallel genetic simulated annealing: A massively parallel SIMD approach", IEEE Transactions on Parallel and Distributed Computing, Vol. 9, No. 2, pp. 126-136, 1998.

[18] Chowdhury, S. R., Roy, A. and Saha, H., "ASIC Design of a Digital Fuzzy System on Chip for Medical Diagnostic Applications", Journal of medical systems, Springer, 2009.

[19] Chowdhury, S. R., Chakrabarti, D. and Saha, H., "Development Of An Fpga Based Smart Diagnostic System For Spirometric Data Processing Applications", International Journal on Smart Sensing and Intelligent Systems, Vol. 1, No. 4, 2008.

[20] Coroyer, C. and Liu, Z., "Effectiveness of heuristics and simulated annealing for the scheduling of concurrent tasks an empirical comparison", Rapport de recherche de l'INRIA Sophia Antipolis (1379), 1991.

[21] Dasgupta, S., Papadimitriou, C. H. and Vazirani, U. V., "Algorithms", McGraw-Hill Science/Engineering/Math, 1st edition, ISBN 978-0073523408, 2006. [online] http://www.cs.berkeley.edu/~vazirani/algorithms.html

[22] DCS task mapper, "a task mapping and scheduling tool for multiprocessor systems", 2010. [online] http://wiki.tut.fi/DACI/DCSTaskMapper

[23] Dorigo, M. and Stützle, T., "Ant Colony Optimization", MIT press, ISBN 0-262-04219-3, 2004.

[24] Ercal, F., Ramanujam, J. and Sadayappan, P., "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning", ACM, pp. 210-221, 1988. XXX

[25] Ferrandi, F., Pilato, C., Sciuto, D. and Tumeo, A., "Mapping and scheduling of parallel C applications with Ant Colony Optimization onto heterogeneous

reconfigurable MPSoCs", Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific, pp. 799-804, 2010.

[26] Girkar, M. and Polychronopoulos, C. D., "Automatic extraction of functional parallelism from ordinary programs", IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 2, pp. 166-178, March 1992.

[27] Givargis, T., Vahid, F. and Henkel, J. "System-level Exploration for Pareto-optimal Configurations in Parameterized Systems-on-a-chip", International Conference on Computer-Aided Design (ICCAD '01), 2001. [online] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.9116&rep=rep1&type=pdf

[28] Glover, F., "Future Paths for Integer Programming and Links to Artificial Intelligence", Computers and Operations Research, Vol. 13, No. 5, pp. 533-549, 1986.

[29] Greenfield, D., Banerjee, A., Lee, J.-G. and Moore, S., "Implications of Rent's Rule for NoC Design and Its Fault-Tolerance", Proceedings of the First International Symposium on Networks-on-Chip, pp. 283-294, 2007.

[30] Gries, M., "Methods for evaluating and covering the design space during early design development", Integration, the VLSI Journal, Vol. 38, Issue 2, pp. 131-183, 2004.

[31] Holland, J. H., "Adaptation in Natural and Artificial Systems", The MIT Press, ISBN 978-0262581110, 1975.

[32] Houshmand, M., Soleymanpour, E., Salami, H., Amerian, M. and Deldari, H., "Efficient Scheduling of Task Graphs to Multiprocessors Using A Combination of Modified Simulated Annealing and List based Scheduling", Third International Symposium on Intelligent Information Technology and Security Informatics, 2010.

[33] IEEE Xplore digital library, 2011. [online] http://ieeexplore.ieee.org/

[34] Jantsch, A., "Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation (Systems on Silicon)", Morgan Kaufmann, 2003.

[35] *jobqueue*, "A tool for parallelizing jobs to a cluster of computers", 2010. [online] http://zakalwe.fi/~shd/foss/jobqueue/

[36] Jueping, C., Lei, Y., Gang, H., Yue, H., Shaoli, W. and Zan, L., "Communication- and Energy-Aware Mapping on Homogeneous NoC Architecture with Shared Memory", 10th IEEE International Conference on Solid-State and Integrated Circuit Technology, 2010. [online] http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5667751

[37] Kahn, G., "The semantics of a simple language for parallel programming", Proceedings of IFIP Congress 74, Information Processing 74, pp. 471-475, 1974. [online] http://www1.cs.columbia.edu/~sedwards/papers/kahn1974semantics.pdf

[38] Kangas, T., "Methods and Implementations for Automated System on Chip Architecture Exploration", Doctoral thesis, Tampere University of Technology, 2006. [online] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.166&rep=rep1&type=pdf

[39] Kangas, T., Kukkala, P., Orsila, H., Salminen, E., Hännikäinen, M., Hämäläinen, T. D., Riihimäki, J. and Kuusilinna, K., "UML-Based Multiprocessor SoC Design Framework", ACM Transactions on Embedded Computing Systems, Vol. 5, No. 2, pp. 281-320, 2006.

[40] Kangas, T., Riihimäki, J., Salminen, E., Kuusilinna, K. and Hämäläinen, T. D., "Using a communication generator in SoC architecture exploration", International Symposium on System-on-Chip 2003, pp. 105-108, 2003. [online] http://daci.digitalsystems.cs.tut.fi:8180/pubfs/fileservlet?download=true&filedir=dacifs&freal=Kangas_-_Using_a_Communication_Gen.pdf&id=70297

[41] Keutzer, K., Malik, S., Newton, A. R., Rabaey, J. M. and Sangiovanni-Vincentelli, A., "System-Level Design: Orthogonalization of Concenrs and Platform-Based Design", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 19, No. 12, 2000.

[42] Kienhuis, B., Deprettere, E. F., Wolf, P. and Vissers, K., "A Methodology to

Design Programmable Embedded Systems - The Y-chart Approach", Embedded Processor Design Challenges, SAMOS 2001, LNCS 2268, pp. 18-37, 2002.

[43] Kim, J.-K., Shivle, S., Siegel, H. J., Maciejewski, A. A., Braun, T. D., Schneider, M., Tideman, S., Chitta, R., Dilmaghani, R. B., Joshi, R., Kaul, A., Sharma, A., Sripada, S., Vangari, P. and Yellampalli, S. S., "Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment", Journal of Parallel and Distributed Computing, Elsevier, Vol. 67, pp. 154-169, 2007.

[44] Kirkpatrick, S., Gelatt, C. D. and Vecchi, M. P., "Optimization by simulated annealing", Science, Vol. 200, No. 4598, pp. 671-680, 1983.

[45] Koch, P., "Strategies for Realistic and Efficient Static Scheduling of Data Independent Algorithms onto Multiple Digital Signal Processors", Doctoral thesis, The DSP Research Group, Institute for Electronic Systems, Aalborg University, Aalborg, Denmark, December 1995.

[46] *kpn-generator*, "A program for generating random Kahn Process Network graphs", 2009. [online] http://zakalwe.fi/~shd/foss/kpn-generator/

[47] Kulmala, A., Lehtoranta, O., Hämäläinen, T. D. and Hännikäinen, M., "Scalable MPEG-4 Encoder on FPGA Multiprocessor SOC", EURASIP Journal on Embedded Systems, Vol. 2006, pp. 1-15, 2006.

[48] Kumar, T. S. R., "On-Chip Memory Architecture Exploration of Embedded System on Chip", Doctoral thesis, Indian Institute of Science, Bangalore, 2008. [online] http://www.serc.iisc.ernet.in/graduation-theses/Rajesh_TS.pdf

[49] Kwok, Y.-K., Ahmad, I. and Gu, J., "FAST: A Low-Complexity Algorithm for Efficient Scheduling of DAGs on Parallel Processors", Proceedings of International Conference on Parallel Processing, Vol. II, pp. 150-157, 1996.

[50] Kwok, Y.-K. and Ahmad, I., "FASTEST: A Practical Low-Complexity Algorithm for Compile-Time Assignment of Parallel Programs to Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 2, pp. 147-159, 1999.

[51] Kwok, Y.-K. and Ahmad, I., "Static scheduling algorithms for allocating directed task graphs to multiprocessors", ACM Comput. Surv., Vol. 31, No. 4, pp. 406-471, 1999.

[52] Laarhoven, P. J. M., Aarts, E. H. L. and Lenstra, J. K., "Job Shop Scheduling by Simulated Annealing", Journal of Operations Research, Vol. 40., No. 1, pp. 113-125, 1992. [online] http://www.jstor.org/stable/171189

[53] Lee, S.-H., Yoon, Y.-C. and Hwang, S.-Y., "Communication-aware task assignment algorithm for MPSoC using shared memory", Elsevier, Journal of Systems Architecture, Vol. 56, No. 7, pp. 233-241, 2010. [online] http://dx.doi.org/10.1016/j.sysarc.2010.03.001

[54] Lehtoranta, O., Salminen, E., Kulmala, A., Hännikäinen, M. and Hämäläinen, T. D., "A parallel MPEG-4 encoder for FPGA based multiprocessor SoC", International conference on Field Programmable Logic and Applications, pp. 380-385, 2005.

[55] Copyright license, "GNU Lesser General Public License, version 2.1", Free Software Foundation Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA, 1999. [online] http://www.gnu.org/licenses/lgpl-2.1.html

[56] Lin, F.-T. and Hsu, C.-C., "Task Assignment Scheduling by Simulated Annealing", IEEE Region 10 Conference on Computer and Communication Systems, Hong Kong, September 1990.

[57] Matousek, J. and Gärtner, B., "Understanding and Using Linear Programming", Springer, ISBN 978-3540306979, 2006.

[58] Nanda, A. K., DeGroot, D. and Stenger, D. L., "Scheduling Directed Task Graphs on Multiprocessors using Simulated Annealing", Proceedings of 12th IEEE International Conference on Distributed Systems, pp. 20-27, 1992.

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

[60] Ylönen, T., Campbell, A., Beck, B., Friedl, M., Provos, N., Raadt, T. and Song, D., "OpenSSH", 1999. [online] *http://openssh.com*

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

[62] Ritchie, D. M., "The development of the C language", The second ACM SIGPLAN conference on History of programming languages, 1993. [online] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.3655&rep=rep1&type=pdf

[63] Rossum, G. and Drake, F. L., "The Python Language Reference", Python Software Foundation, 2010. [online] http://www.sfr-fresh.com/unix/misc/Python-2.7-docs-pdf-a4.tar.gz:a/docs-pdf/reference.pdf

[64] SA+AT C reference implementation. [online] http://zakalwe.fi/~shd/task-mapping

[65] Salminen, E., "On Design and Comparison of On-Chip Networks", Doctoral thesis, Tampere University of Technology, 2010. [online] http://dspace.cc.tut.fi/dpub/handle/123456789/6543

[66] Salminen, E., Kulmala, A. and Hämäläinen, T. D., "HIBI-based Multiprocessor SoC on FPGA", IEEE International Symposium on Circuits and Systems (ISCAS), Vol. 4, pp. 3351-3352, 2005.

[67] Satish, N. R., "Compile Time Task and Resource Allocation of Concurrent Applications to Multiprocessor Systems", University of California, Berkeley, Doctoral thesis, Technical Report No. UCB/EECS-2009-19, 2009. [online] http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-19.html

[68] Satish, N. R., Ravindran, K. and Keutzer, K., "Scheduling Task Dependence Graphs with Variable Task Execution Times onto Heterogeneous Multiprocessors", Proceedings of the 8th ACM international conference on Embedded software, 2008. [online] http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-42.html

[69] Satish, N., Sundaram, N. and Kreutzer, K., "Optimizing the use of GPU Memory in Applications with Large data sets", International Conference on High Performance Computing (HiPC), 2009. [online] http://www.cs.berkeley.edu/~narayans/Publications_files/HiPC2009.pdf

[70] Sato, M., "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors", ACM, Proceedings of the 15th international symposium on System Synthesis, pp. 109-111, 2002.

[71] Google's scholar search, 2011. [online] http://scholar.google.com

[72] Weigert, J., Schroeder, M. and Laumann, O., Text terminal program multiplexer: "GNU Screen", 1987. [online] http://www.gnu.org/software/screen/

[73] Shafik, R. A., "Investigation into Low Power and Reliable System-on-Chip Design", Doctoral thesis, University of Southampton, 2010. [online] http://eprints.ecs.soton.ac.uk/21331/

[74] Shafik, R. A. and Al-Hashimi, B. M., "Reliability Analysis of On-Chip Communication Architectures: An MPEG-2 Video Decoder Case Study", Elsevier Journal of Microprocessors and Microsystems, 2010. [online] http://eprints.ecs.soton.ac.uk/20924/1/micpro2010.pdf

[75] Shafik, R. A., Al-Hashimi, B. M., Kundu, S. and Ejlali, A., "Soft Error-Aware Voltage Scaling Technique for Power Minimization in Application-Specific Multiprocessor System-on-Chip", Journal of Low Power Electronics, Vol. 5, No. 2. pp. 145-156, 2009.

[76] Shafik, R. A., Al-Hashimi, B. M., Chakrabarty, K., "Soft error-aware design optimization of low power and time-constrained embedded systems", Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1462-1467, 2010. [online] http://eprints.ecs.soton.ac.uk/18170/6/10.6_4_0678.pdf

[77] Shroff, P., Watson, D., Flann, N. and Freund, R., "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments", The fifth IEEE Heterogeneous Computing Workshop, pp. 98-104, 1996.

[78] Sih, G. C. and Lee, E. A., "A Compile-Time Scheduling Heuristics for Interconnection-Constrained Heterogeneous Processor Architectures", IEEE Transaction on Parallel and Distributed Systems, Vol. 4, No. 2, pp. 175-187, 1993.

[79] Singh, A. K., Srikanthan, T., Kumar, A. and Jigang, W., "Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms", Elsevier, Journal of Systems Architecture, Vol. 56, pp. 242-255, 2010. [online] http://www.es.ele.tue.nl/~akash/files/kumarJSA2010b.pdf

[80] Singh, A. K., Jigang, W., Kumar, A. and Srikanthan, T., "Run-time mapping of multiple communicating tasks on MPSoC platforms", Procedia Computer Science, Vol. 1, Issue 1, ICCS 2010, pp. 1019-1026, 2010. [online] http://dx.doi.org/10.1016/j.procs.2010.04.113

[81] *Standard Task Graph Set*, "A collection of directed acyclic task graphs for benchmarking task distribution algorithms", 2003. [online] http://www.kasahara.elec.waseda.ac.jp/schedule,

[82] Sundaram, N., Raghunathan, A. and Chakradhar, S., "A framework for efficient and scalable execution of domain specific templates on GPUs", Proceedings of the IEEE International Parallal and Distributed Processing Symposium, 2009. [online] http://www.eecs.berkeley.edu/~narayans/About_Me_files/ipdps2009.pdf

[83] Tagel, M., Ellervee, P. and Jervan, G., "Design Space Exploration and Optimisation for NoC-based Timing Sensitive Systems", 12th Baltic Electronics Conference, Tallinn, Estonia, 2010.

[84] Wang, M. and Bodin, F., "Compiler-directed memory management for heterogeneous MPSoCs", Elsevier, Journal of Systems Architecture, available online, 2010. [online] http://dx.doi.org/10.1016/j.sysarc.2010.10.008

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

[86] Wolf, W., “The future of multiprocessor systems-on-chips”, Design Automation Conference 2004, pp. 681-685, 2004.

[87] Wolf, W., Jerraya, A. A. and Martin, G., “Multiprocessor System-on-Chip (MP-SoC) Technology”, IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, Vol. 27, No. 10, 2008.

[88] Xu, J. and Hwang, K., “A simulated annealing method for mapping production systems onto multicomputers”, Proceedings of the sixth conference on Artificial intelligence applications, IEEE Press, ISBN 0-8186-2032-3, pp. 130-136, 1990.

PUBLICATION 1

Hybrid Algorithm for Mapping Static Task Graphs on Multiprocessor SoCs

Heikki Orsila, Tero Kangas, and Timo D. Hämäläinen
Institute of Digital and Computer Systems
Tampere University of Technology
P.O. Box 553, 33101 Tampere, Finland
Email: {heikki.orsila, tero.kangas, timo.d.hamalainen}@tut.fi

***Abstract*— Mapping of applications on multiprocessor System-on-Chip is a crucial step in the system design to optimize the performance, energy and memory constraints at the same time. The problem is formulated as finding solutions to an objective function of the algorithm performing the mapping and scheduling under strict constraints. Our solution is a new hybrid algorithm that distributes the computational tasks modeled as static acyclic task graphs The algorithm uses simulated annealing and group migration algorithms consecutively and it combines a non-greedy global and greedy local optimization techniques to have good properties of both ways. The algorithm begins as coarse grain optimization and moves towards fine grained optimization. As a case study we used ten 50-node graphs from the Standard Task Graph Set and averaged results over 100 optimization runs. The hybrid algorithm gives 8% better execution time on a system with four processing elements compared to simulated annealing. In addition, the number of iterations increased only moderately, which justifies the new algorithm in SoC design.**

I. Introduction

Contemporary embedded system applications demand increasing computing power and reduced energy consumption at the same time. Multiprocessor System-on-Chip implementations have become more popular since it is often more reasonable to use several low clock rate processors than a single high-performance one. In addition, the overall performance can be increased by distributing processing on several microprocessors raising the level of parallelism.

However, efficient multiprocessor SoC implementation requires exploration to find an optimal architecture as well as mapping and scheduling of the application on the architecture. This, in turn, calls for optimization algorithms in which the cost function consists of execution time, communication time, memory, energy consumption and silicon area constraints, for example. The optimal result is obtained in a number of iterations, which should be minimized to make the optimization itself feasible. One iteration round consists of application task mapping, scheduling and as a result of that, evaluation of the cost function. In a large system this can take even days.

The principal problem is that in general the mapping of an application onto a multiprocessor system is an NP-problem. Thus verifying that any given solution is an optimum needs exponential time and/or space from the optimization algorithm. Fortunately it is possible in practice to device heuristics that can reach near optimal results in polynomial time and space for common applications.

We model the SoC applications as static acyclic task graphs (STGs) in this papers. Distributing STGs to speedup the execution time is a well researched subject [1], but multiprocessors SoC architectures represent a new problem domain with significantly more requirements compared to traditional multiprocessor systems.

This paper presents a new algorithm especially targeted to map and schedule applications for multiprocessor SoCs in an optimal way. In this paper the target is to optimize the execution time, but the algorithm is also capable of optimizing memory and energy consumption compared to previous proposals.

The hybrid algorithm applies a non-greedy global optimization technique known as simulated annealing (SA) and a greedy local optimization technique known as the group migration (GM) algorithm.

The basic concepts and the related work of task parallelization, SA and GM are presented in Section II. The contribution of this paper is the hybrid task mapping algorithm, which is described in Section III. The algorithm was evaluated with a set of task graphs and compared to the pure SA algorithm as reported in Section IV. Finally, the concluding remarks are given.

II. Related Work

Wild et. al considered simulated annealing and tabu search for SoC application parallelization [2]. Their algorithm uses a special critical path method to identify important tasks that need to be relocated. Also, Vahid [3] considered pre-partitioning to shrink optimization search space, and in addition they apply various clustering algorithms, such as group migration algorithm, to speedup the execution time. This is similar approach to us, in which we enhance simulated annealing with the group migration. Lei et al. [4] considered a two-step genetic algorithm for mapping SoC applications to speedup the execution time. Their approach starts with coarse grain optimizations and continues with fine grain optimizations as in our approach. Compared to our algorithm, they have stochastic algorithms in both steps. Our algorithm adds more reliability as the second step is deterministic.

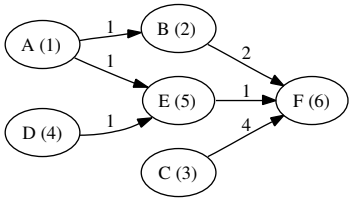


Fig. 1. An example STG with computational costs in nodes and communication costs at edges. Node F is the result node that is data dependent on all other nodes

A. Static Task Graphs

Nodes of the STG are finite deterministic computational tasks, and edges represent data dependencies between the nodes. Computational nodes block until their data dependencies are resolved. Node weights represent the amount of computation associated with a node. Edge weights represent amount of communication needed to transfer results between the nodes.

Fig. 1 shows an STG example. Node F is the result node which is data dependent on all other nodes. Nodes A, C and D are nodes without data dependencies, and hence they are initially ready to run.

Each node is mapped to a specific processing element in the SoC. The STG is fully deterministic, meaning that the complexity of the computational task is known in advance, and thus no load balancing technique, such as task migration, is not needed. The distribution is done at compile time, and the run-time is deterministic. When a processing element has been chosen for a node it is necessary to schedule the STG on the system. Schedule determines execution order and timing of execution for each processing element. The scheduler has to take into account the delays associated with nodes and edges, and data dependencies which affect the execution order.

B. Simulated Annealing

SA is a probabilistic non-greedy algorithm [5] that explores search space of a problem by annealing from a high to a low temperature state. The greediness of the algorithm increases as the temperature decreases, being almost greedy at the end of annealing. The algorithm always accepts a move into a better state, but also into a worse state with a changing probability. This probability decreases along with the temperature, and thus the algorithm becomes greedier. The algorithm terminates when the maximum number of iterations have been made, or too many consecutive moves have been rejected.

Fig. 2 shows the pseudo-code of the SA algorithm used in the hybrid algorithm. Implementation specific issues compared to the original algorithm are mentioned in Section IV-A. The **Cost** function evaluates badness of a specific mapping by calling the scheduler to determine execution time. S_0 is the initial state of the system, and T_0 is the initial temperature. **Temperature–Cooling** function computes a new temperature as a function of initial temperature T_0 and iteration i. **Move** function makes a random move to another state. **Random** function returns a random value from the interval $[0, 1)$. **Prob**

```
SIMULATED-ANNEALING(S_0, T_0)
 1  S ← S_0
 2  C ← COST(S_0)
 3  S_best ← S
 4  C_best ← C
 5  Rejects ← 0
 6  for i ← 1 to i_max
 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             Rejects ← 0
19        else Rejects ← Rejects + 1
20             if Rejects ≥ Rejects_max
21                then break
22  return S_best
```

Fig. 2. Pseudo-code of the simulated annealing algorithm

function computes a probability that a move that increases the cost is accepted.

C. Group Migration Algorithm

Group migration algorithm ([3], [6]) is a greedy local search technique that changes mapping of STG nodes one by one, accepting only moves that improve current solution as determined by the scheduler.

Fig. 3 shows pseudo-code of the GM algorithm used in the hybrid algorithm. The function **Group–Migration** calls the function **GM–Round** as long as the solution improves. Function **GM–Round** tries to move each task one by one from its PE to all other PEs. If it finds a move that decreases cost, it records the change, and restores the original mapping and goes to the next task. After all tasks have been tried, it takes the best individual move, applies that move on the mapping, and marks the associated task as non-movable. Any task that has been marked non-movable will not be considered as a movable candidate again. Then the algorithm starts from the beginning, trying each movable task again. This is continued until no cost decrease is found.

