Hardware Support for Simulated Annealing and Tabu Search

Reinhard Schneider and Reinhold Weiss

[schneider | weiss]@iti.tu-graz.ac.at
Institute for Technical Informatics
Technical University of Graz, AUSTRIA

Abstract. In this paper, we present a concept of a CPU kernel with hardware support for local-search based optimization algorithms like Simulated Annealing (SA) and Tabu-Search (TS). The special hardware modules are: (i) A linked-list memory representing the problem space. (ii) CPU instruction set extensions supporting fast moves within the neighborhood of a solution. (iii) Support for the generation of moves for both algorithms, SA and TS. (iv) A solution mover managing several solution memories according to the optimization progress. (v) Hardware addressing support for the calculation of cost functions. (vi) Support for nonlinear functions in the acceptance procedure of SA. (vii) A status module providing on-line information about the solution quality. (v) An acceptance prediction module supporting parallel SA algorithms.
Simulations of a VHDL implementation show a speedup of up to 260 in comparison to an existing implementation without hardware support.

1 Introduction

Simulated Annealing (SA)[1] and Tabu-Search (TS)[2][3] are algorithms that are well suited to solving general combinatorial optimization problems which are common in the area of real-time multiprocessor systems. Tindell et al.[4] solved a standard real-time mapping task with several requirements using SA. Axelsson[5] applied SA, TS and genetic algorithms, all three based on the local search concept[6], to the problem of HW/SW Codesign. In [7] the authors introduced a complete tool for handling parallel digital signal processing systems based on parallel SA.

All research projects mentioned use, like many others, SA, TS or other algorithms based on local search to find solutions for partitioning, mapping and scheduling problems in parallel systems. The results show that these algorithms are able to solve even difficult problems with a good solutions quality. The main drawback is the slow optimization speed. This is particularly true for SA. Many researchers have tried to reduce execution time in different ways. One way is to optimize the algorithm itself, which depends strongly on the application and has a limited possible speedup[8]. Another approach is to parallelize SA[9]. With parallel simulated annealing (PSA) it is possible to achieve greater speedup, independent of the problem, without compromising solution quality[10]. PSA is

already successfully applied to multiprocessor scheduling and mapping[11]. But even with PSA on up-to-date processor hardware, it takes a very long time to compute a multiprocessor schedule for realistic system complexity. This still prevents the on-line use of SA in dynamic systems, and it is also the main reason why our research focuses on supporting SA and TS by dedicated processor hardware. It is evident that a processor supporting local search also simplifies non real-time applications using SA and TS.

Abramson[12] showed that with a custom computing machine (CCM) it is possible to outperform software implementation by several orders of magnitude. Other hardware- implementations ([13][14]) also showed a significant speedup in comparison with a software-implementation. CCMs are very efficient for the problem they are designed for. Unfortunately they can not solve other problems. Even a small change in the characteristic of the problem or an unexpected increase of the problem size means that the CCM itself has to be re-designed. Eschermann et al.[15] tried to build a more flexible processor for SA where fewer parts of the algorithm are implemented in hardware so that different problems can be solved. Unfortunately, this processor has not been developed any further. Up to now, there is no processor available that explicitly supports local search or other nature-inspired algorithms.

Our solution combines the flexibility of a programmable CPU with the speed of dedicated hardware in a flexible, modular concept.

2 Local Search

Optimization algorithms based on local search (LS) have in common that they start from any arbitrary solution and try to find better solutions by stepping through a *neighborhood* of solutions. The neighborhood of a solution i is defined by the neighborhood function $N(i)$. A real cost value c can be mapped to each solution i by a *cost function* $c(i)$. The problem is to find a globally optimal solution $i*$, such that $c* = c(i*) \leq c(i)$ for all solutions i.