III. The Proposed Hybrid Algorithm

A. Mapping

The hybrid algorithm presented in this paper uses SA and GM algorithms to support each other. Fig. 4 shows pseudo-code of the main optimization loop. Initially the mapping is set by the function **Fast–Premapping** shown in Fig. 5. Fast premapping distributes node mappings so that parents of a

```
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  M_cost ← COST(S)
 3  Moved ← [False] * N_tasks
 4  for i ← 1 to N_tasks
 5  do M_task = NIL
 6     M_PE = NIL
 7     for t ← 0 to N_tasks − 1
 8     do if Moved[t] = True
 9           then continue
10        S_old ← S
11        for A ← 0 to N_PEs − 1
12        do if A = A_old
13              then continue
14           S[t] ← A
15           if COST(S) < M_cost
16              then continue
17           M_cost = COST(S)
18           M_task = t
19           M_agent = A
20        S ← S_old
21     if M_agent = NIL
22       then break
23     Moved[M_task] ← True
24     S[M_task] ← M_agent
25  return S
```

Fig. 3. Pseudo-code of the group migration algorithm

child are on different PEs. As an example, nodes in the Fig. 1 would be premapped to 3 PE system as follows: $F \mapsto PE\ 1$, $B \mapsto 1$, $E \mapsto 2$, $C \mapsto 3$, $A \mapsto 1$, and $D \mapsto 2$. SA and GM algorithms are called sequentially until the optimization terminates. There are two specialties in this approach.

First, SA is called many times, but with each time the initial temperature is half of that of the previous iteration. The SA algorithm itself can visit a good state but leave it for a worse state with certain probability, since the algorithm is not totally greedy. To overcome this deficiency our SA implementation returns the best state visited from any SA algorithm invocation, and the next call of SA begins from the best known state. Furthermore, since initial temperature is halved after each invocation, the algorithm becomes greedier during the optimization process. This process is iterated until a final temperature T_{final} has been reached. This enables both fine and coarse grain search of the state space with reasonable optimization cost. At each invocation it becomes harder for

```
OPTIMIZATION()
1  S ← FAST-PREMAPPING()
2  T ← 1.0
3  while T > T_final
4  do S ← SIMULATED-ANNEALING(S, T)
5     if UseGroupMigration = True
6       then S ← GROUP-MIGRATION(S)
7     T ← T/2
8  return S
```

Fig. 4. Pseudo-code of the main optimization loop

```
FAST-PREMAPPING()
 1  Assigned ← [False] * N_tasks
 2  S ← [NIL] * N_tasks
 3  S[ExitNode] ← 0
 4  F ← EmptyFifo
 5  FIFO-PUSH(F, ExitNode)
 6  while F ≠ EmptyFifo
 7  do n ← FIFO-PULL(F)
 8     A ← S[node]
 9     for each parent p of node n
10     do if Assigned[p] = True
11           then continue
12        Assigned[p] = True
13        S[p] = A
14        FIFO-PUSH(F, p)
15        A ← (A + 1) mod N_agents
16  return S
```

Fig. 5. Pseudo-code of the fast premapping algorithm

the algorithm to make drastic jumps in the state space.

B. Scheduling

This paper considers node weights in task graphs as execution time to perform computation on a processing element and edge weights as time to transfer data between PEs in a SoC communication network. The scheduler used in this system is a B-level scheduler [1]. Since nodes are mapped before scheduling, B-level priorities associated with the nodes remain constant during the scheduling. This approach accelerates optimization, because B-level priorities need not to be recalculated. There exists better scheduling algorithms in the sense that they produce shorter schedules, but they are more expensive in optimization time [7]. When scheduling is finished the total execution time of the STG is known, and thus the cost of the mapping can be evaluated.

IV. RESULTS

A. Case Study Arrangements

The case study experiment used 10 random graphs, each having 50 nodes, from the Standard Task Graph set [8]. We used random graphs to evaluate optimization algorithms as

fairly as possible. Non-random applications may well be relevant for common applications, but they are dangerously biased for mapping algorithm comparison. Investigating algorithm bias and classifying computational tasks based on bias are not topics in this paper. Random graphs have the property to be neutral of the application, and thus mapping algorithms that do well on random graphs will do well on average applications.

Optimization was run 10 times independently for each task graph. Thus 100 optimization runs were executed for both algorithms. The objective function for optimization was the execution time of a mapped task graph. The random graphs did not have exploitable parallelism for more than 4 PEs so that was chosen as the maximum number of PEs for the experiment. Thus speedup of mapped task graphs was obtained for 2, 3, and 4 PEs.

The SoC was a message passing system where each PE had some local memory, but no shared memory. The PEs were interconnected with a single dynamically arbitrated shared bus that limits the SoC performance because of bus contention.

The optimizations were executed on a 10 machine Gentoo Linux cluster. Optimization runs were distributed by using *rsync* to copy input and output files between machines and *SSH* to execute shell scripts remotely. Initially one machine uploaded input files and optimization software to all machines, and then commanded each machine to start optimization software. After optimization was finished on all machines, the results were downloaded back. All machines were 2.8 GHz Intel Pentium 4 machines with 1 GiB of memory. Execution time for optimization is given in Section IV-C.

The optimization system was written with Python language. It is an object-oriented dynamically typed language that is interpreted during execution. Python language was chosen to save development time with object-oriented high-level techniques. The optimization system code is highly modular in object-oriented fashion.

The implementation of simulated annealing algorithm has the following specific issues. **Temperature–Cooling** function shown in Fig. 2. computes a new temperature with formula $T_0 * p^i$, where p is the percentage of temperature preserved on each iteration. **Move** function makes a random state change so that $N_{tasks}T_0$ random tasks are moved separately to a random PE, where N_{tasks} is the number of tasks in the task graph. **Prob** function is

$$\mathbf{Prob}(\Delta C, T) = \frac{1}{1 + exp(\frac{\Delta C}{0.5 C_0 T})},$$

where $C_0 = \mathbf{Cost}(S_0)$. The C_0 term is a special normalization factor chosen to make annealing parameters more compatible with different graphs.

B. SA and GM Iterations

Fig. 6 shows the progress of annealing without the group migration algorithm for a random 50-node STG. The cost function is the execution time for executing the mapped graph on two processing elements. This figure shows how each invocation of SA improves the overall solution, but improving

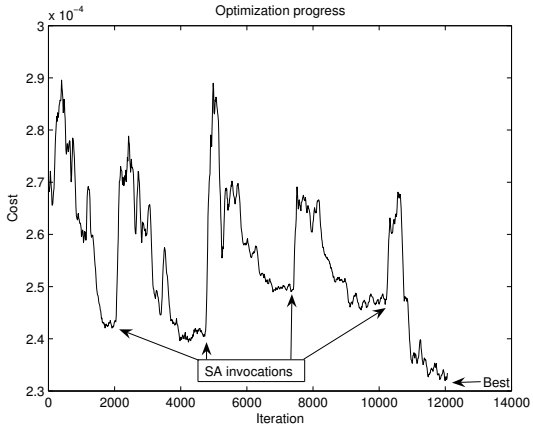


Fig. 6. Simulated annealing of a 50 node random STG. Cost function is the execution time for executing the mapped graph on two processing elements

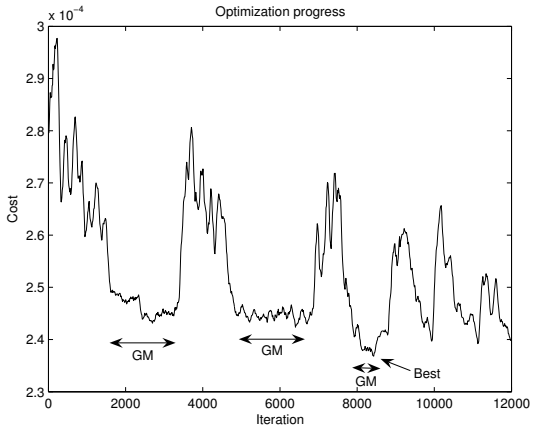


Fig. 7. Simulated annealing combined with group migration algorithm of a 50 node random STG

is decelerated between each call. The first downfall is the largest, and after that downfalls are less drastic.

Second, SA and GM are used to complement each other. SA can climb from a local minima to reach a global minimum. SA does not search for local minima, so it does not exploit local similarity to reach better states by exploring local neighborhoods. The GM algorithm is used to locally optimize SA solutions as far as possible. Fig. 7 shows progress of combined SA and GM optimization. At around iteration 2250 the GM algorithm is able find a locally better solution and thus the objective function value decreases. SA might not have found that solution. At around iteration 8300 GM finds the best solution by local optimization.

C. Hybrid Algorithm

The hybrid algorithm was compared with pure simulated annealing by obtaining speedups for parallelized task graphs. Speedup is defined as t_o/t_p, where t_o is the original execution time of the mapped graph and t_p is the parallelized execution time.

TABLE I

AVERAGED SPEEDUPS AND NUMBER OF COST FUNCTION EVALUATIONS FOR ALGORITHMS

	2 PEs	3 PEs	4 PEs
SA speedup	1.467	1.901	2.103
Hybrid speedup	1.488	1.977	2.278
Difference-%	1.4	4.0	8.3
SA cost evaluations	1298k	857k	767k
Hybrid cost evaluations	1313k	1047k	1322k
Evaluations difference-%	3.6	14.9	34.5

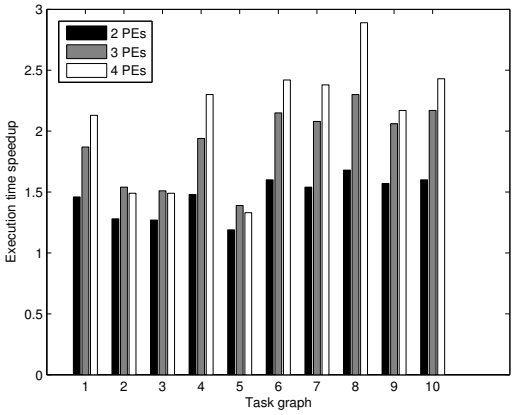


Fig. 8. Per task graph speedups for SA algorithm with 2, 3, and 4 processing elements

The results are shown in Table I. The advantage of the hybrid algorithm increases as the number of PEs increases. With 2 PEs, the benefit is small, but with 3 PEs it is 4.0% better speedup on average for 10 graphs with 10 independent runs. With 4 PEs the benefit is 8.3% better speedup (0.175 speedup units). However, greater speedup is achieved with the cost of optimization time.

Fig. 8 shows speedup values for each graph when SA algorithm was used for optimization. There is 10 bar sets in the figure, and each bar set presents 3 values for 2, 3, and 4 processing elements respectively. All values are averaged over 10 independent runs. Fig. 9 shows same values for the hybrid algorithm.

Optimization time is determined by the number of cost function evaluations as tabulated in Table I. The hybrid algorithm has 3.6%, 14.9%, and 34.5% more cost function evaluations for 2, 3, and 4 PEs respectively. Total running time for optimizations was 40036 seconds for SA, and 62007 seconds for the hybrid algorithm. Thus hybrid algorithm ran 55% longer in wall time. Compared to other proposals [2], the average improvement of 8% is considered very good.

V. CONCLUSION

The new method applies both local and global optimization techniques to speedup execution time of static task graphs. The new method is a hybrid algorithm that combines simulated annealing and group migration algorithms in a novel fashion.

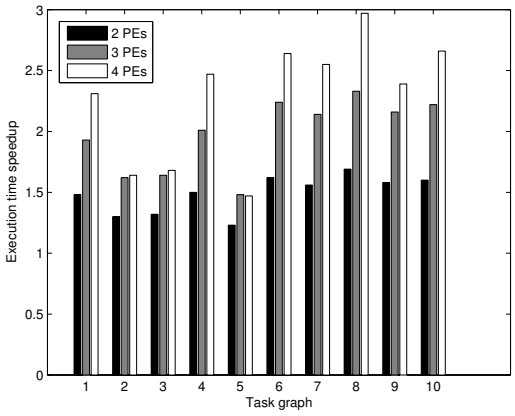


Fig. 9. Per task graph speedups for the hybrid algorithm with 2, 3, and 4 processing elements

The algorithm takes advantage of both greedy and non-greedy optimization techniques.

Pure simulated annealing and the hybrid algorithm were compared. The results show 8.3% speedup increase for the hybrid algorithm with 4 PEs averaged over 100 test runs with the expense of 34.5% iteration rounds.

Further research is needed to investigate simulated annealing heuristics that explore new states in the mapping space. A good choice of mapping heuristics can improve solutions and accelerate convergence, and it is easily applied into the existing system. Further research should also investigate how this method applies to optimizing other factors in a SoC. Execution time is one factor, and memory and power consumption are others.

REFERENCES

[1] Y.-K. Kwok and I. Ahmad, *Static scheduling algorithms for allocating directed task graphs to multiprocessors*, ACM Comput. Surv., Vol. 31, No. 4, pp. 406-471, 1999.

[2] T. Wild, W. Brunnbauer, J. Foag, and N. Pazos, *Mapping and scheduling for architecture exploration of networking SoCs*, Proc. 16th Int. Conference on VLSI Design, 2003.

[3] F. Vahid, *Partitioning sequential programs for CAD using a three-step approach*, ACM Transactions on Design Automation of Electronic Systems, Vol. 7, No. 3, pp. 413-429, 2002.

[4] T. Lei and 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), 2003.

[5] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, *Optimization by simulated annealing*, Science, Vol. 200, No. 4598, pp. 671-680, 1983.

[6] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems*, Pearson Education, 1994.

[7] I. Ahmad and Y.-K. Kwok, *Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors*, Parallel Architectures, Algorithms, and Networks, 1996.

[8] *Standard task graph set*, http://www.kasahara.elec.waseda.ac.jp/schedule, 2003.

PUBLICATION 2

Parameterizing Simulated Annealing for Distributing Task Graphs on Multiprocessor SoCs

Heikki Orsila, Tero Kangas, Erno Salminen and Timo D. Hämäläinen
Institute of Digital and Computer Systems
Tampere University of Technology
P.O. Box 553, 33101 Tampere, Finland
Email: {heikki.orsila, tero.kangas, erno.salminen, timo.d.hamalainen}@tut.fi

Abstract— **Mapping an application on Multiprocessor System-on-Chip (MPSoC) is a crucial step in architecture exploration. The problem is to minimize optimization effort and application execution time. Simulated annealing is a versatile algorithm for hard optimization problems, such as task distribution on MPSoCs. We propose a new method of automatically selecting parameters for a modified simulated annealing algorithm to save optimization effort. The method determines a proper annealing schedule and transition probabilities for simulated annealing, which makes the algorithm scalable with respect to application and platform size. Applications are modeled as static acyclic task graphs which are mapped to an MPSoC. The parameter selection method is validated by extensive simulations with 50 and 300 node graphs from the Standard Graph Set.**

I. Introduction

Efficient MPSoC implementation requires exploration to find an optimal architecture as well as mapping and scheduling of the application on the architecture. The large design space must be pruned systematically, since the exploration of the whole design space is not feasible. This, in turn, calls for optimization algorithms in which the cost function consists of execution time, communication time, memory, energy consumption and silicon area constraints, for example. The iterative algorithms evaluate a number of application mappings for each resource allocation candidate. For each mapping, an application schedule is determined to evaluate the cost.

This paper presents a new method to automatically select parameters for the simulated annealing (SA) algorithm [1]. Parameter selection is needed because SA is a meta-algorithm that doesn't specify all the necessary details. Our algorithm selects annealing schedule and transition probabilities to maximize application performance and minimize optimization time. The algorithm is targeted to map and schedule applications for MPSoCs. However, the algorithm is not limited to MPSoC architectures or performance optimization.

The SoC applications are modeled as acyclic static task graphs (STGs) in this paper. Parallelizing STGs to speedup the execution time is a well researched subject [2], but MPSoC architectures present more requirements, such as application execution time estimation for architecture exploration, compared to traditional multiprocessor systems. Nodes of the STG are finite deterministic computational tasks, and edges represent dependencies between the nodes. Computational nodes block until their data dependencies are resolved, i.e. they have all needed data. Node weights represent the amount of computation associated with a node. Edge weights represent amount of communication needed to transfer results between the nodes. The details of task graph parallelization for an MPSoC can be found, for example, in [3]. SA is used to place all tasks onto specific processing elements (PEs) to parallelize execution. Alternative solutions for the problen can be found, for example, in [2].

The basic concepts and the related work of task parallelization with SA is presented in Section II. The contribution of this paper is adaptation and parametrization of SA for task mapping, which is described in Section III. The algorithm was evaluated with a set of task graphs and compared to the pure SA algorithm as reported in Section IV. Finally, the concluding remarks are given.

II. Related Work

A. Algorithms for Task Mapping

Architecture exploration needs automatic tuning of optimization parameters for architectures of various sizes. Without scaling, algorithm may spend excessive time optimizing a small systems or result in a sub-optimal solution for a large system. Wild *et al.* [4] compared SA, Tabu Search (TS) [5] and various other algorithms for task distribution. The parameter selection for SA had geometric annealing schedule that did not consider application or system architecture size, and thus did not scale up to bigger problems without manual tuning of parameters.

Braun *et al.* [6] compared 11 optimization algorithms for task distribution. TS outperformed SA in [4], but was worse in [6], which can be attributed to different parameter selection used. Braun's method has a proper initial temperature selection for SA to normalize transition probabilities, but their annealing schedule does not scale up with application or system size, making both [4] and [6] unsuitable for architecture exporation.

Our work presents a deterministic method for deciding efficient annealing schedule and transition probabilities to minimize iterations needed for SA, and, hence, allows efficient architecture exploration also to large systems. The method determines proper initial and final temperatures and the number of necessary iterations per temperature level to avoid unnecessary optimization iterations, while keeping application

1-4244-0622-6/06/$20.00

performance close to maximum. This will save optimization time and thus speed up architecture exploration.

B. Simulated Annealing

SA is a probabilistic non-greedy algorithm [1] that explores search space of a problem by annealing from a high to a low temperature state. The algorithm always accepts a move into a better state, but also into a worse state with a changing probability. This probability decreases along with the temperature, and thus the algorithm becomes greedier. The algorithm terminates when the final temperature is reached and sufficient number of consecutive moves have been rejected.

Fig. 1 shows the pseudo-code of the SA algorithm used with the new method for parameter selection. Implementation specific issues compared to the original algorithm are explained in Section III. The $Cost$ function evaluates execution time of a specific mapping by calling the scheduler. S_0 is the initial mapping of the system, T_0 is the initial temperature, and S and T are current mapping and temperature, respectively. **New_Temperature_Cooling** function, a contribution of this paper, computes a new temperature as a function of initial temperature T_0 and iteration i. R is the number of consecutive rejects. $Move$ function moves a random task to a random PE, different than the original PE. $Random$ function returns an uniform random value from the interval $[0, 1)$. **New_Prob** function, a contribution of this paper, computes a probability for accepting a move that increases the cost. R_{max} is the maximum number of consecutive rejections allowed after the final temperature has been reached.

```
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 ← 0 to ∞
 7  do T ← NEW_TEMPERATURE_COOLING(T_0, i)
 8     S_new ← MOVE(S, T)
 9     C_new ← COST(S_new)
10     ΔC ← C_new − C
11     r ← RANDOM()
12     p ← NEW_PROB(ΔC, T)
13     if ΔC < 0 or r < p
14        then if C_new < C_best
15                then S_best ← S_new
16                     C_best ← C_new
17             S ← S_new
18             C ← C_new
19             R ← 0
20        else if T ≤ T_f
21                then R ← R + 1
22                     if R ≥ R_max
23                        then break
24  return S_best
```

Fig. 1. Pseudo-code of the simulated annealing algorithm

III. The Parameter Selection Method

The parameter selection method configures the annealing schedule and acceptance probability functions.

$New\_Temperature\_Cooling$ function is chosen so that annealing schedule length is proportional to application and system architecture size. Moreover the initial temperature T_0 and final temperature T_f must be in the relevant range to affect acceptance probabilities efficiently. The method uses

$$New\_Temperature\_Cooling(T_0, i) = T_0 * q^{\lfloor \frac{i}{L} \rfloor},$$

where L is the number of mapping iterations per temperature level and q is the proportion of temperature preserved after each temperature level. This paper uses $q = 0.95$. Determining proper L value is important to anneal more iterations for larger applications and systems. This method uses

$$L = N(M - 1),$$

where N is the number of tasks and M is the number of processors in the system. Also, $R_{max} = L$.

A traditionally used acceptance function is

$$Trad\_Prob(\Delta C, T) = \frac{1}{1 + exp(\frac{\Delta C}{T})},$$

but then probability range for accepting moves is not adjusted to a given task graphs because ΔC is not normalized. The acceptance probability function used in this method has a normalization factor to consider only relative cost function changes. Relative cost function change adapts automatically to different cost function value ranges and graphs with different task execution times.

$$New\_Prob(\Delta C, T) = \frac{1}{1 + exp(\frac{\Delta C}{0.5 C_0 T})},$$

where $C_0 = Cost(S_0)$, the initial cost of the optimized system. Figure 2 shows relative acceptance probabilities.

The initial temperature chosen by the method is

$$T_0^P = \frac{k t_{max}}{t_{minsum}},$$

where t_{max} is the maximum execution time for any task on any processor, t_{minsum} the sum of execution times for all tasks on the fastest processor in the system, and $k \geq 1$ is a constant. Constant k, which should practically be less than 10, gives a temperature margin for safety. Section IV-A will show that $k = 1$ is sufficient in our experiment. The rationale is choosing an initial temperature where the biggest single task will have a fair transition probability of being moved from one PE to another. The transition probabilities with respect to temperature and $\Delta C_r = \frac{\Delta C}{C_0}$ can be seen in Figure 2. Section IV will show that efficient annealing happens in the temperature range predicted by the method. The chosen final temperature is

$$T_f^P = \frac{t_{min}}{k t_{maxsum}},$$

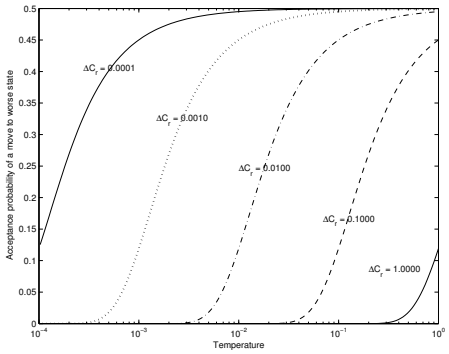


Fig. 2. Probabilities for the normalized probability function: $\Delta C_r = \frac{\Delta C}{C_0}$

where t_{min} is the minimum execution time for any task on any processor and t_{maxsum} the sum of execution times for all tasks on the slowest processor in the system. Choosing initial and final temperature properly will save optimization iterations. On too big a temperature, the optimization is practically *Monte Carlo* optimization because it accepts moves to worse positions with a high probability. And thus, it will converge very slowly to optimum because the search space size is in $O(M^N)$. 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.

IV. Results

A. Experiment

The experiment uses 10 random graphs with 50 nodes and 10 random graphs with 300 nodes from the Standard Task Graph set [7] to validate that the parameter selection method chooses good acceptance probabilities (New_Prob) and annealing schedule (T_0^P, T_f^P, and L). Random graphs are used to evaluate optimization algorithms as fairly as possible. Non-random applications may well be relevant for common applications, but they are dangerously biased for general parameter estimation. Investigating algorithm bias and classifying computational tasks based on the bias are not topics of this paper. Random graphs have the property to be neutral of the application.

Optimization was run 10 times independently for each task graph. Each graph was distributed onto 2-8 PEs. Each anneal was run from a high temperature $T_0 = 1.0$ to a low temperature $T_f = 10^{-4}$ with 13 different L values. The experiment will show that $[T_f, T_0]$ is a wide enough temperature range for optimization and that $[T_f^P, T_0^P]$ is a proper subset of $[T_f, T_0]$ which will yield equally good results in a smaller optimization time. L values are powers of 2 to test a wide range of suitable parameters. All results were averaged for statistical reliability. The optimization parameters of the experiment are shown in Table I.

The SoC platform was a message passing system where each PE had some local memory, but no shared memory. The PEs were interconnected with a single dynamically arbitrated shared bus that limits the SoC performance because of bus contention. The optimization software was written in C language and executed on a 10 machine Gentoo Linux cluster each machine having a 2.8 GHz Intel Pentium 4 processor and 1 GiB of memory. A total of $2.03G$ mappings were tried in 909 computation hours leading to average $620\frac{mappings}{s}$.

TABLE I
Optimization parameters for the experiment

Parameter	Value	Meaning
L	$1, 2, 4, \ldots, 4096$	Iterations per temperature level
T_0	1	Initial temperature
T_f	10^{-4}	Final temperature
M	2 - 8	Number of processors
N	50, 300	Number of tasks

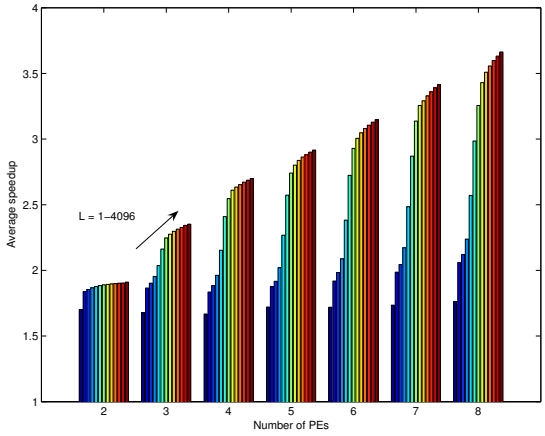


Fig. 3. Averaged speedups for 300 node graphs with M=2-8 processing elements and different L values ($L = 1, 2, \ldots, 4096$) for each processing element set.

B. Experimental Results

Figure 3 shows averaged speedups for 300-node task graphs with respect to number of iterations per temperature level and number of PEs. Speedup is defined as $\frac{t_1}{t_M}$, where t_i is the graph execution time on i PEs. The bars show that selecting $L = N(M-1)$, where $N = 300$ is the number of tasks and $M \in [2, 8]$ gives sufficient iterations per temperature level to achieve near best speedup (over 90% in this experiment) when the reference speedup is $L = 4096$. The Figure also shows that higher number of PEs requires higher L value which is logically consistent with the fact that higher number of PEs means to a bigger optimization space.

Figure 4 shows average speedup with respect to temperature. Average execution time proportion for a single task in a 300 node graph is $\frac{1}{300} = 0.0033$. With our normalized acceptance probability function this also means the interesting temperature value for annealing, $T = 0.0033$, falls safely within the predicted temperature range $[T_f^P, T_0^P] = [0.0004, 0.0074]$ computed with $k = 1$ from 300 node graphs. The Figure shows that optimization progress is fastest at that range. The full range $[T_f, T_0]$ was annealed to show convergence outside

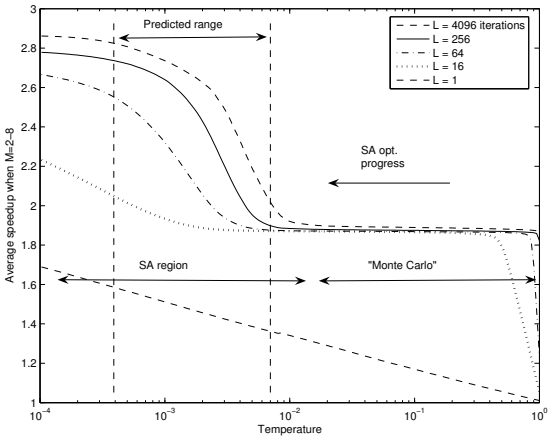


Fig. 4. Averaged speedup with respect to temperature for 300 node graphs with different L values.

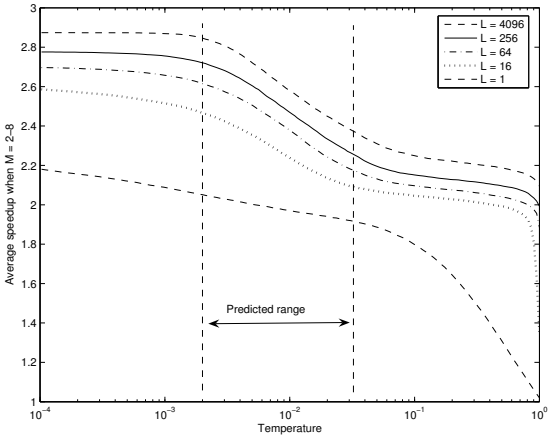


Fig. 5. Averaged speedup with respect to temperature for 50 node graphs with different L values.

the predicted range. The method also applies well for the 50 node graphs, as shown in Figure 5, where the interesting temperature point is at $T = \frac{1}{50} = 0.02$. The predicted range computed for 50 node graphs with $k = 1$ is $[0.0023, 0.033]$ and the Figure shows that steepest optimization progress falls within that range.

Annealing the temperature range $[10^{-2}, 1]$ with 300 nodes in Figure 4 is avoided by using the parameter selection method. That range is essentially Monte Carlo optimization which converges very slowly and is therefore unnecessary. The temperature scale is exponential and therefore that range consists of half the total temperature levels in the range $[T_f, T_0]$. This means approximately 50% of optimization time can be saved by using the parameter selection method. The main benefit of this method is determining an efficient annealing schedule.

The average speedups with respect to number of mapping evaluations with different L values is shown in Figure 6. Optimization time is a linear function of evaluated mappings.

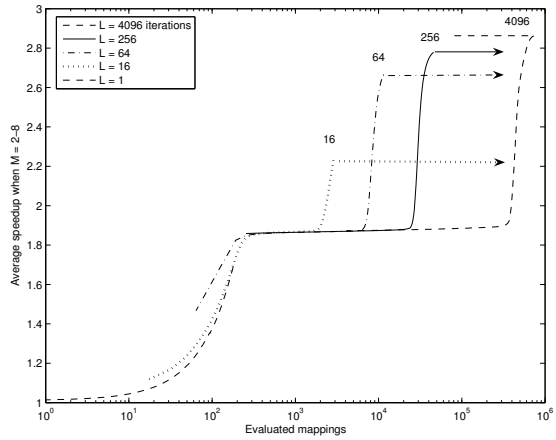


Fig. 6. Averaged speedup with respect to mapping evaluations for 300 node graphs with different L values.

This Figure also strenghtens the hypothesis that $L = N(M - 1) = 300, \ldots, 2100$ is sufficient number of iterations per temperature level.

V. Conclusion

The new parameter selection method was able to predict an efficient annealing schedule for simulated annealing to both maximize application execution performance and also minimize optimization time. Near maximum performance was achieved by selecting the temperature range and setting the number of iterations per temperature level automatically based on application and platform size. The number of temperature levels was halved by the method. Thus the method increased accuracy of architecture exploration and accelerated it.

Further research is needed to investigate simulated annealing heuristics that explore new states in the mapping space. A good choice of mapping heuristics can improve solutions and accelerate convergence, and it is easily applied into the existing system. Further research should also investigate how this method applies to optimizing memory, power consumption, and other factors in a SoC.

References

[1] S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi, *Optimization by simulated annealing*, Science, Vol. 200, No. 4598, pp. 671-680, 1983.

[2] Y.-K. Kwok and I. Ahmad, *Static scheduling algorithms for allocating directed task graphs to multiprocessors*, ACM Comput. Surv., Vol. 31, No. 4, pp. 406-471, 1999.

[3] 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), pp. 146-150, 2005.

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

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

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

[7] *Standard task graph set*, [online]: http://www.kasahara.elec.waseda.ac.jp/schedule, 2003.

PUBLICATION 3

H. Orsila, T. Kangas, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, *Automated Memory-Aware Application Distribution for Multi-Processor System-On-Chips*, Journal of Systems Architecture, Volume 53, Issue 11, ISSN 1383-7621, pp. 795-815, 2007.

Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 53 (2007) 795–815
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

Automated memory-aware application distribution for Multi-processor System-on-Chips

Heikki Orsila [a,*], Tero Kangas [b], Erno Salminen [a], Timo D. Hämäläinen [a], Marko Hännikäinen [a]

[a] Tampere University of Technology, Institute of Digital and Computer Systems, PO Box 553, FIN-33101 Tampere, Finland
[b] Nokia Technology Platforms, Visiokatu 6, 33720 Tampere, Finland

Received 22 December 2005; received in revised form 13 November 2006; accepted 15 January 2007
Available online 14 February 2007

Abstract

Mapping of applications on a Multi-processor 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; Multi-processor

1. Introduction

The problem being solved is increasing performance and decreasing energy consumption of Multi-processor 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

* Corresponding author. Tel.: +35 8407325989.
E-mail address: heikki.orsila@tut.fi (H. Orsila).

1383-7621/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2007.01.013

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 solutions. 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.

Fig. 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

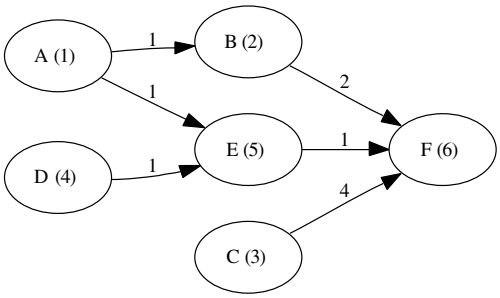


Fig. 1. Example of a static task graph. Values in parenthesis inside the nodes represent execution time, and values at edges represent communication sizes. The communication time will be determined by the interconnect, its availability and data size.

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 cannot 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.

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 communica-

tion 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

Our MP-SoC application distribution framework is shown in Fig. 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

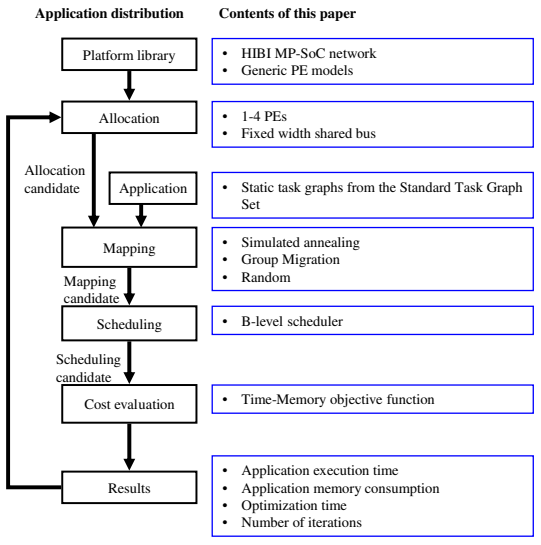


Fig. 2. MP-SoC application distribution framework.

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 Fig. 3, and explanations of symbols are given in Table 1. SA cannot 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 Fig. 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
```

Fig. 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 $r = \log\left(\frac{T_0}{T_f}\right)$ 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

$$L = N(M-1) \tag{1}$$

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 Eq. (2) depends on the proportion p and the initial and final temperatures as

$$T_0 p^x = T_f \Rightarrow x = \frac{\log\frac{T_f}{T_0}}{\log p}. \tag{2}$$

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

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

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

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

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

A common choice for acceptance probability is shown in Eq. (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 Eq. (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].

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

Fig. 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.

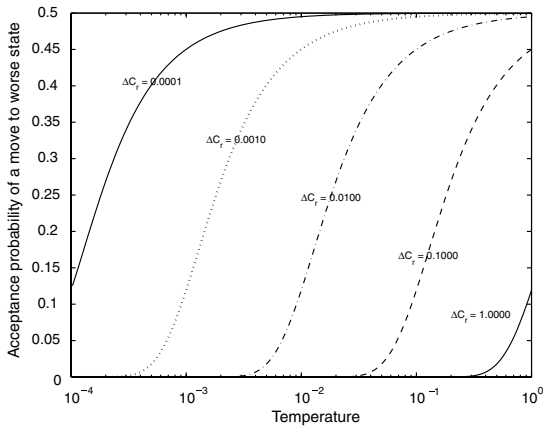


Fig. 4. Acceptance probabilities for the normalized probability function with respect to relative cost function change $\Delta C_r = \frac{\Delta C}{C_0}$.

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

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

```
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
```

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

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 Fig. 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.

The obvious deficiency in the RN heuristics is fixed by our RM move (Random Modifying move) heuristics presented in Fig. 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.

Fig. 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.

5.3. Group migration algorithm

Group migration is used to map task graphs onto multiple PEs in a greedy fashion. This is a general-

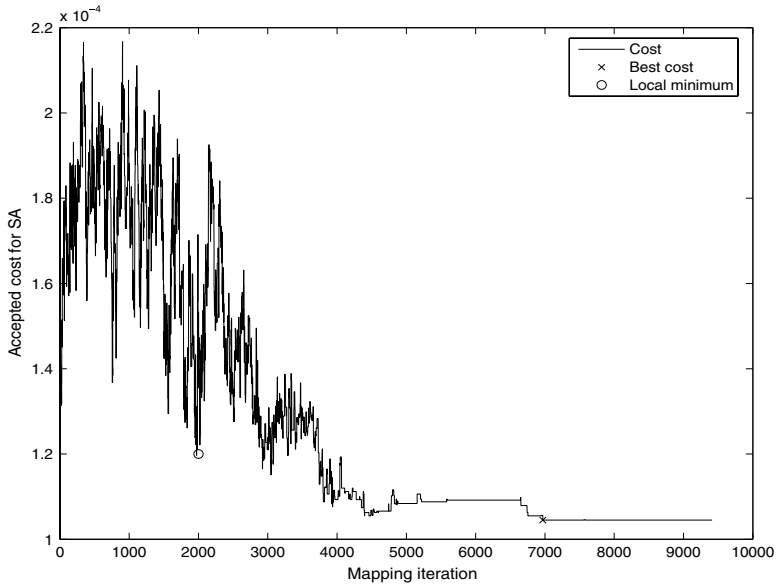


Fig. 7. An example of annealing from a high to a low temperature state for a 50 node STG with 2 PEs. × marks the mapping iteration that has the lowest cost function value.

ization 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 Fig. 8 and variables that are new compared to SA are explained in Table 2.

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

```
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
```

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

Table 2
Summary of new variables used in the group migration algorithm

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

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 Fig. 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.

Fig. 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