Iterative Improvement. A basic version of LS is *iterative improvement*. With this technique, the neighborhood $N(i)$ is searched starting from the current solution i. Then, either the first better solution (*first improvement*) or the solution with the lowest costs within the neighborhood (*best improvement*) is chosen as the new solution. Improved techniques like SA or TS use different strategies to overcome the problem of getting caught in a local minimum. But all these techniques are based on the same few basic functions. The following pseudo-code describes the basic structure of a local-search based algorithm:

```
i=Generate-Initial-Solution
REPEAT
     Move=Select-a-Move-within-Neighborhood-N(i)       (1)
     i'=Apply-Move(i, Move)                            (2)
     dC=Compute-Change-in-Cost(i, i')                  (3)
     IF accept THEN i=i'                               (4)
UNTIL Stopping-Condition-is-true
```

Key functions of this algorithm are: (1) The selection of a *move*, which means the selection of a transition from one solution to another. (2) Performing the move to obtain the new solution. (3) Computing the difference in costs, and (4) deciding whether to accept the new solution or not. The definition of the neighborhood and the way of computing the costs depend on the problem that has to be solved. The way of selecting a solution from the neighborhood and the criteria for accepting a new state depend on the algorithm:

Simulated Annealing. The selection of a move in SA is based on a stochastic process. This means that one move is chosen at random out of all possible moves within the neighborhood. Therefore, the quality of the pseudo random number generator is important in order not to omit any solution. In SA, a move which leads to an improvement of cost is always accepted, deteriorations of costs are accepted if they fulfill the *Metropolis Criterion* - in analogy to the annealing procedure of metals.

Tabu Search. TS always searches the whole neighborhood. The best solution within the neighborhood is taken as a new solution. In order to avoid getting trapped in a local minimum, TS works with the search history: Solutions that have already been selected some time before are forbidden (taboo). These solutions or the moves that lead to these solutions, respectively, are stored in a *tabu list*. Solutions in the tabu list may still be accepted if they are extraordinary (e.g., if they are significantly better than all other solutions in the neighborhood). These solutions are also stored in a list called the *aspiration list*.

3 Hardware Support

The analysis of possible hardware support for LS-based algorithms was governed by the following objectives:

- The flexibility of the final system should be maximized.
- Hardware support should be modular so that more than one optimization algorithm can be accelerated by the same hardware.
- As many parts of the algorithms as possible should be realized in hardware.
- The final processor should support parallelization.
- The employment in real-time systems should be supported.

The first goal was achieved by designing the hardware support as a CPU kernel extension. Thus, the flexibility of a fully programmable CPU remained. Additionally, data transfer time drops out in this concept because the hardware modules directly interact with the CPU, the bus system and the main memory.

The second goal was achieved by a strict modular design. Different optimization algorithms could be supported by different combinations of the modules. This concept also satisfied the third objective, namely the realization of special, algorithm-dependent functions as modules and their integration in the system.

Parallelization techniques were analyzed only for SA. An acceptance prediction module was introduced which efficiently supports the *decision tree decomposition* algorithm[16], where the processors work in a pipelined, overlapped mode.

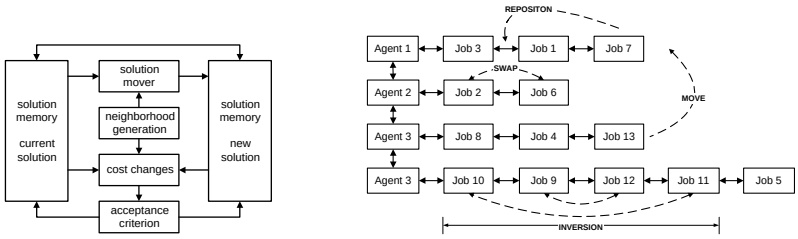

Fig. 1. Left: *Structure of a typical local search based algorithm.* **Right:** *Linked-list representation of the generalized assignment problem with fundamental instructions to generate a neighbor solution.*

In real-time systems, it is most important to *know* about the status of the optimization process. As all optimization algorithms *approach* the optimal solution, knowledge about the quality of the solution reached so far must exist in order to make a decision at a crucial time. Therefore, a statistical module was designed that continuously checks the current status of the optimization process.