```
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
```

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

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.

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 3600 lines of C code. On a 2.0 GHz AMD Athlon 64 computer, the software is able to evaluate 12,000 mappings in a second for a 50 node STG with 119 edges, or 600,000 nodes or 1.43 million edges in a second. The software did 2094 mappings in a second for a 336 node STG with 1211 edges, or 703,584 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. Fig. 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.

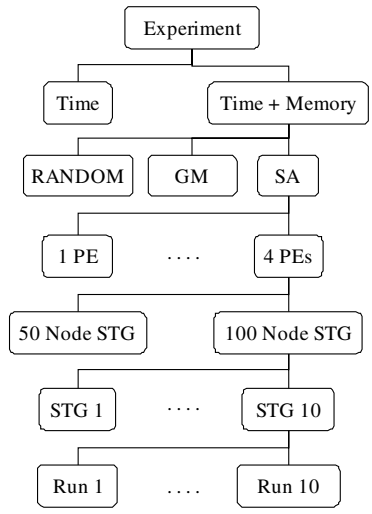


Fig. 10. Outline of the experiment procedure.

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 Fig. 4.

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

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.

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
Total number of mappings for one simulated annealing descend

	2 PEs	3 PEs	4 PEs
52 tasks	9300	18,700	28,000
102 tasks	18,300	36,600	54,900

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.

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 Fig. 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.

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

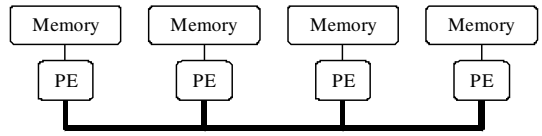


Fig. 11. Block diagram of the MP-SoC execution platform.

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.

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 cannot 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.

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

Table 6
MP-SoC execution platform data

PEs	1–4
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

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

$$t = \frac{\mathrm{Lat} + \left\lceil \frac{8\mathrm{Si}}{W} \right\rceil}{f}, \tag{7}$$

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.

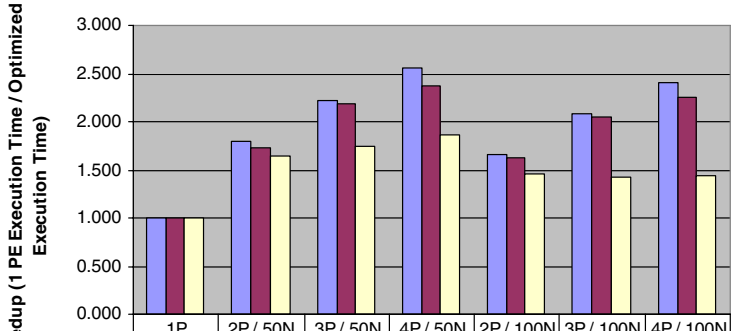
CCR is computed by dividing the average edge weight with the average node weight of the graph. CCR values on 50 and 100 node graphs for 2, 3 and 4 PE cases are 0.98, 0.65 and 0.49, respectively. As the rationale in related work section explains, values near 1.0 are in the relevant range of parallel computing. 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 cannot 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.

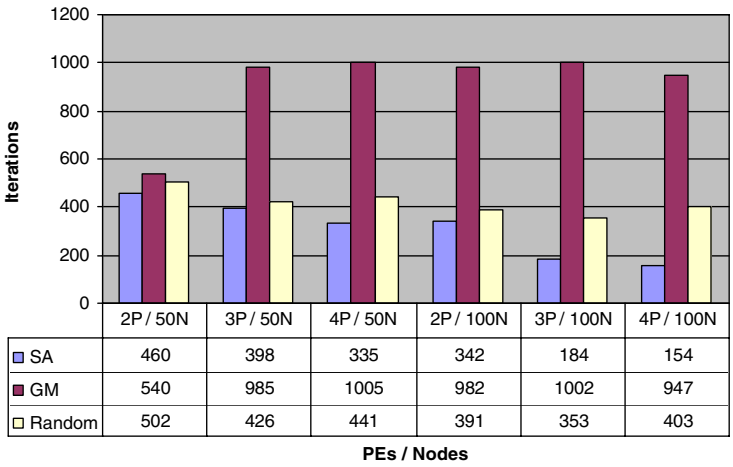
8.1. Time optimization comparison

Fig. 12 presents the speedup values for each algorithm in the time optimization case with 2–4 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 Fig. 13. 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.



	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

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



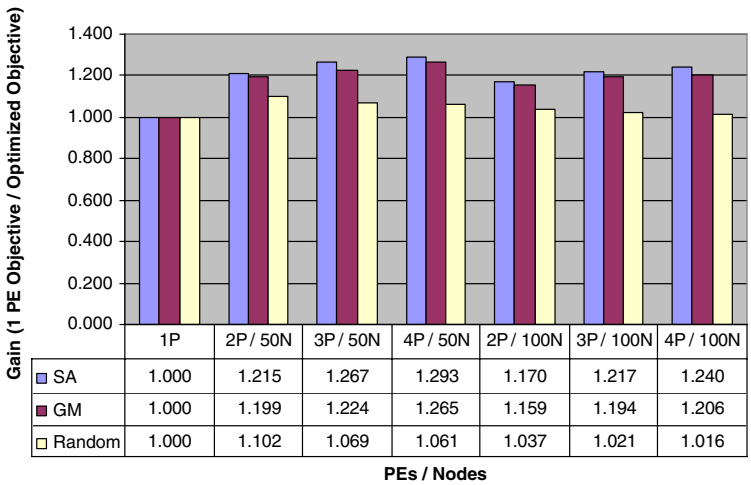
	2P / 50N	3P / 50N	4P / 50N	2P / 100N	3P / 100N	4P / 100N
SA	460	398	335	342	184	154
GM	540	985	1005	982	1002	947
Random	502	426	441	391	353	403

Fig. 13. Mean iterations to reach best solution on time optimization for 50 and 100 node graphs.

8.2. Memory-time optimization comparison

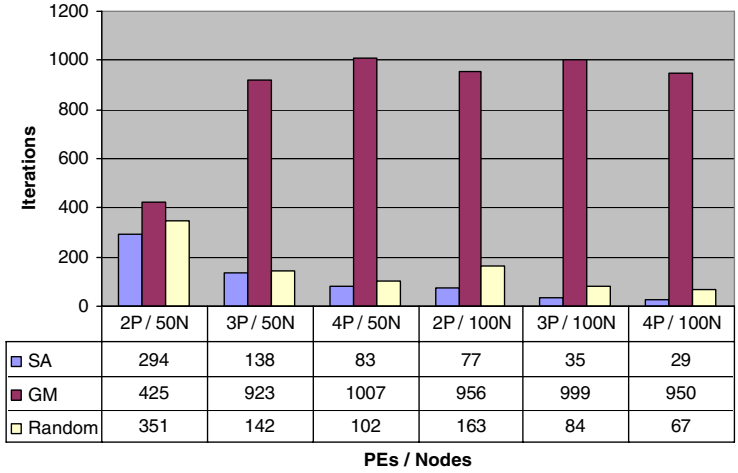
Fig. 14 shows the gain values for the memory-time optimization case with 2–4 PEs for 50 and 100 node task graphs. Average gains in combined 50 and 100 node cases for SA, GM and random mapping are 1.234, 1.208 and 1.051, respectively. Thus, SA wins GM by 2.2% and random by 17%. SA reaches the best solution in 12% of iterations compared to GM, which is significantly faster, as shown in Fig. 15. The memory-gain units are small as numeric values, and their meaning must be analyzed separately in terms of speedup and memory usage.

Fig. 16 shows the memory gain values for the memory-time optimization case, and Fig. 17 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



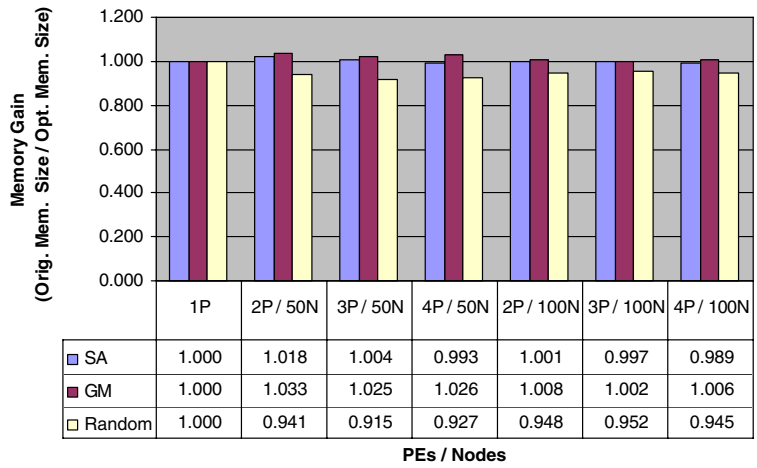
	1P	2P / 50N	3P / 50N	4P / 50N	2P / 100N	3P / 100N	4P / 100N
SA	1.000	1.215	1.267	1.293	1.170	1.217	1.240
GM	1.000	1.199	1.224	1.265	1.159	1.194	1.206
Random	1.000	1.102	1.069	1.061	1.037	1.021	1.016

Fig. 14. Mean 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	294	138	83	77	35	29
GM	425	923	1007	956	999	950
Random	351	142	102	163	84	67

Fig. 15. Mean iterations to reach best solution on memory and time optimization for 50 and 100 node graphs.



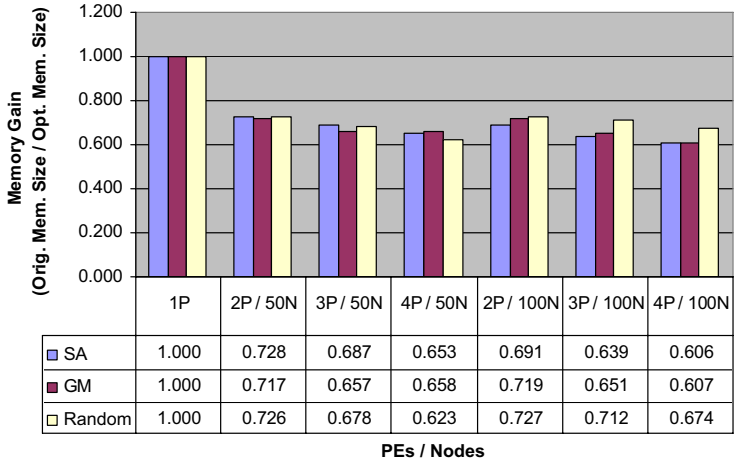
	1P	2P / 50N	3P / 50N	4P / 50N	2P / 100N	3P / 100N	4P / 100N
SA	1.000	1.018	1.004	0.993	1.001	0.997	0.989
GM	1.000	1.033	1.025	1.026	1.008	1.002	1.006
Random	1.000	0.941	0.915	0.927	0.948	0.952	0.945

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

decrease throughput and increase latency, which should be avoided.

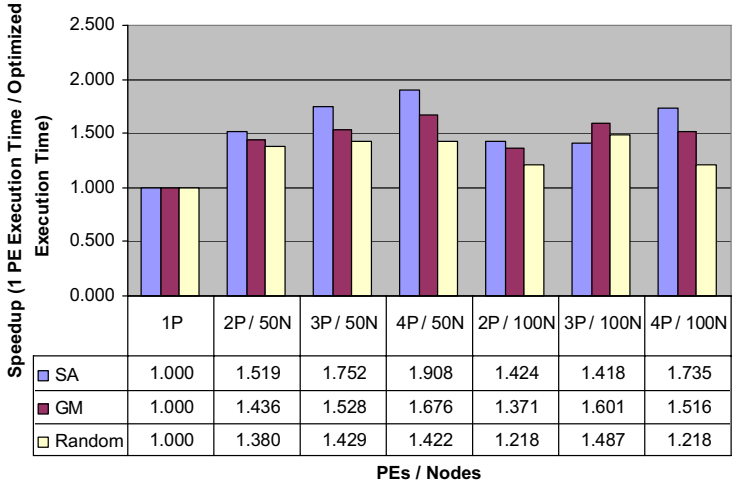
The GM case is interesting because it shows that computation actually uses 2% less memory for data buffers than a single PE. This happens because it is possible to off-load temporary results away from originating PE and then free buffers for other computation. Now comparing memory gain values to the 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. Fig. 18 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 16×16 pixel image blocks



	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

Fig. 17. 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	1.519	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

Fig. 18. Mean speedups on memory and time optimization for 50 and 100 node graphs.

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.

9. Parameter selection

To analyze the effect of Eq. (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 2). 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.

Figs. 19 and 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.

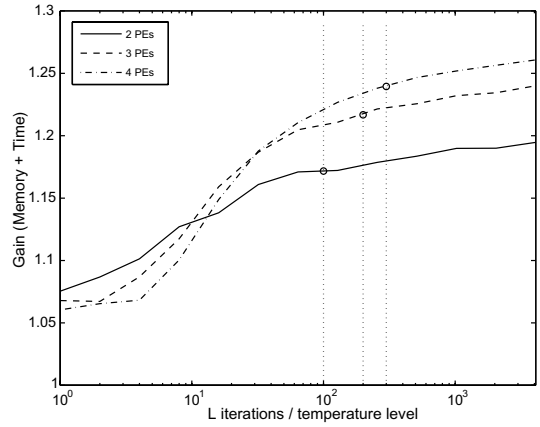


Fig. 20. Effect of L parameter (log scale) for memory and time optimizing 100 node graphs. The circles mark $L = N(M-1)$ case for each PE number.

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 Fig. 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 satu-

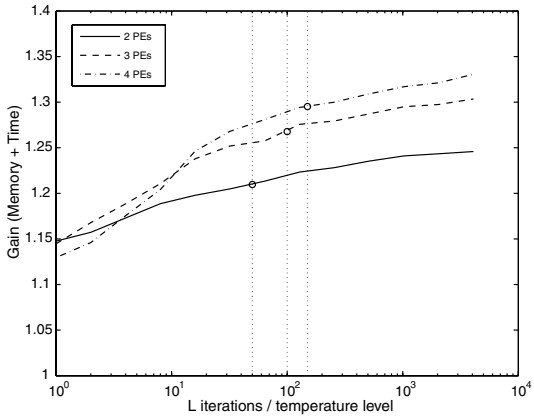


Fig. 19. Effect of L parameter (log scale) for memory and time optimizing 50 node graphs. The circles mark $L = N(M-1)$ case for each PE number.

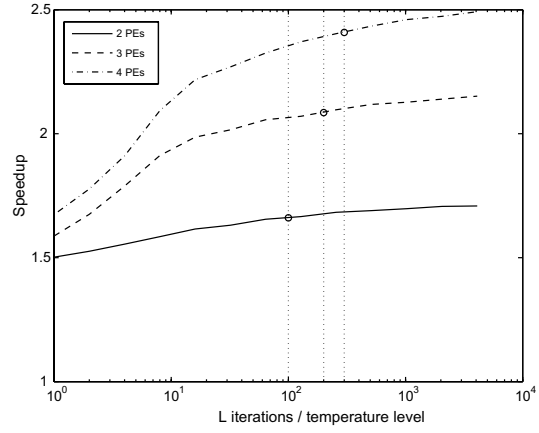


Fig. 21. Effect of L parameter (log scale) for time optimizing 100 node graphs. The circles mark $L = N(M-1)$ case for each PE number.

Table 7
Effect of L parameter on speedup and gain for time and memory + time optimization cases, respectively

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

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.

rated 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].

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.

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, in: Proceedings of the 16th International Conference on VLSI Design, 2003, pp. 376–381.
[3] O. Sinnen, L. Sousa, Communication contention in task scheduling, IEEE Transactions on Parallel and Distributed Systems 16 (6) (2005) 503–515.
[4] S. Kirkpatrick, C.D. Gelatt Jr., M.P. Vecchi, Optimization by simulated annealing, Science 200 (4598) (1983) 671–680.
[5] V. Cerny, Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm, Journal of Optical Theory and its Applications 45 (1) (1985) 41–51.
[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 15 (9) (2004) 824–834.

[7] T. Lei, S. Kumar, A two-step genetic algorithm for mapping task graphs to a network on chip architecture, in: Proceedings of the Euromicro Symposium on Digital System Design (DSD'03), 2003, pp. 180–187.
[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 (1379) (1991).
[9] F. Glover, E. Taillard, D. de Werra, A User's Guide to Tabu Search, Annals of Operations Research 21 (1993) 3–28.
[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 61 (2001) 810–837.
[11] D. Spinellis, Large production line optimization using simulated annealing, Journal of Production Research 38 (3) (2000) 509–541.
[12] B.W. Kernighan, S. Lin, An efficient heuristics procedure for partitioning graphs, The Bell System Technical Journal 49 (2) (1970) 291–307.
[13] Y.-K. Kwok, I. Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, Journal of Parallel and Distributed Computing 59 (3) (1999) 381–422.
[14] D. Gajski, F. Vahid, S. Narayan, J. Gong, Specification and Design of Embedded Systems, Prentice-Hall, Englewood Cliffs, NJ, 1994, ISBN 0131507311.
[15] H. Orsila, T. Kangas, T.D. Hämäläinen, Hybrid algorithm for mapping static task graphs on multiprocessor SoCs, in: International Symposium on System-on-Chip (SoC 2005), 2005.
[16] O. Sinnen, L. Sousa, Task scheduling: considering the processor involvement in communication, in: Proceedings of the 3 rd International Workshop on Algorithms, Models, and Tools for Parallel Computing on Heterogeneous Networks, ISPDC'04/HetroPar'04, 2004, pp. 328–335.
[17] R. Szymanek, K. Kuchcinski, A constructive algorithm for memory-aware task assignment and scheduling, in: International Conference on Hardware Software Codesign, Proceedings of the 9th International Symposium on Hardware/Software Codesign, 2001, pp. 147–152.
[18] Y.-K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Computing Survey 31 (4) (1999) 406–471.
[19] L. Ingber, Very fast simulated re-annealing, Mathematical and Computer Modelling 12 (8) (1989) 967–973.
[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, in: Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL2005), 2005, pp. 380–385.
[22] T. Bui, C. Heighman, C. Jones, T. Leighton, Improving the performance of the Kernighan-Lin and simulated annealing graph bisection algorithms, in: Proceedings of the 26th ACM/IEEE Conference on Design Automation, 1989, pp. 775–778.
[23] G. Ascia, V. Catania, M. Palesi, An evolutionary approach to network-on-chip mapping problem, in: The 2005 IEEE Congress on Evolutionary Computation, vol. 1, pp. 112–119.
[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 18 (1) (1999) 3–13.
[25] J. Hou, W. Wolf, Process partitioning for distributed embedded systems, in: Proceedings of the Fourth International Workshop on Hardware/Software Co-Design (Codes/CASHE '96), 1996, pp. 70–76.
[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, in press.
[27] H. Orsila, T. Kangas, E. Salminen, T.D. Hämäläinen, Parameterizing simulated annealing for distributing task graphs on multiprocessor SoCs, iin: International Symposium on System-on-Chip 2006, Tampere, Finland, November 14–16, 2006.

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 Ph.D. 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 Ph.D. 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.

PUBLICATION 4

Optimal Subset Mapping And Convergence Evaluation of Mapping Algorithms for Distributing Task Graphs on Multiprocessor SoC

Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen
Institute of Digital and Computer Systems
Tampere University of Technology
P.O. Box 553, 33101 Tampere, Finland
Email: {heikki.orsila, erno.salminen, marko.hannikainen, timo.d.hamalainen}@tut.fi

***Abstract*— Mapping an application on Multiprocessor System-on-Chip (MPSoC) is a crucial step in architecture exploration. The problem is to minimize optimization effort and application execution time. Applications are modeled as static acyclic task graphs which are mapped to an MPSoC. The analysis is based on extensive simulations with 300 node graphs from the Standard Graph Set.**

We present a new algorithm, Optimal Subset Mapping (OSM), that rapidly evaluates task distribution mapping space, and then compare it to simulated annealing (SA) and group migration (GM) algorithms. OSM was developed to make architecture exploration faster. Efficiency of OSM is $5.0\times$ and $2.4\times$ than that of GM and SA, respectively, when efficiency is measured as the application speedup divided by the number of iterations needed for optimization. This saves 81% and 62% in wall clock optimization time, respectively. However, this is a trade-off because OSM reaches 96% and 89% application speedup compared to GM and SA. Results show that OSM and GM have opposite convergence behavior and SA comes between these two.

I. Introduction

An efficient MPSoC implementation requires exploration to find an optimized architecture as well as mapping and scheduling of the application. A large design space must be pruned systematically, since the exploration of the whole design space is not feasible. However, fast optimization procedure is desired in order to cover reasonable design space. Iterative algorithms evaluate a number of application mappings for each resource allocation candidate. For each mapping, an application schedule is determined to evaluate the cost. The cost function may consider multiple parameters, such as execution time, communication time, memory, energy consumption and silicon area constraints.

SoC applications can be modeled as acyclic static task graphs (STGs) [1]. Nodes of the STG are finite, deterministic computational tasks, and edges denote dependencies between the nodes. Node weights represent the amount of computation associated with a node. Edge weights model the amount of communication needed to transfer results between the nodes. Computational nodes block until their data dependencies are resolved, i.e. when they have all needed data. Details of the task graph parallelization system employed in this paper can be found in [2].

This paper presents a new mapping algorithm called Optimal Subset Mapping (*OSM*) to speedup architecture exploration. It is compared to two variants of Simulated Annealing algorithm (*SA* and *SA+AT*) [2][3], Group Migration (*GM*), and their combination (*Hybrid*) [4]. Furthermore, a random algorithm is used as basis for comparison. The system has 2, 4, or 8 identical processing elements (PEs). Ten random 300-node STGs are selected from [5].

II. Related Work

Braun *et al.* [6] compared 11 optimization algorithms for distribution of tasks without data dependencies. 512 tasks were parallelized onto 16 machines and total execution time was measured for each heuristics. Our system schedules 300 tasks with data dependencies for 8 PEs. It must also be noted that our tasks have 10 times more edges than tasks in [6]. They do approximately $3\ 000$ mappings for each SA run, which can be shown to be far too few for 16 machines and 512 tasks in the case that tasks are dependent [2]. Our SA implementation does approximately $200\ 000$ mappings for a single run with dependent tasks. Their paper does not present convergence properties of SA as a function of iterations, and we are not aware of related work that measures SA convergence for task mapping with respect to iterations and the number of PEs.

Our earlier work [3] presented a heuristics for automatically selecting temperature schedule for SA to speedup convergence of dependent tasks. Also, [2] presented heuristics to select total iteration number for reasonable efficiency for SA with dependent tasks. This paper will merge those results and compare them to various mapping algorithms, including the new OSM.

III. Studied algorithms

The mapping algorithms can be classified as follows. First, is the algorithm deterministic (same results on every independent run) or probabilistic (result varies between runs). Second, does algorithm accept a move to worse state along the run (non-greedy) or only better moves (greedy). Hence, 4 categories can be identified.

Architecture exploration needs an automatic selection of optimization parameters depending on the architecture and application sizes. Otherwise, an algorithm may spend excessive time optimizing a small systems or result in a sub-optimal solution for a large system. The goal is to avoid unnecessary optimization iterations, while keeping application performance close to architecture limits. This will save optimization time and thus speed up architecture exploration.

The term *move* means here the change of the location (PE) of one or multiple tasks. All studied algorithms (except random) ensure that move is always made to a different PE, which saves many iterations. This is a crucial detail forgotten in many papers. For example, randomizing a single task for 2 PEs will result in 50% of iterations being useless because the task is not actually moved anywhere.

STGs are used because there exists well known efficient and near optimal scheduling algorithms for them. This ensures that the observed differences are due to mapping. Harder scheduling properties would diminish accuracy of mapping analysis. All presented algorithms are agnostic of STG structure, and so they will also work with general process networks like Kahn Process Networks (KPN) [7]. These algorithms are also used in our Koski flow, that has a KPN-like process network [8]. Koski is a high-level design tool for multiprocessor SoCs and applications.

Details of the used algorithms can be found in [2][3][4] but use of these algorithms is presented next. The new OSM algorithm is introduced in detail.

For each algorithm, tasks are initially mapped to one PE.

A. Group Migration (GM)

Group migration (GM), also known as Kernighan-Lin graph partitioning algorithm [9], is a deterministic algorithm that moves one task at time and finds an optimal mapping for that. It accepts only moves to a better state (one with smaller cost). Therefore, it is greedy algorithm and may get stuck to a local minimum. This happens when there is no single move that improves (decreases) the cost, and GM terminates. This algorithms does not need any parameters. The exact algorithm used here is presented in [2]. The worst case iteration count is in $O((M-1)N^2)$, where M is the number of PEs and N the number of tasks. A starting point near a local optimum will converge much more rapidly.

B. Variants of Simulated Annealing (SA)

SA is a probabilistic non-greedy algorithm [10] that explores search space of a problem by annealing from a high to a low temperature state. This paper uses two versions of SA that are presented in [2] and [3]. Algorithm performs random changes in mapping with respect to the current mapping state.

SA algorithm always accepts a move into a better state, but also into a worse state with a probability that decreases along with the temperature. Thus the algorithm becomes greedier on low temperatures. The acceptance probability function and the number of iterations per temperature level is set by the method presented in [2]. The annealing schedule function, initial temperature T_0 and the final temperature T_f are selected by the method in [3]. The algorithm terminates when the final temperature is reached and sufficient number of consecutive moves have been rejected.

The basic version of simulated annealing is referred here as *SA* and one with automatic temperature selection as *SA+AT*. In general, SA+AT achieves nearly the same performance as SA but in considerably fewer iterations. The Hybrid algorithm [4] uses SA for initial optimization and finishes the mapping with GM. The parameters for SA variants are temperature range (initial and final), number of temperature levels, scaling between levels, and number of iteration on each level. Furthermore, move heuristic, acceptance function, and end condition must be defined.

The total number of iterations for SA is

$$(\frac{\log \frac{T_f}{T_0}}{\log q} + 1)N(M-1), \tag{1}$$

where q is the temperature scaling factor [2].

C. Random

A simple random mapping algorithm is included just to obtain basis for algorithm comparison. The algorithm tries random mappings without regarding the results from previous iterations, hence it is probabilistic and non-greedy. The only parameter is the number of random mappings.

IV. Optimal Subset Mapping (OSM)

The new Optimal Subset Mapping (OSM) algorithm takes a random subset of tasks and finds the optimum mapping in that subset by trying all possible mappings (brute-force search). This is called a round. Tasks outside the subset are left in place. OSM is probabilistic because it chooses a subset randomly at every round. It is also greedy because it only accepts an improvement to the best known solution at each round. OSM algorithm was inspired by Sequential Minimal Optimization (SMO) algorithm [11]. SMO is used for solving a quadratic programming problem and has a static subset size of 2, but the subset size in OSM is dynamic during run-time. Also, the total number of iterations in OSM is bounded by task graph and architecture characteristics.

The pseudo-code of OSM is shown in Fig. 1. Variable S denotes the current (mapping) state, C is the cost, X is subset size, and R is a round number used to track progress of the algorithm. Function **Cost**(S) evaluates the cost function for the mapping state S and minimum S is sought. Function **Pick_Random_Subset**(S, X) picks a random subset of X separate tasks from mapping S. Function **Apply_Mapping**(S, S_{sub}) takes whole mapping S and subset mapping S_{sub}. It copies mappings from S_{sub} to S.

Initially, the subset size $X = 2$. If no improvement has been found within last $R_{max} = \lceil \frac{N}{X_{max}} \rceil$ rounds, the subset size X is increased by 1. If there was some improvement, X is decreased by 1. The subset size X is bounded to $[X_{min}, X_{max}]$, where $X_{min} = 2$. The algorithm terminates

```
OPTIMAL_SUBSET_MAPPING(S)
 1  S_best ← S
 2  C_best ← COST(S)
 3  X ← 2
 4  for R ← 1 to ∞
 5  do C_old_best ← C_best
 6     S ← S_best
 7     Subset ← PICK_RANDOM_SUBSET(S, X)
 8     for all possible mappings S_sub in Subset
 9     do S ← APPLY_MAPPING(S, S_sub)
10        C ← COST(S)
11        if C < C_best
12          then S_best ← S
13               C_best ← C
14     if modulo(R, R_max) = 0
15       then if C_best = C_old_best
16              then if X = X_max
17                     then break
18                   X ← X + 1
19              else X ← X − 1
20            X ← MAX(X_min, X)
21            X ← MIN(X_max, X)
22  return S_best
```

Fig. 1. Pseudo-code of Optimal Subset mapping algorithm.

when none of the last R_{max} rounds improved the solution and maximum subset size is reached ($X = X_{max}$).

Upper bound for subset size X is needed to limit the number of iterations. It can be derived as

$$M^X = cN^{c_N} M^{c_M}, \tag{2}$$

where N is the number of tasks and M is the number of PEs. c, c_N and c_M are arbitrary positive coefficients used to limit iterations with respect to system size defined by N and M. It is recommended that $c_N, c_M \geq 1$ to reach acceptably good results. Solution to (2) is

$$X_{max} = \lfloor \frac{\log(c) + c_N \log(N) + c_M \log(M)}{\log M} \rfloor. \tag{3}$$

As a consequence, the number of iterations increases as N and M increase. The total number of mappings for R_{max} rounds is in

$$O(\frac{N^{1+c_N} M^{c_M}}{\log N + \log M}). \tag{4}$$

V. EXPERIMENT SETUP

The experiment uses 10 random graphs with 300 nodes from the Standard Task Graph set [5]. The communication weights were generated randomly from uniform distribution. The resulting *communication-to-computation* ratios varied between graphs. The minimum, average and maximum byte/s for tasks in graphs are 8.1 Mbyte/s, 217.8 Mbyte/s, 525.6 Mbyte/s. This is the rate at which tasks will produce data in these graphs. Random graphs are used to evaluate optimization algorithms as fairly as possible. Non-random applications may well be relevant for common applications, but they are dangerously biased for general algorithm comparison. Investigating algorithm bias and classifying computational tasks based on the bias are outside the scope of this paper. Random graphs have the property to be neutral of the application. The task graphs are summarized in Table I along with HW platform and measurement setup.

TABLE I

APPLICATION AND ARCHITECTURE PARAMETERS FOR THE EXPERIMENT

	Variable (note)	Value
Task graphs	# graphs	10
	# tasks per graph (N)	302
	# edges per graph (1)	1594, 5231, 8703
	comp time per task [us] (1)	3.2, 5.1, 7.0
	comm vol per task [byte] (1)	26, 1111, 3679
	comm/comp -ratio [Mbyte/s] (1)	8, 218, 526
	max theor. parallelism [no unit] (1)	4.3, 7.9, 12.8
HW Platform	# PEs (M)	2, 4, 8
	PE freq [MHz]	50
	Bus Freq [MHz] (2)	10, 20, 40
	Bus width [bits]	32
	Bus bandwidth [Mb/s] (2)	320, 640, 1280
	Bus arb. latency [cycles/send]	8
Algorithms	# runs per graph per alg (3)	10
	algorithms	6
	determ, non-greedy	1: OSM
	determ, greedy	1: GM
	stoch., non-greedy	4: SA, SA+AT, hybrid, random
	stoch, greedy	-

Notes:
(1) = min, avg, max
(2) = values for 2,4,8 PEs, respectively
(3) = only 1 run for GM

The SoC platform is a message passing system where each PE has some local memory, but no shared memory. Each graph was distributed onto 2, 4 and 8 identical PEs. The PEs are interconnected with a single, dynamically arbitrated shared bus that limits the SoC performance due to bus contention. Bus frequency is low in order to highlight the differences between algorithms when HW resources are very limited. However, bus frequency is scaled with system size, as shown.

Total of 6 algorithms are compared. Optimization was run 10 times independently for each task graph, except with GM that needs only 1 run due to its deterministic behavior. The optimization software was written in C language and executed on a 10-machine Gentoo Linux cluster, each machine having a single 2.8 GHz Intel Pentium 4 processor and 1 GiB of memory. A total of $2 \cdot 10^9$ mappings were tried in 1869 computation hours (78 days) leading to average $297 \frac{mappings}{s}$.

The optimization parameters of the experiment are shown in Table II.

TABLE II

OPTIMIZATION PARAMETERS FOR THE EXPERIMENT

Alg.	Variable (note)	Value
SA, SA+AT, Hybrid	# iter per T, ($L = N \cdot (M\text{-}1)$) (1)	602, 1208, 2416
	# temperature levels	181
	# temperature scaling	q=0.95
	range of T (SA and hybrid) (2)	T_0= 1.0, T_f=0.0001
	range of T (SA+AT)	T range coefficient k=2
	annealing schedule (T_0, i)	$T_0 \cdot q^{floor(i/L)}$
	move heuristic	move 1 random task
	acceptance function	$(1 + \exp(\Delta C / (0.5\ C_0\ T))^{-1}$
	end condition	$T=T_f$ AND L rejected moves
Rand	# max interations	262 144
GM	no params needed	-
OSM	coefficient c	1.0
	exponent c_N	1.0
	exponent c_M	1.0
	subset size X [#tasks] (1)	9, 5, 3
	# iterations per subset (1)	512, 1024, 512

Notes:

(1) = values for 2,4,8 PEs, respectively

(2) = T_0 and T_f computed automatically in SA+AT

VI. RESULTS

For simplicity, the cost function considers only the execution time. Hence, gain equals speedup and speedup is defined as $\frac{t_1}{t_M}$, where t_i is the graph execution time on i PEs. The results are discussed according to average gain, progress of gain with respect to required iterations, and differences between graphs.

A. Gain

Figure 2 shows averaged speedups for each algorithm. Random mapping is clearly the worst algorithm and the difference between it and others grows with the number of PEs. Other 5 algorithms have almost equal performance, Hybrid being marginally better than others and GM and OSM slightly worse than others. SA, SA+AT and Hybrid are only marginally different in gain, from 0.01 to 0.04 gain units difference.

The average speedup grows with system size. For 2 PEs, total PE utilization varied from 77% to 99.7%. For 4 PEs, from 52% to 76%. And, for 8 PEs, from 37% to 51%. Interconnect utilization was nearly 100% as parallelization is communication bounded. Therefore, the gains are clearly lower than theoretical maximum parallelism. Average theoretical maximum parallelism is 7.9 for these graphs. It is defined by dividing the sum of computation times by the computation time of the critical path and neglecting the communication costs.

Variance in gain values is small, but there is a notable difference in the number of iterations that algorithms require during optimization. This will be analyzed next.

B. Time behavior of algorithms

Figure 3 shows the averaged speedups with respect to number of iterations for 8 PE system. The results with 2 and 4 PEs are similar but omitted for brevity. Note that

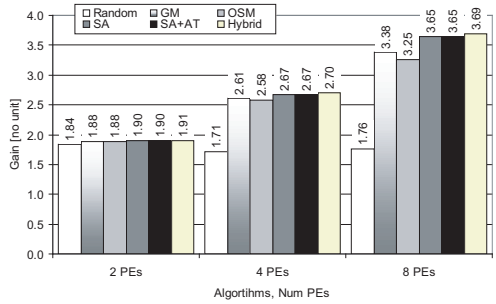


Fig. 2. Achieved gain averaged over 10 STGs.

the time for running an optimization algorithm is directly proportional to the number of iterations when graph size N and the architecture size M are fixed. Iterations are shown on a logarithmic scale and the first 1 000 evaluations are omitted from the figure.

GM needs many iterations to achieve any speedup but once that occurs the speedup increases very rapidly. A total opposite is the new OSM algorithm. It reaches almost the maximum speedup level with very limited number of iterations. SA, SA+AT and Hybrid lie between these extremes, and they achieve the highest overall speedup.

Random mapping saturates quickly and further iterations are unable to provide any speedup.

Hybrid algorithm converges very slowly due to a simple but inefficient temperature schedule. But once it is in the right temperature range (200 000 iterations), it converges up very rapidly. SA+AT has an optimized temperature range that starts rapid convergence already at 20 000 iterations and reaches the maximum before the Hybrid starts converging. The Hybrid algorithm does many independent annealings in different temperature ranges, and also uses group migration, and thus reaches a slightly higher maximum than SA+AT. If normal SA were plotted on Figure 3, it would follow Hybrid algorithm exactly till 380 000 iterations, because Hybrid algorithm begins with a normal SA.

C. Trade-off between gain and required iterations

Clearly, algorithms proceed at different speeds, i.e. gain increases with varying slope. The average slope is defined as

$$average\_gain\_slope = \frac{gain}{\#iterations}.$$

It defines how much the gain increases with one iteration on average.

Figure 4 shows the average gain slope values of algorithms relative to that of random mapping. Random mapping algorithm was chosen to be a reference to measure ease of parallelization. The slopes of GM and Hybrid decrease rapidly with system size, i.e. they spend rapidly increasing time with larger systems. Considering this trade-off between optimization result and time needed, OSM and SA+AT are the best algorithms.

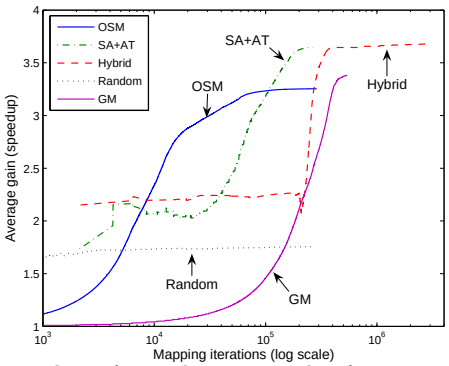


Fig. 3. Evolution of gain with respect to number of iterations with 8 PEs.

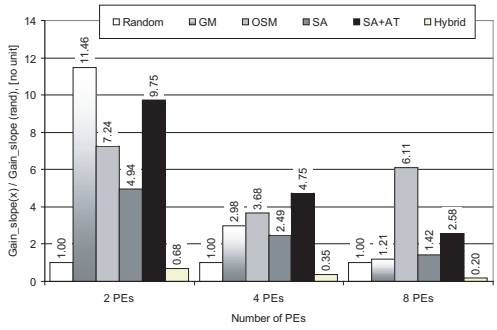


Fig. 4. The best gain divided by the number of iterations. Values are relative to random mapping.

TABLE III
ROUNDS AND MAPPING ITERATIONS FOR OSM

PEs	rounds (min, avg, max)	Thousands of iterations (min, avg, max)
2	271, 380, 611	34.1, 37.2, 73.6
4	239, 469, 899	80.6, 115.4, 259.1
8	199, 428, 1099	57.1, 88.8, 293.9

For 4 PEs, OSM algorithm is 23%, 48%, −23% and 951% more efficient than GM, SA, SA+AT and Hybrid, respectively. Efficiency is defined as achieved gain divided by the number of mapping iterations needed. The save in wall clock optimization time are 20%, 35%, −25% and 91%, respectively. This makes SA+AT a clear winner because it is only 0.03 gain units slower than Hybrid, but noticeably the fastest.

For 8 PEs, OSM algorithm is 405%, 330%, 137% and 2955% more efficient in average gain slope compared to GM, SA, SA+AT and Hybrid, respectively. The save in wall clock optimization time are 81%, 79%, 62% and 97%, respectively. That is, OSM is very efficient. However, SA+AT has a 12% or 0.40 units higher gain number than OSM, which makes SA+AT a very good candidate for the 8 PE case.

Table III shows rounds and iterations for OSM algorithm. The average number of rounds varies from 380 to 469 due to its parameter selection scheme, and the average number of iterations scales up with the number of processors.

D. Differences between the graphs

There are differences in obtained gain depending on the graph. The progress of OSM, SA+AT and GM for each graph is shown in Figure 5. Each line represents different graph. Results are for 8 PEs. OSM starts saturating always at the same point, after 10^4 iterations, for every graph, as illustrated in Figure 5(a). However, the gains differ at most +50% (gain of 3.7 vs 2.5). Both GM and SA+AT had similar difference between the best and worst case graphs. SA+AT achieves the largest speedup among algorithms at the beginning (iterations 1 - 1 000) due to its random mapping style in the beginning of annealing. Consequently, the differences between graphs become visible already at small iteration counts whereas they are observed only at the end with OSM and GM. The difference in iteration counts between graphs varied at most by factor of 3 for OSM, due to its subset size changing policy. Other algorithms were varied much less than that.

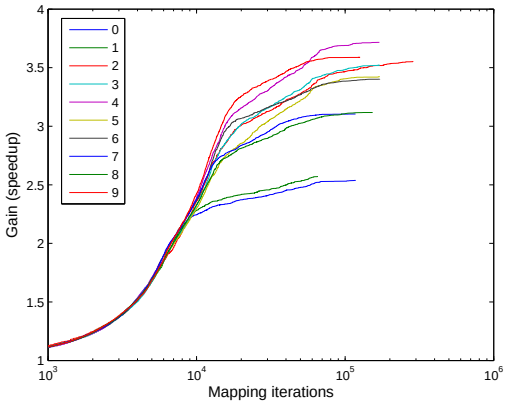
It turned out that the same graphs performed worse with every algorithm. The studied graphs varied in terms of connectivity and branching (number of edges) and computation amount. We carefully analyzed the correlation between the static properties of task graphs and achieved gain. However, no causal relation was found. For example, the two worst graphs had many edges (7109 and 8703) but the best had also many (7515). It is our optimistic hope that the presented results therefore present the "general" case as well.

E. Discussion

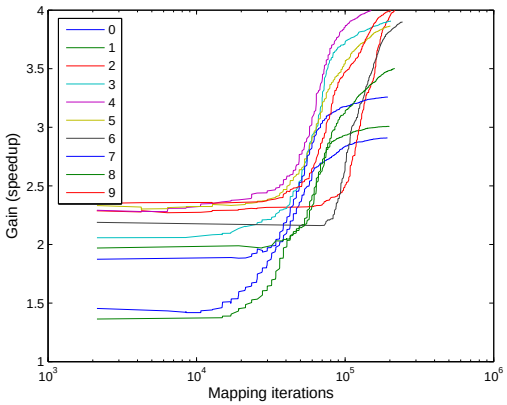
The Hybrid algorithm reaches the best speedup, but it is only marginally better than other SA variants. GM and OSM are clearly worse in 8 PE case, but do almost as well in 2 and 4 PE cases. This shows that Hybrid and SA variants are more scalable than OSM and GM. However, in terms of average gain slope, OSM and SA+AT the most scalable algorithms (see Section VI-C).

Hybrid and SA converge so slowly that they are useless for large scale architecture exploration. SA+AT is as good as SA in speedup but converges much more rapidly due to its parameter selection method. GM converges slowly compared to SA variants. OSM converges very rapidly, and therefore we suggest to use it early in the exploration. However, its final result is not as high as SA, which possibly means that another algorithm should continue after OSM, or OSM should be improved.

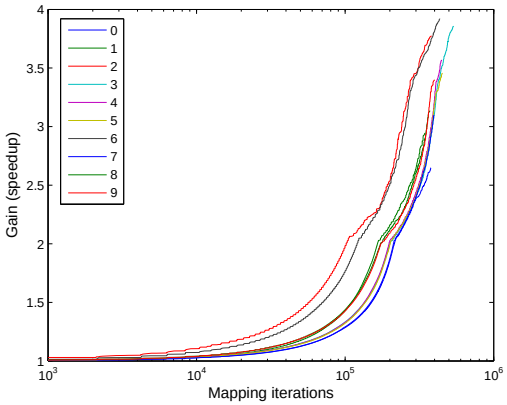
Hybrid and SA variants are insensitive to initial values due to their random nature in high temperatures. GM is highly sensitive to initial values due to its deterministic and greedy nature, and therefore we advice against using it without independent runs from different initial values. Effect of initial values to OSM is an open question, but it is reasonable to assume it depends on the maximum subset size and graph structure.



(a) OSM progress



(b) SA+AT progress



(c) GM progress

Fig. 5. OSM, SA+AT and GM progress plotted for each graph.

VII. Conclusion

This paper presented a new mapping algorithm, Optimal Subset Mapping (OSM), and it is compared to 5 other algorithms. OSM was developed to make architecture exploration faster. The results show large differences on the number or required iterations during the optimization so that OSM is a strong candidate for a rapid mapping algorithm when the number of iterations is taken into account. Simulated annealing with automatic temperature selection (SA+AT) gives nearly the best gain with reasonable effort, but OSM is faster in convergence. When only the speedup is measured, differences are small among algorithms.

Also, the paper presented convergence properties of 5 algorithms with respect to iteration number, number of processors and different random graphs. Convergence properties of OSM and GM have opposite behavior and SA comes between these two. The convergence figures presented in this paper should help architecture explorers choose a suitable algorithm for task mapping. OSM and SA+AT are the recommended choices of these algorithms.

Future research should try to integrate rapid convergence properties of OSM to other algorithms, create a non-greedy version of OSM to have similar advantages as SA+AT, increase the granularity of subsets of tasks (map several subsets of tasks optimally, instead of mapping a subset of tasks optimally), and analyze the relation between graph properties and gain.

References

[1] Y.-K. Kwok and I. Ahmad, *Static scheduling algorithms for allocating directed task graphs to multiprocessors*, ACM Comput. Surv., Vol. 31, No. 4, pp. 406-471, 1999.

[2] H. Orsila, T. Kangas, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, *Automated Memory-Aware Application Distribution for Multi-Processor System-On-Chips*, Journal of Systems Architecture, 2007, Elsevier, In print.

[3] 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 (SoC 2006), Tampere, Finland, November 14-16, 2006, pp. 73-76.

[4] 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), pp. 146-150, 2005.

[5] *Standard task graph set*, [online]: http://www.kasahara.elec.waseda.ac.jp/schedule, 2003.

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

[7] G. Kahn, *The semantics of a simple language for parallel programming*, Information Processing, pp. 471-475, 1974.

[8] 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, ACM, 2006.

[9] 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.

[10] S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi, *Optimization by simulated annealing*, Science, Vol. 200, No. 4598, pp. 671-680, 1983.

[11] J. Platt, *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*, Microsoft Research Technical Report MSR-TR-98-14, 1998.

PUBLICATION 5

Evaluation of Heterogeneous Multiprocessor Architectures by Energy and Performance Optimization

Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen
Department of Computer Systems
Tampere University of Technology
P.O. Box 553, 33101 Tampere, Finland
Email: {heikki.orsila, erno.salminen, marko.hannikainen, timo.d.hamalainen}@tut.fi

***Abstract*— Design space exploration aims to find an energy-efficient architecture with high performance. A trade-off is needed between these goals, and the optimization effort should also be minimized. In this paper, we evaluate heterogeneous multiprocessor architectures by optimizing both energy and performance for applications. Ten random task graphs are optimized for each architecture, and evaluated with simulations. The energy versus performance trade-off is analyzed by looking at Pareto optimal solutions. It is assumed that there is a variety of processing elements whose number, frequency and microarchitecture can be modified for exploration purposes. It is found that both energy-efficient and well performing solutions exist, and in general, performance is traded for energy-efficiency. Results indicate that automated exploration tools are needed when the complexity of the mapping problem grows, starting already with our experiment setup: 6 types of PEs to select from, and the system consists of 2 to 5 PEs. Our results indicate that our Simulated Annealing method can be used for energy optimization with heterogeneous architectures, in addition to performance optimization with homogeneous architectures.**

I. Introduction

An efficient multiprocessor SoC (MPSoC) implementation requires automated exploration to find an efficient HW allocation, task mapping and scheduling [1]. Heterogeneous MPSoCs are needed for low power, high performance, and high volume markets [2]. The central idea in multiprocessing SoCs is to increase performance while decreasing energy consumption. This is achieved by efficient communication between cores and keeping clock frequency low.

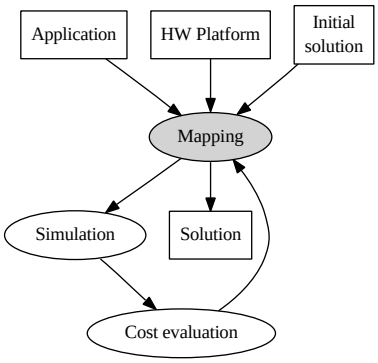
Mapping means placing each application component to some processing element (PE). Scheduling means determining execution order of the application components on the platform. A large design space must be pruned systematically, since the exploration of the whole design space is not feasible [1]. Fast optimization procedure is desired in order to cover reasonable design space. However, this comes with the expense of accuracy. Iterative optimization algorithms evaluate a number of application mappings for each resource allocation candidate. For each mapping, the application is scheduled and simulated to evaluate the cost of the solution, i.e. the value of the objective function. The objective function may consider multiple parameters, such as execution time, communication time, memory, energy consumption and silicon area constraints. Figure 1(a) shows the mapping process.

We present an experiment where a set of hardware architectures is generated by random, and applications are mapped on them. Hardware architectures are 2 to 5 PE systems with both singlescalar and superscalar PEs with frequencies from 100MHz to 300MHz. The total area of the system is limited to $8mm^2$. Applications are 300 node acyclic static task graphs (STGs) [3]. Figure 1(b) shows the application, its mapping, and the hardware platform. The application is optimized for each architecture with respect to the energy for that is consumed when the application is run. Resulting energy and execution time values for each architecture are analyzed to find Pareto optimal architectures. Hence, applications are optimized with a single objective (energy), but architectures are analyzed by two objectives.

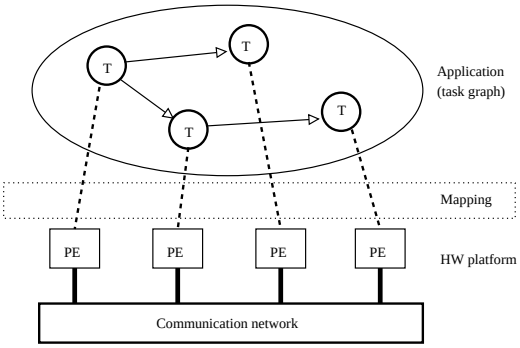
With the constraints of our experiment, the results show there is a clear trade-off for energy and performance. Low number of PEs means weak performance, but low power. High number of PEs means more performance, but loses energy-efficiency. Also, increasing number of PEs creates demand for automated exploration tools, as the mapping problem becomes more important and increasingly harder.

II. Related Work

Our earlier work [4] evaluated various mapping algorithms to determine optimization convergence rate when application performance was maximized. This paper adds heterogeneous PEs to the problem. Simulated annealing (SA) was found to be an efficient algorithm, and therefore, it is used also in this paper. SA is a probabilistic non-greedy algorithm [5] that explores search space of a problem by annealing from a high to a low temperature state. These methods are also used in our Koski flow [6]. Koski is a high-level design tool for multiprocessor SoCs and applications. Koski utilizes Kahn Process Networks [7] for application modeling.



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



(b) An example MPSoC that is optimized. The system consists of the application and the HW platform. T denotes a task, and PE denotes a processing element.

Fig. 1. Diagram of the optimization process and the system that is optimized

III. Experiment Setup

A. Objective And Optimization Algorithm

An experiment was done to investigate trade-offs between energy and application performance. The objective function (1) is to minimize the total energy consumption (static + dynamic). The energy is measured in relative values. It is not a physical energy unit.

$$E = T(P_s + P_d) = T(\sum_{i=1}^{N} A_i f_{max} + k \sum_{i=1}^{N} A_i f_i U_i) \quad (1)$$

where T is the execution time of the application, determined in simulation. N is the number of PEs, A_i is the area of PE i and f_i is its frequency. f_{max} is the maximum frequency of any PE or interconnect, which is at least 200MHz in this experiment because the bus operates at 200MHz. Utilization U_i is the proportion of non-idle cycles of the PE i. The HW architecture defines values A_i and f_i whereas the mapping indirectly defines T and values U_i.