Modular View. A modular view of local search based algorithms is depicted in Figure 1, left side. The basic modules are: (i) Two solution memories, (ii) a neighborhood generator, (iii) a solution mover, (iv) support for the calculation of the cost function, and (v), support for the calculation of the acceptance criterion. Modules (i)-(iii) are closely coupled, as they work on the same data structure. Module (iv) strongly depends on the problem, module (v) depends on the algorithm used. Still, there are possibilities to build hardware support for these two modules. Additional modules (an acceptance prediction module and a status module) are implemented to support parallelization and real-time systems.

Solution Memory. A fast solution memory is fundamental as the movement within the solution space is the most frequent operation in local search. In order to speed up moves, it is necessary to find an appropriate problem representation in memory. Generally, a combinatorial optimization can be described by (i) elements of different type and (ii) relations between them. E.g., the generalized assignment problem (GAP) could look as depicted in Figure 1, right side. The problem of mapping tasks to processors is a special case of the GAP problem, where the jobs represent tasks and the agents represent the processors. In this model, the elements and their content form the static description of the problem. All possible combinations of relations represent the solution space. One set of relations represents a special solution. Moving in the solution space means to change relations. Based on this definition, it is easy to define a neighborhood: A solution $i*$ is defined to be within the neighborhood $N(i)$ if the differences in relations is small (e.g. one different relation).

Linked lists[17] and matrices are efficient ways of representing relations between two types of elements. We decided to implement both a memory based on a linked-list representation, and a memory based on a matrix representation. In the special list/matrix memory (*solution memory*), it is only pointers to the

static elements in main memory that are stored. Thus, the solution memory is independent of the kind of problem solved and the size of the appropriate elements. Four basic operations (see Figure 1, right side) on the list representation allow movement in the neighborhood: (i) Moving an element means removing it from one list and appending it to an other one *(MOVE)*. (ii) An element can be reordered within a list *(REPOSITION)*. (iii) Two elements of the same list can be exchanged (*SWAP*). And finally, (iv) the order of a chain of elements can be inverted (*INVERSION*).

Neighborhood Generation. The generation of the neighborhood depends on the algorithm used. In SA, a new solution is generated at random. Therefore, a set of hardware pseudo random number generators (*PRNG*) is proposed. One of them has to chose the move, another one has to select the source element (job), the third one has to chose the destination agent and/or position in the list according to the selected move. In TS, all possible moves within the neighborhood have to be searched. The neighborhood generator has to check if a selected move is forbidden (*tabu*) or not. This is done by comparing the move with the tabu list and the aspiration list. This search is accelerated by managing the lists in hardware .

Solution Mover. As the current solution and the new solution must be stored until the acceptance decision has been taken, the linked-list memory is duplicated. The result of the acceptance decision determines which memory has to be synchronized to the other. TS also needs to store the best solution reached within the neighborhood. Additionally, the best solution reached so far is stored. This is important for real-time systems, where the optimization has to stop after a fixed time, and the best solution so far should be available. The solution mover module applies the list operation suggested by the neighborhood generator and manages all solution memories and transactions between them.

Cost Function. The cost function strongly depends on the problem solved. A function completely realized in hardware decreases flexibility dramatically. Hence, we suggest to implement only *addressing support* for the cost function: Providing an easy way to access the elements that have been affected by the move, e.g., by a list of these elements, supports in particular cost functions that can be computed incrementally. As the order and size of this list depends on the problem, we suggest to provide a *user-programmable hardware module* (e.g., an FPGA-based module) which is tightly coupled to the linked-list memory. This allows the adaptation of the sequence of elements to any individual problem before the optimization process is started.