Coefficient k is the factor that changes the relative proportion of static versus dynamic energy. Energy values are comparable when k value is constant. The effect of dynamic power can be eliminated by setting $k = 0$. The static power part P_s of the objective function depends on the number of transistors (relative to A) and their speed (relative to f_{max}). The dynamic power part P_d depends on total capacitance (relative to A), switching frequency f_i, and activity U_i. Supply voltage is assumed fixed.

Simulated annealing algorithm was used to optimize energy (1) by changing application mappings. The algorithm is specified in [8], but modified in two ways. First, the objective function is the application energy on a given platform. Second, the algorithm is run twice for each solution, and the second run always starts from the best solution of the first run. This was done to increase confidence in results, as the SA is stochastic. SA temperature margin value 2 was used to scale initial and final temperatures [4].

B. Simulated HW Platform

The simulated SoC platform for the experiment is a message passing system where each PE has some local memory, but no shared memory. The system is simulated on the behavior level. Each PE and interconnection resource is available for a single action at a time. PE task context switch overhead is 0 cycles, but bus arbitration time is 8 cycles for each transfer.

Figure 1(b) presents simulated HW platform. PEs are interconnected with two shared buses that are independently and dynamically arbitrated. The shared buses limit SoC performance due to contention, latency and throughput. Bus frequency is 200MHz for both buses and they are 32 bits wide. The bus silicon area is $0.1mm^2$ per processor. Each node in the task graph sends a specific number of bytes after computation, thus creating contention on the shared buses. Which ever bus is free at a time is used for communication by using FIFO arbitration, i.e. which ever PE comes first gets the bus.

Table I shows different types of PEs, each presented with a letter. Multiprocessor architectures were varied using these PEs. An architecture consists of 2 to 5 PEs, and the total area has $8mm^2$ upper-bound. Architectures are presented with fingerprint codes from these letters. Parameter p is the average number of instructions per cycle. Frequency f has 3 values: 100MHz, 200MHz and 300MHz. Processor speed is measured in millions of operations per second, which equals $p * f$. A is the area in square millimeters. Each PE can be implemented as a singlescalar or as a superscalar version. The superscalar version can execute $p = 1.8$ instructions per clock. The singlescalar version has area $A = 1mm^2$, superscalar has $A = 2mm^2$. 2 values of p and 3 values of f implies 6 different PEs. A task graph node of n cycles can be computed in $\frac{n}{fp}$ time.

C. Architecture Fingerprinting

Architecture fingerprinting is used to present results. An architecture is characterized by a series of letters from A to F. Letters are labels for different PEs specified in Table I. Letters are assigned in the order of increasing number of operations per second. Letter A is assigned for the slowest PE, and letter F

TABLE I

AVAILABLE PROCESSOR TYPES

PE type	f (MHz)	p ($\frac{Ops}{cycle}$)	Speed ($\frac{MOps}{s}$)	A (mm^2)
A	100	1.0	100	1
B	100	1.8	180	2
C	200	1.0	200	1
D	300	1.0	300	1
E	200	1.8	360	2
F	300	1.8	540	2

TABLE II

PROPORTION OF HOW MANY TIMES A GIVEN NUMBER AND TYPE OF PE WAS IN THE EXPERIMENT'S 141 FINGERPRINTS. TABLE VALUES ARE EXPLAINED IN SECTION III-D.

PE type	1 PE (%)	2 PEs (%)	3 PEs (%)	4 PEs (%)	5 PEs (%)	PE prop. (%)
A	28.4	10.6	2.8	0.7	0.7	43.3
B	28.4	7.1	1.4	0.0	0.0	36.9
C	30.5	9.2	4.3	1.4	0.7	46.1
D	31.9	8.5	5.7	1.4	0.7	48.2
E	31.9	10.6	2.1	0.0	0.0	44.7
F	29.1	9.2	1.4	0.0	0.0	39.7
Any	0.0	14.2	27.7	33.3	24.8	

is assigned for the fastest PE. Each PE in the architecture gets a single letter in the architecture fingerprint. The letters are organized into alphabetical order to facilitate human brain's pattern recognition, i.e. make it easier to see slow, fast and same type of PEs. For example, AAB fingerprint means a three PE architecture with two PEs of type A and one PE of type B. The two PEs of type A are in the beginning of the series to display that there are exactly two instances of A in the architecture and that they are the slowest PEs.

The architecture fingerprint can be extended for heterogeneous interconnections by separating PE selections and interconnection selections with a dash (-). However, the interconnection is the same for all architectures in this paper, and therefore, it is omitted.

D. Random Architectures

141 different architectures were generated. Homogeneous architectures, the architectures with only one type of PE, were inserted manually, and the rest were generated randomly. The total area for each architecture was limited to $8mm^2$. Table II shows the proportion of how many times a given number and type of PE was in all the fingerprints. Rows indicate PE types, and columns indicate proportion of PEs. The last row shows the proportion of architectures with a given number of PEs. The last column shows the proportion of architectures that had at least one PE of that row's type. For example, row A's third column value means that 10.6% of architectures had exactly 2 PEs of type A. Last row's third column value means that 14.2% of the fingerprints had exactly 2 PEs. Row A's last column shows that $0.433 * 141 = 61$ architectures had at least one A type PE. The table has zeroes due to area constraints. For example, an architecture with 4 B type PEs does not exist, because one B type PE takes $2mm^2$, and its associated interconnect area is $0.1mm^2$. Therefore, the total area is $4 * (2 + 0.1)mm^2 > 8mm^2$.

TABLE III

ATTRIBUTES AND LIMITS OF THE EXPERIMENT

Attribute	Values
Number of architectures	141
Maximum architecture area	$8mm^2$
Number of PEs in each architecture	2 to 5
Number of 300 node graphs	10
Objective function to optimize energy	$T(\sum A_i f_{max} + k \sum A_i f_i U_i)$, where i is from 1 to N PEs
k values	0, 1, 4
Optimization algorithm	Simulated annealing

E. Applications

The experiment uses ten random task graphs with 300 nodes from the Standard Task Graph set [9]. Random graphs are used to avoid bias in algorithms and results. Nodes of the STG are finite, deterministic computational tasks, and edges denote dependencies between the nodes. Node weights represent the amount of computation associated with a node. Edge weights model the amount of communication needed to transfer results between the nodes. Computational nodes block until their data dependencies are resolved, i.e. when they have all needed data. The edge weights were generated randomly from uniform distribution.

STGs are used because there exists well known efficient and near optimal scheduling algorithms for them [3]. This ensures that the observed differences in optimization results are due to mapping, not scheduling. More complex scheduling properties would diminish accuracy of mapping analysis. However, this experiment is agnostic of the STG structure, and so it could be done with general process networks like Kahn Process Networks (KPN).

F. Experiment Data

Table III shows attributes that were varied in the experiment. Each of the 10 task graphs was optimized and simulated against each architecture. This was done for three values of $k = 0, 1$ or 4 to change static versus dynamic energy balance. Thus, total of $10 * 141 * 3 = 4230$ simulations was run.

G. Software

The optimization software and simulator was written in C language and executed on a 9 machine Red Hat Enterprise Linux WS release 3 cluster, each machine having a single 2.8 GHz Intel Pentium 4 processor and 1 GiB of memory. Jobs were distributed to a cluster with *jobqueue* [10] (version control snapshot *2008-05-30*) by using OpenSSH [11] and rsync [12]. No special clustering software or configurations was used. A total of $8.48 \cdot 10^8$ mappings was evaluated in optimization in 27.4 computation days leading to average of $358 \frac{mappings}{s}$. Rapid mapping evaluation is a benefit of STGs.

IV. RESULTS

Figure 2 plots energy versus execution time for each architecture for $k = 1$ that emphasizes static energy. Figure 3 plots the same data for $k = 4$, i.e. bigger weight on dynamic energy. Energy values are summed and time values are summed for

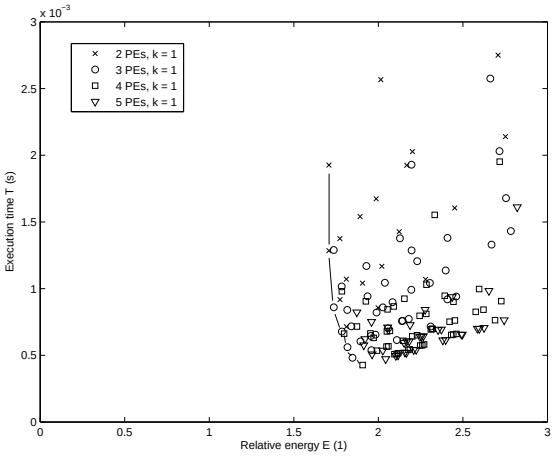


Fig. 2. Energy-time plot for different architectures with $k = 1$

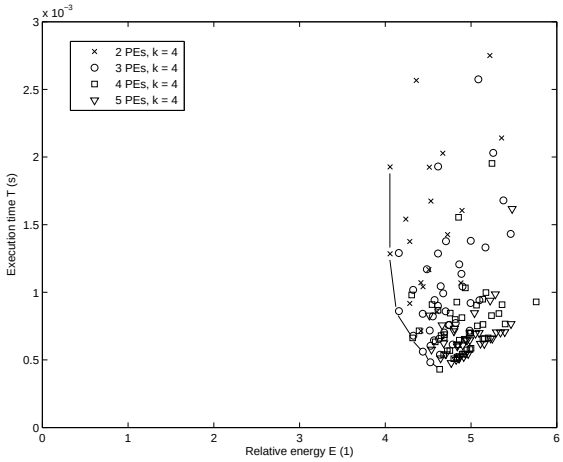


Fig. 3. Energy-time plot for different architectures with $k = 4$

each application. Energy E is the sum of objective function values (1) for a given architecture. Execution time T is the sum of execution times for a given architecture. Time is measured in seconds. Time is comparable even between different k values, but energy is not. A pair (E, T) presents a data point, or architecture, in the figure. E varied in range $[1.7, 3.1]$ for $k = 1$, and $[4.0, 5.8]$ for $k = 4$. Execution time sum T varied in range $[0.43, 3.85]$ms for both cases.

Pareto optimal solution boundary is marked with straight lines. These are not absolute Pareto optimums as not all possible architectures were evaluated. A Pareto optimal architecture is such that improving either energy or execution time leads to worsening the other factor. That is, in the Pareto optimal architectures, there are no two architectures where the other is better in terms of both energy and execution time.

TABLE IV

TOP 10 ARCHITECTURES TOGETHER WITH 3 PARETO OPTIMAL ARCHITECTURES FOR $k = 0$. ARCHITECTURES ARE LABELED WITH RESPECT TO THEIR ARCHITECTURE FINGERPRINTS. PARETO OPTIMAL ARCHITECTURES ARE MARKED WITH *. BEST VALUES IN EACH COLUMN ARE IN BOLD.

Arch. finger-print	E (1)	T (ms)	$\frac{E_{AA}}{E}$ (1)	$\frac{T_{AA}}{T}$ (1)	U_P (%)	U_I (%)	A (mm^2)
CC*	**0.925**	1.926	**1.999**	1.999	**100**	10	**2.4**
DD*	0.925	1.285	**1.999**	2.998	**100**	14	**2.4**
CCC	0.928	1.289	1.993	2.989	**100**	28	3.6
DDD*	0.928	0.860	1.991	4.481	99	41	3.6
CCE	0.935	1.016	1.977	3.789	99	35	4.6
CE	0.935	1.375	1.977	2.800	**100**	13	3.4
DF	0.936	0.917	1.976	4.199	**100**	20	3.4
DDF*	0.936	0.678	1.975	5.677	99	51	4.6
CCCC	0.941	0.980	1.965	3.931	98	52	4.8
CEE	0.941	0.840	1.964	4.583	99	42	5.6
DFF*	0.943	0.561	1.961	6.864	99	62	5.6
FFF*	0.953	0.482	1.939	7.999	98	73	6.6
DFFF*	1.001	**0.428**	1.847	**9.005**	90	**96**	7.8

A. Top Architectures

Table IV, Table V and Table VI show top 10 and all Pareto optimal architectures for cases $k = 0$, $k = 1$ and $k = 4$, respectively. $k = 0$ case is practically pure performance optimization, although measured in energy, but $k = 1$ and $k = 4$ are strictly energy optimization. Energy E and total execution time T are absolute values, and they are comparable to the slowest 2-PE architecture AA. $\frac{E_{AA}}{E}$ is the energy gain over AA architecture, the bigger the better. $\frac{T_{AA}}{T}$ is the speedup over AA architecture, the bigger the better. U_P is the mean PE utilization. U_I is the mean interconnect utilization. A is the area measured in square millimeters.

CC wins energy with all values of k. In $k = 1$ case, it is 1.7% more energy-efficient than the nearest 3 PE solution, CCC. It is $4,5\%$ more energy-efficient than the nearest 4 PE solution, CCCC. When the role of the dynamic energy increases in $k = 4$, the differences are larger: 2.5% and 6.2% against CCC and CCCC, respectively.

DFFF is the fastest architecture, and also a Pareto optimum. It has the highest performance processors given the area constraints. Note that 5 PE solutions do not have performance advantage over 4 PE solutions due to area constraints. For all values of k, DFFF runs at $4.5\times$ speed compared to the most energy-efficient architecture CC. However, it consumes only 8.2% ($k = 0$) to 14.3% ($k = 4$) more energy.

Most energy-efficient architectures have lower interconnect utilization U_I and higher processor utilization U_P than the fastest architectures. Lower processor utilization in high performance architectures can be explained with high peeks of performance demand that they can satisfy. Low performance architectures have longer task queues during peeks, which balances the load in time, but makes the critical path longer.

Approximately half the architectures are homogeneous in top 10.

Table VII and Table VIII show the proportion of how many times a given number and type of PE was in top 10 least

TABLE V

TOP 10 ARCHITECTURES TOGETHER WITH 3 PARETO OPTIMAL ARCHITECTURES FOR $k = 1$. FIGURE 2 SHOWS ARCHITECTURES GENERATED FOR $k = 1$ CASE.

Arch. finger-print	E (1)	T (ms)	$\frac{E_{AA}}{E}$ (1)	$\frac{T_{AA}}{T}$ (1)	U_P (%)	U_I (%)	A (mm^2)
CC*	**1.707**	1.926	**1.541**	1.999	**100**	10	**2.4**
DD*	1.708	1.285	**1.541**	2.998	**100**	14	**2.4**
CCC	1.736	1.289	1.516	2.988	99	27	3.6
DDD*	1.737	0.860	1.515	4.479	99	40	3.6
CE	1.773	1.376	1.484	2.800	**100**	13	3.4
DF	1.773	0.917	1.484	4.199	**100**	19	3.4
CCE	1.783	1.016	1.476	3.791	**100**	34	4.6
CCCC	1.784	0.980	1.475	3.929	98	50	4.8
DDF*	1.784	0.678	1.475	5.679	99	51	4.6
DDDD*	1.798	0.663	1.464	5.810	96	73	4.8
DFF*	1.817	0.561	1.448	6.864	99	61	5.6
FFF*	1.847	0.482	1.425	7.998	98	72	6.6
DFFF*	1.907	**0.427**	1.380	**9.028**	90	**95**	7.8

TABLE VI

TOP 10 ARCHITECTURES TOGETHER WITH 3 PARETO OPTIMAL ARCHITECTURES FOR $k = 4$. FIGURE 3 SHOWS ARCHITECTURES GENERATED FOR $k = 4$ CASE.

Arch. finger-print	E (1)	T (ms)	$\frac{E_{AA}}{E}$ (1)	$\frac{T_{AA}}{T}$ (1)	U_P (%)	U_I (%)	A (mm^2)
CC*	**4.055**	1.927	**1.228**	1.999	**100**	9	**2.4**
DD*	4.056	1.285	**1.228**	2.998	**100**	14	**2.4**
CCC	4.158	1.290	1.198	2.986	99	26	3.6
DDD*	4.159	0.861	1.197	4.476	99	40	3.6
CD	4.239	1.541	1.175	2.500	**100**	12	**2.4**
CE	4.285	1.376	1.162	2.800	**100**	13	3.4
DF	4.285	0.917	1.162	4.199	**100**	19	3.4
CCCC	4.308	0.980	1.156	3.930	98	49	4.8
DDDD*	4.319	0.663	1.153	5.812	96	71	4.8
CCE	4.325	1.018	1.151	3.785	99	33	4.6
DFF*	4.438	0.561	1.122	6.862	99	60	5.6
FFF*	4.526	0.482	1.100	7.990	98	70	6.6
DFFF*	4.633	**0.430**	1.075	**8.952**	91	**93**	7.8

energy consuming architectures for cases $k = 1$ and $k = 4$.

In $k = 1$ and $k = 4$ cases, 2 PE solutions filled 4 and 5 of the top 10 positions, respectively. 3 PE solutions filled 4 and 3 in those cases. 4 PE solutions filled 2 positions in both cases. 2 and 3 PE solutions seem suitable for low energy applications. However, 3 and 4 PE solutions have high performance.

There are no A and B type PEs in the top 10. This can be attributed to poor performance and energy inefficiency. f_{max} is a determining factor for static energy (1), and it puts processors with frequency less than f_{max} into disadvantage. The minimum value of f_{max} is 200MHz, because the interconnect is clocked at 200MHz. For this reason, A and B types are not favored. However, C and E types have the advantage of not increasing f_{max}. C type was the most common processor among low-energy architectures.

B. Pareto Optimal Solutions

Pareto optimal solutions are labeled with asterisk (*) in Table V and Table VI.

For the $k = 1$ case, static energy proportion for Pareto optimal architectures varies between 52% and 54%. For the $k = 4$ case, it varies between between 21% and 23%. Thus, the energy profile is rather uniform for both cases.

Pareto optimal solutions have 2, 3 and 4 PEs. 2-PE solutions do well due to low energy. 3 and 4 PE systems are do well due to a trade-off between energy and performance.

Figure 2 and Figure 3 show the clustering of solutions in the design space. Pareto optimal solutions constitute mere 5% and 6% of all solutions (7 and 8 out of 141) for $k = 1$ and $k = 4$, respectively. Therefore, automatic exploration is needed even when design space is limited to only 6 types of PEs and 2 to 5 PEs per architecture. It is not feasible to try out these solutions by manual work.

TABLE VII

PROPORTION OF HOW MANY TIMES A GIVEN NUMBER AND TYPE OF PE WAS IN TOP 10 ARCHITECTURES WITH $k = 1$. TABLE VALUES ARE EXPLAINED IN SECTION III-D.

PE type	Once (%)	Twice (%)	3 times (%)	4 times (%)	PE prop.
A	0	0	0	0	0
B	0	0	0	0	0
C	10	20	20	10	60
D	10	20	20	10	60
E	20	0	0	0	20
F	20	0	0	0	20

TABLE VIII

PROPORTION OF HOW MANY TIMES A GIVEN NUMBER AND TYPE OF PE WAS IN TOP 10 ARCHITECTURES WITH $k = 4$. TABLE VALUES ARE EXPLAINED IN SECTION III-D.

PE type	Once (%)	Twice (%)	3 times (%)	4 times (%)	PE prop.
A	0	0	0	0	0
B	0	0	0	0	0
C	20	20	10	10	60
D	20	10	10	10	50
E	20	0	0	0	20
F	10	0	0	0	10

C. Optimization Convergence

Figure 4 and Figure 5 plot ratio $\frac{E_{AA}}{E}$ against mapping iterations for each Pareto optimal solution. The number of iterations it takes to win AA increases as the number of PEs increases. This comes from increased complexity of the mapping problem and the SA mapping algorithm that scales up iterations with respect to architecture complexity, the number of PEs. 4 PE architectures take over 20000 more iterations than 3 PE architectures to reach the level of AA (the gain value 1.0).

Our earlier work [8] presented an automated parameterization method for SA mapping. Originally it was only used for homogeneous architectures and performance optimization. The energy-time trade-offs presented in this paper indicate that the method can also be used for heterogeneous architectures and energy optimization.

In order to reach energy-efficiency of even AA architecture, it takes tens of thousands of mappings for 4 PE systems. Hence, it is a non-trivial problem in most cases. This creates demand for automated mapping (exploration). This may require behavior level simulation due to simulation time,

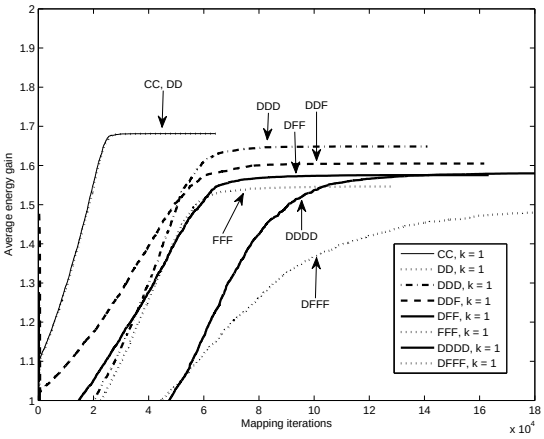


Fig. 4. Average energy gain plotted against mapping optimization iterations for the $k = 1$ case. Gain is computed as reference energy value divided by an energy value. Average gain is normalized to the average best objective value of the AA architecture. DD architecture's value 1.7 means AA consumed 1.7 times the energy of DD.

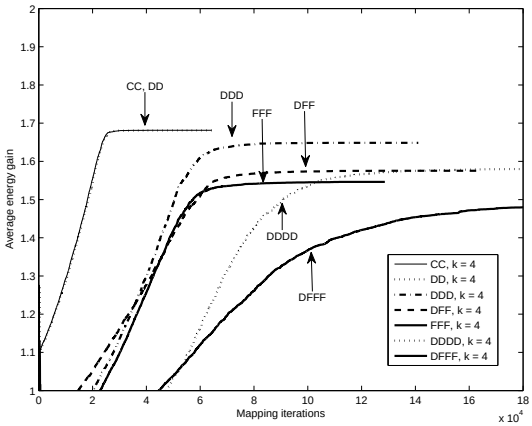


Fig. 5. Average energy gain plotted against mapping optimization iterations for the $k = 4$ case.

as was done in this paper. More accurate simulation may need thousands of CPUs. Fortunately, thousands of CPUs are reachable with current development budgets.

V. Conclusion

We evaluated heterogeneous architectures by optimizing both energy and performance for applications. The energy versus performance trade-off was analyzed by looking at Pareto optimal solutions. It was found that both energy-efficient and well performing solutions exist, and in general, performance is traded for energy-efficiency. Results indicated that automated exploration tools are needed when the mapping problem complexity grows, starting already with our experiment setup: 6 types of PEs to select from, and the system consists of 2 to 5 PEs.

Also, the results show that our Simulated Annealing method can be used in energy optimization for heterogeneous architectures, as well as in performance optimization for homogeneous architectures.

In the future, we plan to utilize SA to directly seek an optimal HW allocation and consider the bus or NoC energy more closely.

References

[1] M. Gries, *Methods for evaluating and covering the design space during early design development*, Integration, the VLSI Journal, Vol. 38, Issue 2, pp. 131-183, 2004.
[2] W. Wolf, *The future of multiprocessor systems-on-chips*, Design Automation Conference 2004, Proceedings. 41st, pp. 681-685, 2004.
[3] Y.-K. Kwok and I. Ahmad, *Static scheduling algorithms for allocating directed task graphs to multiprocessors*, ACM Comput. Surv., Vol. 31, No. 4, pp. 406-471, 1999.
[4] H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, *Optimal Subset Mapping And Convergence Evaluation of Mapping Algorithms for Distributing Task Graphs on Multiprocessor SoC*, International Symposium on System-on-Chip, 2007.
[5] S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi, *Optimization by simulated annealing*, Science, Vol. 200, No. 4598, pp. 671-680, 1983.
[6] 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, ACM, 2006.
[7] G. Kahn, *The semantics of a simple language for parallel programming*, Information Processing, pp. 471-475, 1974.
[8] 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, pp. 73-76.
[9] *Standard task graph set*, ONLINE: http://www.kasahara.elec.waseda.ac.jp/schedule, 2003.
[10] *jobqueue*. Software. ONLINE: *http://zakalwe.fi/ shd/foss/jobqueue/*
[11] *OpenSSH*, software, ONLINE: *http://openssh.org*
[12] *rsync*, software, ONLINE: *http://rsync.samba.org*

PUBLICATION 6

H. Orsila, E. Salminen, T. D. Hämäläinen, *Best Practices for Simulated Annealing in Multiprocessor Task Distribution Problems*, Chapter 16 of the Book "Simulated Annealing", ISBN 978-953-7619-07-7, I-Tech Education and Publishing KG, pp. 321-342, 2008.

16

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

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.

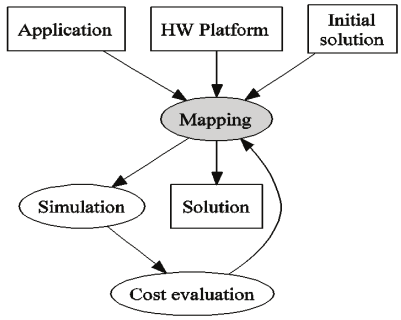


Figure 1(a). Optimization process. Boxes indicate data. Ellipses indicate operations. This chapter focuses on the mapping part.

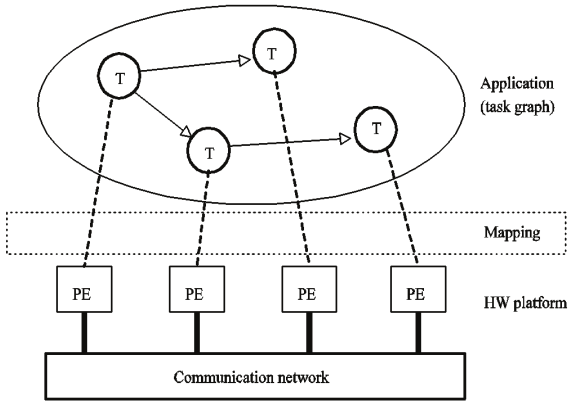


Figure 1(b). The system that is optimized. The system consists of the application and the HW platform. PE is processing element.

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.

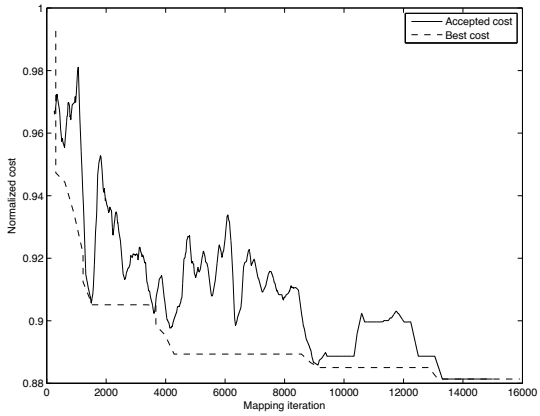


Figure 2. Cost per iteration plotted for Simulated Annealing when mapping a *100* task application to a *4* processor system. The cost is normalized so that initial cost $C_0 = 1.0$. The plot is average filtered with a *256* sample window to hide the chaotic nature of the random process. This is also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