Acceptance Criterion. In TS, the best solution found so far is always accepted. Thus, no additional hardware is needed. In SA, moves with a cost improvement are always accepted. If costs rise, SA decides on the acceptance of the move by evaluating the *Metropolis criterion* $e^{\frac{E_i - E_j}{T}} > random(0,1)$. Negative cost differences $(E_i - E_j)$ are weighted by a control parameter T and transformed by non-linear operation. The move is then accepted if the result of $e^{\frac{E_i - E_j}{T}}$ (always between 0 and 1) is greater than a random number between 0 and 1. A hardware pseudo random number generator improves performance significantly,

as the random number is provided without CPU interaction. Additionally, as the result of the exponential function is compared with a random number, no high accuracy is needed. Therefore, a hardware lookup table with pre-calculated values for each value of T is sufficient. By means of these tables, the evaluation of the exponential function is done in one cycle.

Status Information. The absolute value of the cost function can not be used as status information, because only its relation to the optimal solution is linked to the quality of the solution. But the optimal solution is not known to the system. Therefore, we use a statistical status information based on the relative cost changes.

Acceptance Prediction. The acceptance prediction module is used to support parallelization in SA. The output value corresponds to the probability of accepting a new move. With this value, a good prediction of the acceptance is available before the actual result of the acceptance criterion is available.

4 Implementation

All modules are implemented using VHDL and are synthesized for emulation on a programmable FPGA chip using Xilinx Foundation software and tools. All modules use parametric problem sizes to be easily adaptable to different systems.

Solution Memory. The module consists of four solution memories, realized as both linked list memories and matrix memories: the current solution, the new solution, the best solution found so far and the best solution in the neighborhood. The latter is used only in TS. Memory synchronization works very fast as all memories are arranged physically side by side and connected by a high speed internal bus.

Move Generator. Moves are generated in two ways: For SA, a set of pseudo random number (PRN) generators, based on cellular automata[18], automatically generates a move. These automata provide excellent PRNs every cycle with a maximum perod of 2^n. For TS, all possible moves have to be considered. These moves are generated sequentially. Each move has to be checked by a move checker . The move checker decides, with the help of the content of the tabu list and the aspiration list, if a move is accepted or not. The search within the lists is realized by parallel comparators.

Status Module. A good estimation of the current status of the optimization can be made by averaging the cost changes over the absolute costs. This only works for problems with a smooth cost function without singular minima, which is the case for mapping tasks in multiprocessor systems.

Acceptance Prediction. The acceptance prediction unit (for SA) uses an averaged cost value, the last cost differences and the last acceptance decision as input values. The output is a prediction value that indicates if the next new solution will be accepted or not. With the help of this value, the network topology of the parallelized processors can change dynamically.

5 Results

Timing results were obtained in two ways: Firstly by simulation with a VHDL simulator, and secondly by calculating the cycles needed per instruction. The time needed for one iteration strongly depends on the time to perform a move in the neighborhood (move generation, solution mover and acceptance decision) and the time needed to calculate the cost difference. The latter strongly depends on the problem and is therefore not discussed any further. The use of our hardware modules shortens the time for move generation and the acceptance decision to one cycle each. The solution mover is more critical. The time needed to perform a particular move depends on the type of memory (linked-list based or matrix-based) and the problem size, which is indicated by parameter n in Table 1.

Table 1. Timing requirements for the solution mover module.

instruction	cycles: matrix memory	cycles: list memory
swap	10	≤ 26
inversion	$n * 8 + 3$	$\leq n * 28 + 6$
remove	$n * 8 + 10$	22
reposition	$n * 8 + 2$	$\leq n * 26 + 2$

In order to assess our solution, a system was designed to solve the travelling salesman problem with SA. Simulations needed 13 cycles for one iteration. With an FPGA running at 13 MHz, the time for one instruction is $1\mu s$. A software implementation on a digital signal processor with a clock speed of 40 MHz needs $86\mu s$. The speedup of the hardware-supported solution is therefore 86 or, assuming that the hardware modules will run with the same speed if directly implemented in a CPU, the speedup will be over 260. The acceptance prediction module showed a hit rate of 90% when suspended for only 10% of the time.