```
SIMULATED_ANNEALING(S_0)
 1  S ← S_0
 2  C ← COST(S_0)
 3  S_best ← S
 4  C_best ← C
 5  R ← 0
 6  for i ← 0 to ∞
 7      do T ← TEMP(i)
 8         S_new ← MOVE(S, T)
 9         C_new ← COST(S_new)
10         ΔC ← C_new − C
11         if ΔC < 0 or ACCEPT(ΔC, T)
12            then if C_new < C_best
13                    then S_best ← S_new
14                         C_best ← C_new
15                 S ← S_new
16                 C ← C_new
17                 R ← 0
18            else  R ← 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
$Accept(\Delta C, T)$	{**False**, **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	
$Terminate(i, R)$	{**False**, **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

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

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 (∆C,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 $0 < T \le 1$. 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.

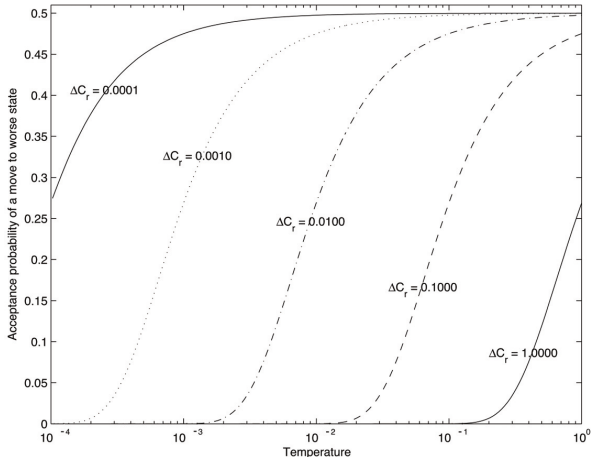


Figure 4. Acceptance probability curves for the normalized inverse exponential function (6) with $\mathbf{q = 0.95}$. The curve represents constant values of $\mathbf{\Delta C_r = \frac{\Delta C}{C_0}}$. Probability of moving to a worse state decreases when the temperature decreases. Moves to slightly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

3.3.4 Normalized exponential form

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

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:

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

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 :

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

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:

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

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 $R \geq R_{\max}$:

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

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:

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

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:

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

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

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

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

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.

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

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 $\frac{\Delta C_{\min}}{0.5C_0}$ in place of $\Delta C_{\min}$ to make (10) compatible with (20) gives

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

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

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

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

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

Solving (23) with respect to p_f gives us

$$p_f \sim \frac{1}{e^k + 1} \tag{24}$$

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 $k = 1$.

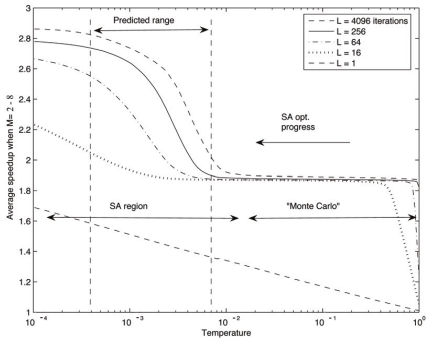


Figure 5. Averaged speedup with respect to temperature for *300* node graphs with different L values. The temperature given with (18)(19) $k = 1$ is labeled „predicted range". Notice that temperature and the number of iterations increase in different directions. The number of mapping iterations increases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

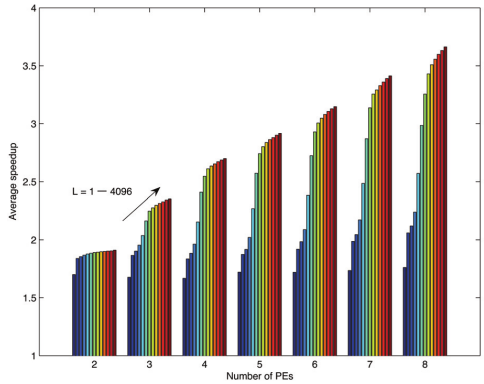


Figure 6. Averaged speedups for 300 node graphs with *M=2-8* processing elements and different *L* values (*L = 1, 2, 4, ..., 4096*) for each processing element set.

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).

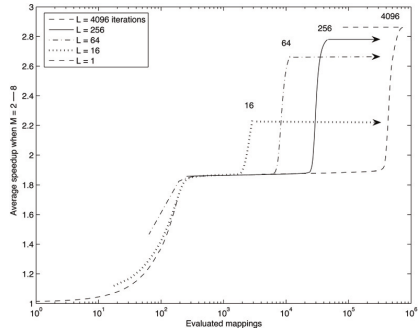


Figure 7. Averaged speedup with respect to mapping evaluations for 300 node graphs with different L values.

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.

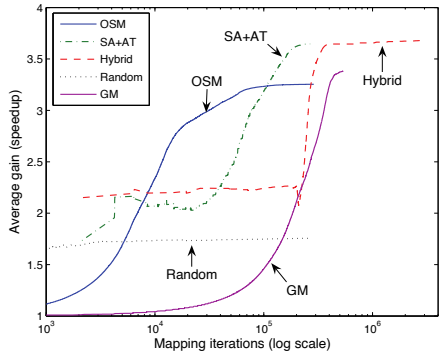


Figure 8. SA convergence speed compared to GM, OSM and Random Mapping algorithms for mapping *300* tasks to *8* processors. SA+AT is a Simulated Annealing algorithm presented in Section 4.3. GM and OSM are greedy heuristics.

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.

PUBLICATION 7

Parameterizing Simulated Annealing for Distributing Kahn Process Networks on Multiprocessor SoCs

Heikki Orsila, Erno Salminen and Timo D. Hämäläinen
Department of Computer Systems
Tampere University of Technology
P.O. Box 553, 33101 Tampere, Finland
Email: {heikki.orsila, erno.salminen, timo.d.hamalainen}@tut.fi

***Abstract*—Mapping an application on Multiprocessor System-on-Chip (MPSoC) is a crucial step in architecture exploration. The problem is to minimize optimization effort and application execution time. Simulated annealing (SA) is a versatile algorithm for hard optimization problems, such as task distribution on MPSoCs. We propose an improved automatic parameter selection method for SA to save optimization effort. The method determines a proper annealing schedule and transition probabilities for SA, which makes the algorithm scalable with respect to application and platform size. Applications are modeled as Kahn Process Networks (KPNs). The method was improved to optimize KPNs and save optimization effort by doing sensitivity analysis for processes. The method is validated by mapping 16 to 256 node KPNs onto an MPSoC. We optimized 150 KPNs for 3 architectures. The method saves over half the optimization time and loses only 0.3% in performance to non-automated SA. Results are compared to non-automated SA, Group migration, random mapping and brute force algorithms. Global optimum solution are obtained by brute force and compared to our heuristics. Global optimum convergence for KPNs has not been reported before. We show that 35% of optimization runs reach within 5% of the global optimum. In one of the selected problems global optimum is reached in as many as 37% of optimization runs. Results show large variations between KPNs generated with different parameters. Cyclic graphs are found to be harder to parallelize than acyclic graphs.**

I. Introduction

An efficient multiprocessor SoC (MPSoC) implementation requires automated exploration to find an efficient HW allocation, task mapping and scheduling [1]. Heterogeneous MPSoCs are needed for low power, high performance, and high volume markets [2]. The central idea in MPSoCs is to increase performance and energy-efficiency. This is achieved by efficient communication between cores and keeping clock frequency low while providing enough parallelism.

Mapping means placing each application component to some processing element (PE). Scheduling means determining execution timetable of the application components on the platform. A large design space must be pruned systematically, since the exploration of the whole design space is not feasible [1]. Fast optimization procedure is desired in order to cover reasonable design space. However, this comes with the expense of accuracy. Iterative optimization algorithms evaluate a number of application mappings for each resource allocation candidate. The application is simulated for each mapping to evaluate the cost of a solution. The cost may depend on multiple factors, such as execution time, energy consumption and silicon area constraints etc. Figure I(a) shows the mapping process.

We present an experiment where a set of applications modeled as Kahn Process Networks (KPNs) [3] are mapped on MPSoCs that have 2 to 4 PEs connected with dual shared bus. Figure I(b) shows a conceptual view of the application, its mapping, and the hardware platform. The application is optimized (mapped) for each architecture with respect to the application execution time. Resulting execution and optimization time values are compared to other mapping methods and global optimum solutions. Global optimum solutions are found by brute force search for small KPNs. We have not seen a similar comparison in any paper. It is found that KPN and architecture structure has a significant impact on optimization.

II. Related Work

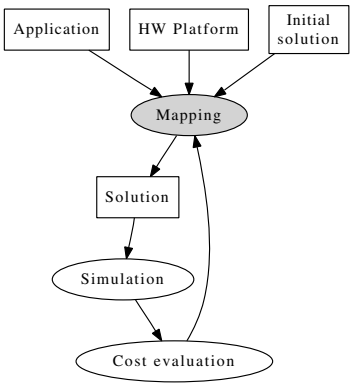
A. Kahn Process Networks (KPNs)

KPN is a distributed model of computation where a directed graph models communicating processes that can be mapped freely to PEs. Nodes in the graph do computation. Edges are communication links that are unbounded FIFOs. Each node executes a program that can read from its incoming FIFOs and write to outgoing FIFOs. There is no restriction to what the process may compute. Our earlier work used Static Task Graphs (STGs) [4]. STG is a special of KPN where the graph is acyclic, and each node does only one read-compute-write cycle, in that order.

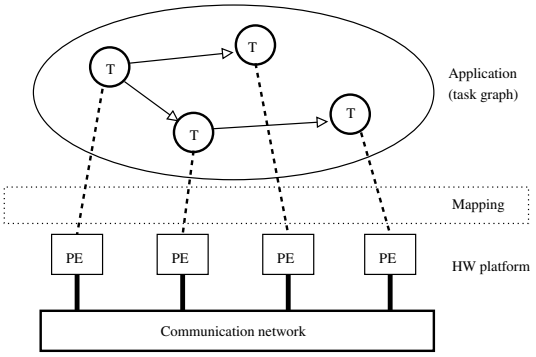
B. Algorithms For Task Mapping

Architecture exploration needs automatic tuning of optimization parameters for architectures of various sizes. Without scaling, algorithm may spend excessive time optimizing a small system, or the solution is sub-optimal for a large system. Wild *et al.* [5] compared SA, Tabu Search (TS) [6] and various other algorithms for task distribution. The parameter selection for SA had geometric annealing schedule that did not consider application or system architecture size, and thus did not scale up to bigger problems without manual tuning of parameters.

Braun *et al.* [7] compared 11 optimization algorithms for task distribution. TS outperformed SA in [5], but was worse in [7], which can be attributed to different parameter selection used. Braun's method has a proper initial temperature selection



(a) Optimization process. Boxes indicate data. Ellipses indicate operations.



(b) An example MPSoC that is optimized. The system consists of the application and the HW platform. T denotes a task, and PE denotes a processing element.

Fig. 1. Diagram of the optimization process and the system that is optimized

for SA to normalize transition probabilities, but their annealing schedule does not scale up with application or system size, making both [5] and [7] unsuitable for architecture exploration.

Our earlier papers have surveyed SA for task distribution [8], devised a method for automatically determining SA parameters [9] for distributing STGs, optimized memory consumption in distributing STGs [10], compared SA to other algorithms with respect to its convergence proprties [11], and optimized task graphs for heterogeneous architectures [12].

This paper extends and improves the method presented in [9]. The transition from STGs to KPNs requires changes to the parameter selection method that determines an efficient annealing schedule. Also, the method is improved by doing sensitivity analysis for processes. Processes with lowest execution times are ignored in the temperature range calculation to save optimization effort. Number of mapping iterations becomes smaller (optimization time is saved) but quality of results is not significantly reduced. Finally, this paper presents comparison to brute force global optimums that does not exist in earlier work. The presented method can be applied with general process networks, not just KPNs in this experiment.

```
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 ← 0 to ∞
 7  do T ← TEMPERATURE(T_0, i)
 8     S_new ← MOVE(S, T)
 9     C_new ← COST(S_new)
10     ΔC ← C_new − C
11     if ΔC < 0 or RANDOM() < ACCEPT(ΔC, T)
12        then if C_new < C_best
13                then S_best ← S_new
14                     C_best ← C_new
15             S ← S_new
16             C ← C_new
17             R ← 0
18        else if T ≤ T_f
19                then R ← R + 1
20                     if R ≥ R_max
21                        then break
22  return S_best
```

Fig. 2. Pseudo-code of the Simulated annealing algorithm. Bolded functions are modified for the presented parameter selection method.

C. Simulated Annealing

SA is a probabilistic non-greedy algorithm [13] that explores search space of a problem by moving from a high to a low temperature state. At each temperature level, SA moves one or several tasks to different PEs and evaluates the cost of new mapping solutions for each move. The algorithm always accepts a move into a better state, but also into a worse state with a changing probability. This probability decreases along with the temperature, and thus the algorithm becomes greedier. The algorithm terminates when the final temperature is reached and sufficient number of consecutive moves have been rejected.

Fig. 2 shows the pseudo-code of the SA algorithm used with the new method for parameter selection. Implementation specific issues compared to the original algorithm are explained in Section III. The $Cost$ function evaluates execution time of a specific mapping by calling the scheduler. S_0 is the initial mapping of the system, T_0 is the initial temperature, and S and T are current mapping and temperature, respectively. **Temperature** function computes a new temperature as a function of initial temperature T_0 and iteration i. R is the number of consecutive rejects. $Move$ function moves a random task to a random PE, different than the original PE. $Random$ function returns an uniform random value from the interval $[0, 1)$. **Accept** function computes a probability for accepting a move that increases the cost. R_{max} is the maximum number of consecutive rejections allowed after the final temperature has been reached.

III. The Parameter Selection Method for KPNs

The parameter selection method defines $Temperature$ and $Accept$ functions for the pseudo-code presented in Figure 2. This creates an efficient annealing schedule with effective transition probabilities. We call this method *Simulated annealing with automatic temperature (SA+AT)*.

A. Temperature Function

$Temperature$ function is chosen so that annealing schedule length is proportional to application and system architecture size. Moreover the initial temperature T_0 and final temperature T_f must be in the relevant range to affect acceptance probabilities efficiently. The method uses

$$Temperature(T_0, i) = T_0 * q^{\lfloor \frac{i}{L} \rfloor}, \quad (1)$$

where L is the number of mapping iterations per temperature level, q is geometric temperature scaling factor, and i is the current iteration count. $\lfloor \frac{i}{L} \rfloor$ means round down. Temperature T is multiplied by q every L iterations. We use $q = 0.95$. It has been found suitable in our earlier papers [9][11][8]. Determining proper L value is important to assign more iterations for larger applications and systems. This method uses

$$L = N(M - 1), \quad (2)$$

where N is the number of tasks and M is the number of PEs in the system. Also, termination condition $R_{max} = L$.

When T_0 and T_f are known, the total number of iterations i_{total} is approximately

$$i_{total} \sim \frac{\log \frac{T_f}{T_0}}{\log q} L \quad (3)$$

Each iteration must determine the cost of mapping which is the most time consuming operation in optimization because it is done by simulation. Therefore, it is important to minimize the number of temperature levels by annealing only at efficient temperature range. Also, too low a value for L will result in poor optimization result, and hence a delicate trade-off is needed.

B. Accept Function

Moving tasks to different PEs affects the cost C (execution time in this case) of the system. The absolute cost value is case-dependent. Therefore the cost change ΔC is normalized in the acceptance function by a factor $C_0 = Cost(S_0)$, which is the initial cost of the non-optimized system. Relative cost $\Delta C_r = \frac{\Delta C}{C_0}$ adapts to problems that have different process execution times and makes the algorithm more general purpose. The accept function is defined as

$$Accept(\Delta C, T) = \frac{1}{1 + exp(\frac{\Delta C}{0.5 C_0 T})}.$$

This puts relevant transition probabilities into temperatures range $(0, 1]$ so that temperatures are more comparable between different problems. As the temperature decreases the accepted cost changes less chaotically and the algorithm becomes greedier.

C. Determining Temperature Upper And Lower Bounds

The initial temperature is chosen by

$$T_0 = \frac{k t_{max}}{t_{minsum}}, \quad (4)$$

where t_{max} is the maximum execution time for any task on any PE, t_{minsum} the sum of execution times for all tasks on the *fastest* PE in the system, and $k \geq 1$ is a constant that gives a temperature safety margin. Section V-A will show that $k = 2$ is sufficient in our experiment. A proper range for k is in $[1, 3]$. Mathematical properties of k is discussed in more detail in [8]. The rationale is choosing an initial temperature where the biggest single task will have a fair transition probability of being moved from one PE to another. Section V-A will show that efficient annealing happens in the temperature range predicted by the method. The chosen final temperature is

$$T_f = \frac{t_{min}}{k t_{maxsum}}, \quad (5)$$

where t_{min} is the minimum execution time for any task on any PE and t_{maxsum} the sum of execution times for all tasks on the *slowest* PE in the system. Derivation of Equations (4)(5) is explained in [8] (Orsila case). T_0 and T_f are inside range $(0, 1]$.

Execution times t_{min} and t_{max} should be determined by profiling or analyzing the KPN. To determine t_{min}, execute the KPN on the *fastest* PE available. Record execution time $t_i > 0$ for each process, i.e. exclude processes that are not executed, and compute $t_{minsum} = \sum t_i$. Sort execution times t_i in increasing order, drop the lowest r percent of values, then set t_{min} equal the lowest remaining value. Dropping the lowest r percent of execution times excludes a set of processes whose execution time is at most proportion $\frac{r}{100}$ of total execution time. This sensitivity analysis is done to save optimization iterations. It is an improvement over the method presented in [9]. We use $r = 5$. Setting $r = 0$ is a safe choice, but it often yields longer annealing schedules and does not improve the solution.

Consider a KPN where one process executes for only a microsecond, and others a millisecond. Ignoring the microsecond process scales T_f up by 1000 saving many temperature levels and optimization effort (3). However, not ignoring it would add $\frac{\log \frac{1}{1000}}{\log q} = 135$ temperature levels of optimization effort, or evaluating $135L$ mappings. Effort of evaluating a single mapping depends on the simulation model. Evaluating a single mapping takes only a fraction of a second in this paper's experiment, but others may use instruction set simulators that take minutes to evaluate a single mapping. Optimizing the placement of the microsecond process is not significant for execution time, but it could double the optimization effort.

Execution time t_{max} is computed in a similar way. Execute the KPN on the *slowest* PE available and record execution time $t_i > 0$ for each process, and compute $t_{maxsum} = \sum t_i$. Set $t_{max} = \max t_i$.

If execution times are non-deterministic, run the profiling several times, and choose the lowest T_f and highest T_0. Alternatively, it is possible to increase the value of k.

Choosing initial and final temperature properly saves optimization iterations. On a high temperature, the optimization is practically *Monte Carlo* optimization because it accepts moves to worse positions with a high probability. And thus, it will converge very slowly to optimum because the search space size is in $O(M^N)$. Also, too low a probability reduces the annealing to greedy optimization. Greedy optimization becomes useless after a short time because it cannot escape local minimums.

IV. Experiment Setup

A. Algorithms

Five optimization algorithms were used to optimize mappings on several architectures. The goal is to minimize KPN's execution time with least mapping iterations.

1) SA+AT presented in Section III
2) SA+ST is the same as SA+AT but uses static temperature bounds: $T_0 = 1.0000$ and $T_f = 0.0001$. That is, automatic temperature range selection is not used.
3) Brute force search is only used for 16 node KPNs due to $O(N^M)$ mapping space size. This gives global optimum results that are compared to SA+AT.
4) *Group migration* [14] is a greedy deterministic clustering algorithm
5) *Random mapping* chooses a random PE for all processes at each iteration. As new solutions do not depend on previous iterations, random mapping is completely unsystematic. It is the simplest non-greedy stochastic heuristics that only reveals inherent parallelism in the problem. Any algorithm should be better than random mapping to justify its existence. Ironically, we got hit by this in convergence experiment in Section V-B.

B. Simulated HW Architecture

Several experiments were run by simulating 3 MPSoC architectures. They differ only in the number of PEs. Parameters of simulated architectures are listed in Table I. The architecture is a message passing system where each PE has some local memory, but no shared memory. Each PE and interconnection resource is available for a single action at a time. PEs are interconnected with two shared buses that are independently and dynamically arbitrated. Shared bus contention, latency and throughput set limits on performance. The system uses an event based time-behavior level simulator that rolls a timewheel. The timewheel uses continuous time with 64 bit floating point accuracy. Execution times for instructions in processes come from the KPN model. A process is blocked until it gets the input that it requests. Each PE has processes that are scheduled in the order that they become ready for execution (FIFO). Shared bus messages are queued in FIFO order. Figure I(b) presents a conceptual view of the architecture.

C. Generated Kahn Process Networks

KPNs were generated with kpn-generator [15] snapshot 2009-01-28. Table II lists parameters for kpn-generator. Acyclic and cyclic directed graphs (KPNs) were generated.

TABLE I
System Architecture Parameters

Parameter	Value
Number of PEs	2 - 4
PE frequency	300 MHz
Number of buses	2
Bus frequency	200 MHz
Bus type	Shared bus, 32 bits wide, 8 cycle arbitration, arbitration policy: FIFO, either bus that can be acquired first

TABLE II
Kahn Process Networks. N is the number of nodes. (*) indicates a sum value for the whole KPN. x is the target distribution value for KPN-generator.

Parameter	Value
N	16, 32, 64, 128 and 256
KPN categories	T1: 50 acyclic graphs with $x = 100\%$, T2: 50 cyclic graphs with $x = 100\%$, T3: 50 cyclic graphs with $x = 10\%$
For each category	10×16 node graphs, 10×32, ... 10×256, totaling 50 graphs for each KPN category
Computation cycles (T)	$2^{13}N$ (*)
Computation events (C)	$8N$ (*)
Communication bytes (S)	$2^{14}N$ (*)
b model value	0.7 for computation time and communication size randomization
kpn-generator parameters	-n N -c C -t T -s S –target-distribution=x

Acyclic graphs are such that there is no directed path from a node back to itself. If this criterion is not met, the graph is cyclic. Cyclic graphs have feedback, and therefore, they are closer to real applications than acyclic graphs.

Target distribution defines the maximum number of write target nodes for each node. Target distribution 10% means a node can only have directed edges to $1 + round(0.1N)$ processes. Lower value means more regular structure. Targets are uniformly randomized. 100% target distribution is very uncommon in real applications. A low target distribution value is more realistic.

Each KPN process is a program that consists of 3 kinds of instructions: reads, computation events that consume CPU cycles, and writes. Arbitrary flow control is allowed inside programs. Therefore, KPN is a general model of computation. The total sum of computation cycles and communication sizes (writes) was generated by a b model that is a random number sequence generator that produces *self-similar fractal-like* patterns [16]. Default value $b = 0.7$ was used.

Varied KPN parameters are number of nodes N, target distribution, cyclicity of graphs, total execution time, total communication size and the number of computation events. There are 3 KPN categories: T1, T2 and T3. Each category has 50 KPNs, totaling 150 KPNs.

D. Setup A and B

Experiments were done with two setups: A and B. Setup A uses category T1 and T2 KPNs from Table II and architecture from Table I. Setup B uses category T1 and T3 KPNs from Table II and the architecture from Table I with the modification that each byte that is sent across the bus also costs one cycle

TABLE III
AUTOMATIC TEMPERATURE: SA+AT VS SA+ST SPEEDUP VALUES WITH SETUP A

	2 PEs SA+AT	2 PEs SA+ST	3 PEs SA+AT	3 PEs SA+ST	4 PEs SA+AT	4 PEs SA+ST
Min	1.662	1.673	2.004	2.008	2.065	2.076
Mean	**1.923**	**1.928**	**2.458**	**2.465**	**2.585**	**2.588**
Std	0.073	0.071	0.152	0.150	0.101	0.098
Med	1.948	1.951	2.486	2.493	2.628	2.629
Max	2.022	2.046	2.665	2.678	2.777	2.770

TABLE IV
AUTOMATIC TEMPERATURE: SA+AT VS SA+ST NUMBER OF MAPPINGS WITH SETUP A

	2 PEs		3 PEs		4 PEs	
	SA+AT	SA+ST	SA+AT	SA+ST	SA+AT	SA+ST
Min	470	2 900	960	5 800	1 590	8 690
Mean	6 500	17 960	13 340	35 910	26 150	53 900
Std	6 540	15 810	13 180	31 610	27 110	47 470
Med	3 530	11 590	7 390	23 170	13 800	34 760
Max	25 150	46 340	51 550	92 930	100 450	140 510

TABLE V
AUTOMATIC TEMPERATURE PARAMETERIZATION VALUES FOR SA+AT WITH SETUP B. MEAN TLEVS IS THE MEAN NUMBER OF TEMPERATURE LEVELS. MAX ITERS. IS OBTAINED FROM EXPERIMENT RESULTS.

	2 PEs		3 PEs		4 PEs	
N	16	256	16	256	16	256
L	16	256	32	512	48	768
T_0 mean	0.2443	0.0379	0.2443	0.0379	0.2443	0.0379
T_f mean	0.0223	0.0012	0.0223	0.0012	0.0223	0.0012
Mean Tlevs	47	67	47	67	47	67
Max iters	1 510	29 420	3 060	59 730	4 840	92 080

for the CPU. The motivation for difference between setup A and B is explained in Section V-B. In both setups SA+AT and SA+ST algorithms were run 10 times independently for each KPN and architecture combination. Initially all nodes are mapped to a single PE.

E. Software

The optimization software and simulator was written in C language and executed on a GNU/Linux cluster of 9 machines, each machine having a 2.8 GHz x86 processor and 1 GiB of memory. Jobs were distributed to a cluster with *jobqueue* [17]. A total of $5.23 \cdot 10^8$ mappings was evaluated in 66.9 computation days leading to average of $90 \frac{mappings}{s}$.

V. EXPERIMENTS AND RESULTS

Three experiments were performed.

A. Automatic Temperature Experiment

This experiment compares speedups obtained with SA+AT and SA+ST for 2, 3 and 4 PEs with setup A and B. The purpose is to study the trade-off between optimization time and optimality of the results.

SA+AT uses dynamic temperature range that is analyzed from the KPN. SA+ST uses a static (non-automatic) temperature range $[0.0001, 1]$. With $q = 0.95$ this yields 180 temperature levels for SA+ST, whereas SA+AT uses approximately half the number of levels. For 2 PEs the $L = N(M-1)$ range is $[16, 256]$, 3 PEs $[32, 512]$ and 4 PEs $[48, 768]$, regardless of the temperature selection method,

Table III shows average speedup values for each case. In the 2 case, SA+AT has 1.923 mean speedup, and SA+ST has 1.928. SA+ST yields only 0.3% larger speedup which is negligible. The 3 PE case also has a 0.3% difference. The 4 PE case has only 0.1%. TableIV shows the number of mappings for each case. For 2 PE case SA+AT uses only 36% of SA+ST iterations. For 3 PE and 4 PE cases the same values are 37% and 49%, respectively. Results for setup B are similar.

Therefore, results indicate that automatic temperature method does not underestimate temperature bounds. A negligible speedup loss of 0.3% is observed but at least half the iterations are saved.

Table V shows values that were generated by the parameterization method. Max iterations was obtained from experiment results.

Figure 3 shows the effect of L to speedup and number of iterations for 4 PEs 256 node cyclic T3 KPNs from setup B. The parameterization method selects $L = 768$ which yields 1.73 speedup, $L = 1024$, 2048, 4096 yield 1.74, 1.76 and 1.80, respectively. Setting $L = 4096$ multiplies number of iterations by 6.5 and increases speedup only by 4.1%. The default L is efficient, but well performing.

B. Convergence Experiment

This experiment compares convergence properties of SA+AT, Group migration and random mapping.

Figure 4 shows SA+AT, Group migration and random mapping speedup convergence plotted as a function of mapping iterations. Speedup is the non-optimized execution time divided by the optimized execution time. Higher is better. The system being optimized has 4 PEs and category T2 KPNs (setup A). The first 1000 iterations are omitted for clarity.

Both SA+AT and random mapping reach average speedup 2.56. Group migration reaches only 2.39 (a greedy algorithm

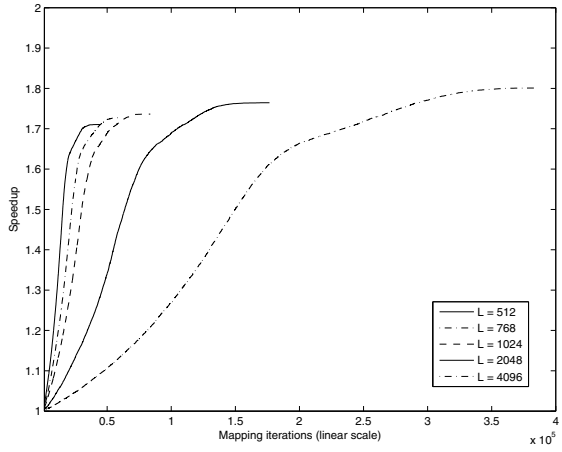


Fig. 3. SA+AT convergence: Setup B: Speedup and the number of iterations as a function of L for 4 PEs 256 node T3 KPNs. The method selects $L = 768$. Both total number of iterations and maximum speedup grow with L. In the leftmost line $L = 512$, rightmost $L = 4096$.

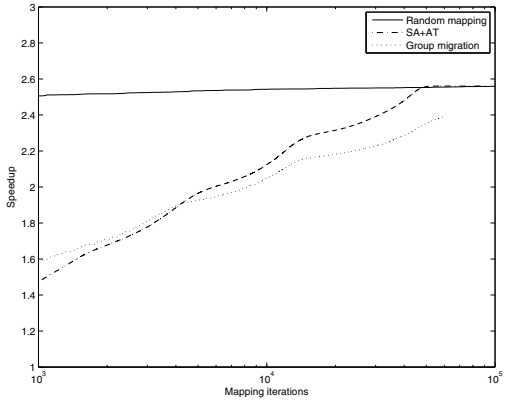


Fig. 4. Convergence: Setup A: Average speedup plotted against mapping optimization iterations. The system being optimized has 4 PEs and category T2 KPNs. Curiously all algorithms reach practically the same speedup. Speedup is computed as the non-optimized execution time divided by the optimized execution time. Higher is better.

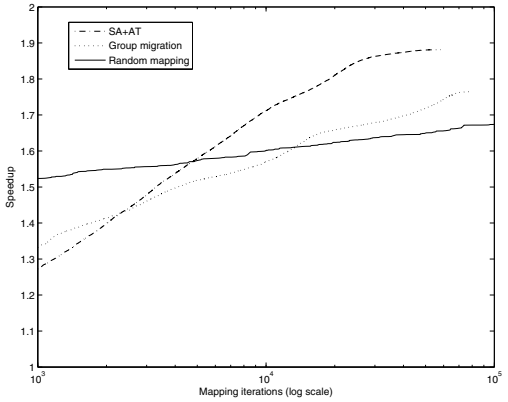


Fig. 5. Convergence: Setup B: Average speedup plotted against mapping optimization iterations for 4 PEs.

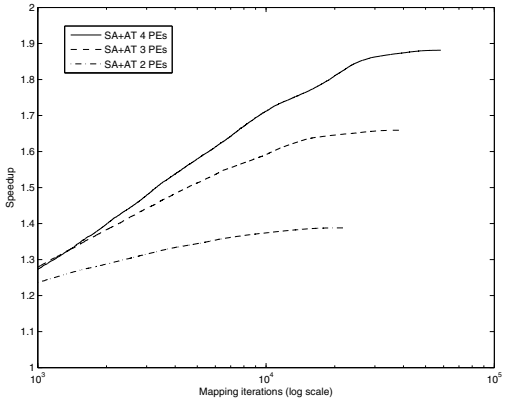


Fig. 6. Convergence: Setup B: Average speedup plotted against mapping optimization iterations for 2, 3 and 4 PEs.

cannot overcome local minimums), which is 6.6% worse than SA+AT and random. Random mapping being as good as SA+AT seems odd. Random mapping has been inferior to SA+AT in all other tests we have seen, which makes this an interesting case. Random mapping should converge very slowly as all iterations are independent of each other, no systematic techniques (such as local search) or memory is used, unlike in SA+AT. SA+AT reached 2.56 at $49\ 300$ iterations. Random mapping converged very rapidly near the maximum. The second iteration already had 2.22 speedup. At 1000 iterations it had 2.51 which is only 2% less than the best solution.

Next we tried category T3 KPNs with 4 PEs. These KPNs differ from previous case by having target distribution 10% which forces more organization into the graph. For example, 256 node graphs can have 27 distinct targets, 16 node graphs can have only 3 targets. Also, we added a CPU cost of 1 cycle per byte for writes from one PE to another. Statistically this encourages PE locality to nodes that interact with each other. Both of these changes, independently or together, changed results in favor of SA+AT over random mapping.

Convergence is shown in Figure 5. SA+AT reaches average speedup 1.88, Group migration 1.76 and random mapping 1.68. Now random mapping is the worst. Much lower random mapping speedups indicate that there is little easy parallelism. Random mapping cannot do local search, and now even the greedy Group migration outperforms it. This is due to increased organization in the graph that encourages PE locality between nodes that interact. We believe previous test with setup A had too little systematic optimization opportunities, and therefore random was equally good as SA+AT, but in this case SA+AT wins because of reparameterization. In both cases SA+AT has better speedup than Group migration.

SA+AT and Group migration show similar convergence between Figures 4 and 5, but random mapping does not.

Figure 6 shows convergence for SA+AT on 2, 3 and 4 PEs. Speedups are averaged: 4 PEs has 1.81, 3 PEs 1.58 and 2 PEs 1.34. There is not much parallelism available, but convergence looks similar in each case. It should be noted that the number of optimization iterations grows with the number of PEs. It is a consequence of our parameterization method that the number of iterations per temperature level is $L = N(M-1)$.

C. Brute Force Experiment

This experiment compares SA+AT heuristics to global optimum solutions obtained by brute force search. Brute force experiment parameters are listed in Table VI. The experiment is run for setup A and B with 16 node KPNs. We were unable to run brute force search with available computational capacity for problems larger than 16 nodes or more than 4 PEs. Without loss of generality, one of the 16 nodes was fixed to a PE to decrease optimization iterations. The brute force search space size becomes $2^{15} \sim 3.2E4$ and $3^{15} \sim 1.4E7$ mappings for 2 and 3 PE cases, respectively. A heuristic random algorithm does not always have the same results. Hence, SA+AT is

TABLE VI
BRUTE FORCE EXPERIMENT PARAMETERS

Parameter	Value
Algorithms	SA+AT, brute force
KPNs	16 node setup A KPNs from T1 and T2 $(10+10)$, 16 node setup B KPNs from T1 and T3 $(10+10)$,
Architectures	2 and 3 PEs from setups A and B
SA+AT runs	2 setups $\times 2$ archs $\times 20$ graphs $\times 1000$ independent optimization runs = 80 000 SA+AT runs

TABLE VII
BRUTE FORCE VS. SA+AT WITH SETUP A (100% TARGET DISTRIBUTION): PROPORTION OF SA+AT RUNS THAT CONVERGED WITHIN p FROM GLOBAL OPTIMUM

Exec. time overhead	Proportion of runs within limit p			
	2 PEs		3 PEs	
$p = \frac{t}{t_o} - 1$	acyclic	cyclic	acyclic	cyclic
+0%	0.043	0.033	0.004	0.002
+1%	0.104	0.085	0.017	0.014
+2%	0.287	0.268	0.090	0.051
+3%	0.547	0.458	0.274	0.120
+4%	0.770	0.647	0.476	0.231
+5%	0.892	0.836	0.678	0.392
+6%	0.948	0.936	0.842	0.593
+7%	0.976	0.987	0.941	0.776
+8%	0.995	0.999	0.983	0.913
+9%	1.000	1.000	0.997	0.975
+10%	1.000	1.000	1.000	0.995
+11%	1.000	1.000	1.000	0.999
+12%	1.000	1.000	1.000	1.000
mean mappings	748	790	1774	1754
median mappings	760	756	1759	1736

run independently 1000 times for each KPN. The results are recorded and compared to global optimum.

The optimality of SA+AT is shown in Tables VII and VIII. Tables show the proportion of 1000 SA+AT runs that got execution time $t \leq (1+p)t_o$, where p is the execution time overhead compared to global optimum execution time t_o. $p = 0\%$ means the global optimum result. Optimum values (mappings) were obtained by brute force search. The last two rows show mean and median values for the number of mappings tried in a single SA+AT run. The results are shown for 2 and 3 PEs with 16 node acyclic and cyclic Kahn Process Networks. Values from Table VIII are plotted in Figure 7. The difference between Tables VII and VIII is the setup, the former uses setup A and the latter uses setup B. Setup A has uniform target distribution (100%). Setup B graphs are more organized (target distribution 10%) and have less free parallelism. The number of needed iterations (i.e. optimization time) is shown in Tables IX and X.

Uniform target distribution (setup A) has an interesting property that the fewer optimization runs have optimum solution ($p = 0\%$) than in setup B, but all solutions in setup A are closer to global optimum. For example, in setup A 3 PE cyclic case, global optimum was reached in 2 out of 1000 SA+AT runs. The same value is 111 for setup B. However, all solutions came within 12% of optimum cost in setup A, but 28% in setup B. Therefore, it is easier to reach global optimum in setup B, but the variance is higher. Moreover, the mean number of mappings to reach global optimum was reduced from 923 150 to 15 230 which is 98% reduction due to 10% target distribution (Table IX and X). Brute force search for setup B 3 PE cyclic graphs took $14.3M$ mappings, but SA+AT took 15 230 mappings on average, which means 99.9% reduction.

TABLE VIII
BRUTE FORCE VS. SA+AT WITH SETUP B (10% TARGET DISTRIBUTION): SEE TABLE VII

Exec. time overhead	Proportion of runs within limit p			
	2 PEs		3 PEs	
$p = \frac{t}{t_o} - 1$	acyclic	cyclic	acyclic	cyclic
+0%	0.358	0.374	0.056	0.111
+1%	0.418	0.435	0.072	0.160
+2%	0.540	0.478	0.101	0.223
+3%	0.612	0.543	0.173	0.275
+4%	0.685	0.622	0.271	0.349
+5%	0.792	0.650	0.348	0.435
+6%	0.856	0.716	0.446	0.493
+7%	0.884	0.730	0.538	0.554
+8%	0.918	0.756	0.623	0.609
+9%	0.940	0.789	0.722	0.692
+10%	0.962	0.817	0.801	0.747
+11%	0.969	0.846	0.867	0.785
+12%	0.980	0.886	0.913	0.820
+13%	0.992	0.922	0.945	0.868
+14%	0.996	0.951	0.967	0.890
+15%	0.997	0.960	0.981	0.907
...	...	...	...	...
+18%	1.000	0.984	0.998	0.953
+20%		0.992	1.000	0.975
+25%		0.999		0.997
+26%		1.000		0.998
+28%				1.000
mean mappings	795	824	1637	1692
median mappings	778	788	1610	1579

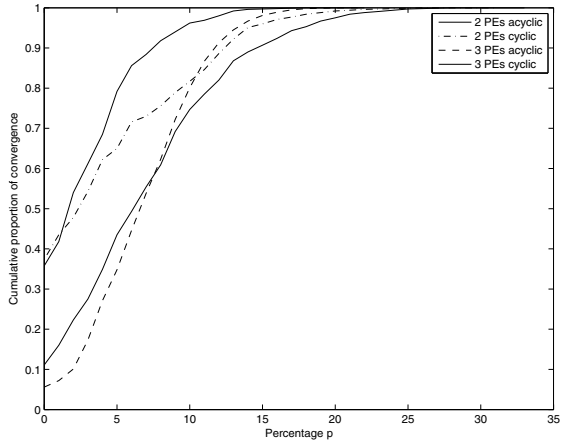


Fig. 7. Brute force vs SA+AT with setup B: Values plotted from Table VIII. 2 PEs acyclic converges fastest, 3 cyclic is the slowest.

Limited target distribution (setup B) behaves quite differently w.r.t. optimization result. The optimal result is more likely achieved, e.g. 35.8% runs for 2 PEs and acyclic graphs. In contrast, the guarantees for optimality are weaker as the worst runs may have overhead of 18-28%. The number of iterations are comparable to setup A except for $p = 0\%$.

Larger architecture naturally makes mapping more difficult. This happens in two ways: the optimal result is less likely found with SA; probability drops by a factor of $3.4\times$-$16.5\times$

TABLE IX
BRUTE FORCE VS. SA+AT WITH SETUP A: APPROXIMATE NUMBER OF MAPPINGS TO REACH GLOBAL OPTIMUM VALUE TIMES $1 + p$

	Number of mappings					
	2 PEs			3 PEs		
p	acyclic SA+AT	cyclic SA+AT	Brute force	acyclic SA+AT	cyclic SA+AT	Brute force
+0%	17 270	24 150	32 770	506 770	923 150	14.3M
+1%	7 220	9 350	...	102 530	128 970	...
+2%	2 600	2 950		19 640	34 390	
+3%	1 370	1 720		6 480	14 600	
+4%	970	1 220		3 720	7 610	
+5%	840	940		2 620	4 480	
+6%	790	840		2 110	2 960	
+7%	770	800		1 890	2 260	
+8%	750	790		1 800	1 920	
+9%	750	790		1 780	1 800	
+10%	$\leq$ 750	$\leq$ 790		1 770	1 760	
+11%	$\leq$ 750	$\leq$ 790		$\leq$ 1 770	1 760	
+12%	$\leq$ 750	$\leq$ 790		$\leq$ 1 770	1 750	

TABLE X
BRUTE FORCE VS. SA+AT WITH SETUP B: SEE TABLE IX

	Number of mappings					
	2 PEs			3 PEs		
p	acyclic SA+AT	cyclic SA+AT	Brute force	acyclic SA+AT	cyclic SA+AT	Brute force
+0%	2 220	2 200	32 770	29 030	15 230	14.3M
+1%	1 900	1 890	...	22 860	10 590	...
+2%	1 470	1 730		16 270	7 610	
+3%	1 300	1 520		9 450	6 170	
+4%	1 160	1 330		6 030	4 850	
+5%	1 000	1 270		4 700	3 890	
+6%	930	1 150		3 670	3 430	
+7%	900	1 130		3 040	3 060	
+8%	870	1 090		2 630	2 780	
+9%	850	1 040		2 270	2 450	
+10%	830	1 010		2 040	2 270	
+11%	820	980		1 890	2 160	
+12%	810	930		1 790	2 070	
+13%	800	890		1 730	1 950	
+14%	800	860		1 690	1 900	
+15%	$\leq$ 800	860		1 670	1 870	
...	...	...		...	...	
+18%		840		1 640	1 780	
+20%		830			1 740	
+25%					1 700	
+26%					1 700	
+28%					1 690	

and iteration count increases $6.9\times$-$38.2\times$. The overhead p of the worst runs increases by few units: setup A from 9% to 12%, setup B from 26% to 28%.

A clear difference was found between KPN categories. Cyclic graphs are harder to map than acyclic. Also, lower target distribution is easier to map than high. We believe this is due to increased dependency of many nodes that increases the effect of a change in mapping.

The results indicate that SA+AT performs quite well; there is 91% probability to find mapping with $p \leq 15\%$, 75% probability to find mapping with $p \leq 10\%$ and 35% probability to find mapping with $p \leq 5\%$.

Also, note that mean and median mappings are almost constant due to parameterization. Also, saving a significant proportion of mapping iterations may only have a small effect on p. Unfortunately, there is no way to tell global optimum in termination condition without brute force search.

VI. CONCLUSIONS

Distributing Kahn Process Networks (KPNs) onto multiprocessor SoCs was analyzed. A Simulated annealing method to optimize process distribution was presented and validated. No significant loss of quality was found in solutions but over half the optimization time was saved.

Results were compared with global optimum solutions obtained from brute force search. It was found that near global optimum are frequently obtained, while in some cases as many as 37% of solutions reach global optimum. We are unaware of such results being published for KPNs. It is possible to save significant proportion of optimization effort while losing little in solution quality.

Future work should verify the method with real applications modeled as KPNs. Also, parameter generated KPNs should be studied to model performance characteristics of real applications to ease design space exploration.

REFERENCES

[1] M. Gries, *Methods for evaluating and covering the design space during early design development*, Integration, the VLSI Journal, Vol. 38, Issue 2, pp. 131-183, 2004.

[2] W. Wolf, *The future of multiprocessor systems-on-chips*, Design Automation Conference 2004, pp. 681-685, 2004.

[3] G. Kahn, *The semantics of a simple language for parallel programming*, Information Processing, pp. 471-475, 1974.

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

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

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

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

[8] H. Orsila, E. Salminen, T. D. Hämäläinen, Chapter in book "Simulated Annealing", ISBN 978-953-7619-07-7, *Best Practices for Simulated Annealing in Multiprocessor Task Distribution Problems*, I-Tech Education and Publishing KG, pp. 321-342, 2008.

[9] H. Orsila, T. Kangas, E. Salminen, T. D. Hämäläinen, *Parameterizing Simulated Annealing for Distributing Task Graphs on multiprocessor SoCs*, Symposium on SoC, Nov 14-16, pp. 73-76, 2006.

[10] H. Orsila, T. Kangas, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, *Automated Memory-Aware Application Distribution for Multi-Processor System-On-Chips*, Journal of Systems Architecture, Elsevier, 2007.

[11] H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, *Optimal Subset Mapping And Convergence Evaluation of Mapping Algorithms for Distributing Task Graphs on Multiprocessor SoC*, Symposium on SoC, 2007.

[12] H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, *Evaluation of Heterogeneous Multiprocessor Architectures by Energy and Performance Optimization*, Symposium on SoC, Nov 4-6, 2008.

[13] S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi, *Optimization by simulated annealing*, Science, Vol. 200, No. 4598, pp. 671-680, 1983.

[14] 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.

[15] *kpn-generator*: http://zakalwe.fi/~shd/foss/kpn-generator/

[16] R. Thid, I. Sander, A. Jantsch, *Flexible bus and noc performance analysis with configurable synthetic workloads*, DSD, pp. 681-688, 2006.

[17] *jobqueue*: http://zakalwe.fi/~shd/foss/jobqueue/