6 Discussion

Nature-inspired algorithms are a fast growing field. New and improved algorithms are developed rapidly. But there is a lack of appropriate computer architectures to support these algorithms. The system described in this paper shows that with an extended CPU it is possible to speed up significantly local-search based algorithms. Even though an ASIC–prototype has to be realized first in order to verify the speedup, the simulation results are respectable. These modules are an attempt to show which functions could be supported by new, intelligent CPU cores. The costs of integrating these modules in a CPU core are small compared to the speedup they provide. The modular concept is very flexible and allows, e.g., support for parallelization. Based on this concept, a lot of new modules can be imagined: Support for other algorithms like genetic algorithm, neuronal networks, qualitative algorithms, etc. A CPU extended by such modules will probably make expensive special solutions dispensable.

References

[1] S Kirkpatrick, C D Gelatt, and M P Vecchi. Optimisation by simulated annealing. *Science*, 220:671–680, 1983.

[2] Fred Glover. Tabu search: 1. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[3] Fred Glover. Tabu search: 2. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[4] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating Hard Real–Time Tasks: An NP-Hard Problem Made Easy. *The Journal of Real–Time Systems*, (4):145–165, 1992.

[5] Jakob Axelsson. Architecture Synthesis an Partitioning of Real-Time Systems: A Comparison of Three Heuristic Search Strategies. In *5th International Workshop on Hardware/Software Codesign*, pages 161–165, March, 24-26 1997.

[6] E. Aarts and K. Lenstra. *Local Search in Combinatorial Optimization*. Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1997.

[7] Claudia Mathis, Martin Schmid and Reinhard Schneider. A Flexible Tool for Mapping and Scheduling Real-Time Applications on Parallel Systems. In *Proceedings of the Third International Conference on Parallel Processing and Applied Mathematics*, Kazimierz Dolny, Poland, September, 5-7 1999.

[8] E. H. L. Aarts and J. H. M Korst. *Simulated Annealing and Boltzmann Machines*. Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Chichester, U.K., 1989.

[9] Tarek M. Nabhan and Albert Y. Zomaya. Parallel simulated annealing algorithm with low communication overhead. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1226–1233, December 1995.

[10] Soo-Young Lee and Kyung Geun Lee. Synchronous and asynchronous parallel simulated annealing with multiple Markov chains:. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):993–1008, October 1996.

[11] Martin Schmid and Reinhard Schneider. A Model for Scheduling and Mapping DSP Applications onto Multi-DSP Platforms. In *Proceedings of the International Conference on Signal Processing Applications and Technology*. Miller Freeman, 1999.

[12] David Abramson. A very high speed architecture for simulated annealing. *j-COMPUTER*, 25(5):27–36, May 1992.

[13] J. Niittylahti. Simulated Annealing Hardware Tool. In *The 2nd International Conference on Expert Systems for Development*, pages 187–191, 1994.

[14] Bang W. Lee and Bing J. Sheu. Paralleled hardware annealing for optimal solutions on electronic neural networks. *IEEE Transactions on Neural Networks*, 4(4):588–599, July 1993.

[15] B. Eschermann, O. Haberl, O. Bringmann, and O. Seitzr. COSIMA: A Self-Testable Simulated Annealing Processor for Universal Cost Functions. In *EuroASIC*, pages 374–377, Los Alamitos, CA, 1992. IEEE Computer Society Press.

[16] Daniel R. Greening. Parallel Simulated Annealing Techniques. In *In Emergent Computation*, pages 293–306. MIT Press, Cambridge, MA, 1991.

[17] A. Postula, D.A. Abramson, and P. Logothetis. A Tail of 2 by n Cities: Performing Combinatorial Optimization Using Linked Lists on Special Purpose Computers. In *The International Conference on Computational Intelligence and Multimedia Applications (ICCIMA)*, Feb, 9-11 1998.

[18] P.D. Hortensius, R.D. McLeod, and H.C. Card. "parallel random number generation for vlsi systems using cellular automata". *IEEE Transactions on Computers*, 38(10):1466–1473, October 1989.