Task Allocation in Distributed Embedded Systems by Genetic Programming *

Allan Tengg, Andreas Klausner
Institute for Technical Informatics
Graz Univerity of Technology
A-8010 Graz, Austria
{tengg, klausner}@iti.tugraz.at

Bernhard Rinner
Institute of Networked and Embedded Systems
Klagenfurt University
A-9020 Klagenfurt, Austria
bernhard.rinner@uni-klu.ac.at

Abstract

In this paper we describe a task allocation method, that utilizes genetic programming to find a suitable solution in an adequate time for this NP-complete combinatorial optimization problem. The underlying distributed embedded system is heterogenous, consisting of different processors with different properties such as core type, clock frequency, available memory, and I/O interfaces, interconnected with different communication media. In our applications, which are described as data flow graphs, the number of tasks to be placed is much larger than the number of processors available. We highlight the difficulties when applying genetic programming to this problem and present our solutions and enhancements, accompanied with some simulation results.

1 Introduction

In our I-SENSE research project [10] we develop and investigate a scalable and embedded architecture for various multi-sensor applications based on so called embedded intelligent sensor nodes with sufficient computing and communication performance. By delegating the CPU-expensive data fusion tasks into the sensor nodes, the requirements concerning the communication bandwidth can be reduced compared to centralized data fusion architectures.

To accomplish high flexibility, we define the functional description of a data fusion system - the so called *fusion model* - almost independently of the present hardware configuration. Together with the *hardware model*, it is the objective, to automatically find a valid mapping from the *fusion model* on this specific hardware.

Solutions for those types of problems have been discussed very often in literature. Instead of trying to accommodate an existing algorithm to our specific task allocation problem, we use genetic programming to aid us in finding a feasible solution. Genetic algorithms have two very promising beneficial features: Firstly, they can be implemented rather quickly and the correctness of the implementation can be verified easily. Secondly, they can be adapted quite smoothly to modified objectives by simply changing the calculation of the fitness score - the core algorithm remains unchanged. Often a straight forward implementation of a genetic algorithm results in a poor behavior, we propose some enhancements in this publication.

The remainder of the paper is organized as follows: Section 2 gives a review about related work. Section 3 explains our system in more detail. Section 4 deals with applying genetic programming to the task allocation problem while section 5 points out some improvements. In section 6 we present the simulation results of our algorithm before section 7 concludes the paper with a short summary.

2 Related Work

Like many other data fusion systems, we also describe the functionality in form of dataflow graphs [4]. Consequentially, the implementation of a generic data fusion architecture has to involve a dataflow graph synthesis tool. The automated compilation of dataflow graphs into programmable hardware is a quite well studied subject in literature, [5] to mention just one. In [1] the problem of generating efficient software implementations from dataflow graphs on a single processor is addressed. Plenty techniques have been presented for synthesizing dataflow graphs onto multiprocessor systems, a problem known to be NP complete [7]. In the past, task allocation problems were often solved by *simulated annealing* [8]. Currently not many publications can be found that report the successful utilization of genetic programming for data flow graph synthesis jobs. A survey of using genetic algorithms to solve task allocation problems can be found in [2].

*This project has been partially supported by the Austrian Research Promotion Agency under grant #812033.

0-7695-3049-4/07 $25.00 © 2007 IEEE
DOI 10.1109/.41

3 The I-SENSE Architecture

For understanding our genetic algorithm for task allocation it is inevitable to give a compact description of the *fusion model* as well as the *hardware model*.

3.1 Fusion Model

The *Fusion Model* consists basically of a set of communicating tasks which may be represented as a task graph $G = (N, E)$. It is assumed to be a weighted directed acyclic graph, consisting of nodes $N = (n_1, n_2, ..., n_m)$ which represent the fusion tasks and the edges $E = (e_{12}, e_{13}, ..., e_{nm})$ the data flow between those tasks.

Each node has some properties, describing the (hardware and resource-) requirements of a task. Every edge from node u to node v (e_{uv}) indicates the required communication bandwidth between those two tasks. A quite simple example of a *fusion model* is shown in figure 1.

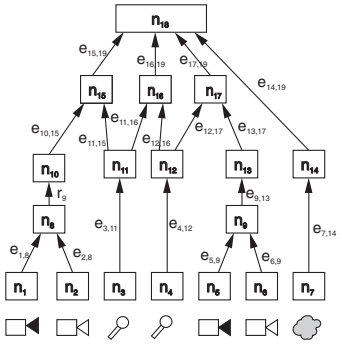


Figure 1. Example of a fusion model

Sensor Interfaces, the tasks at the bottom of the fusion tree, must reside on specific processors for obvious reasons. All other tasks can run on any processor of the system, as long as this processor has sufficient resources available.

3.2 Hardware Model

The *hardware model* describes the distributed embedded system where the fusion application should run on. In our case it consists of a set of connected hardware nodes ($N_1 ... N_3$, in figure 2). Each hardware node has at least one general purpose CPU and optionally some digital signal processors coupled via PCI and ports to connect sensors. Every processor allows us to query and use its free resources (i. e., computing power, on/off chip memory, I/O ports, ...) and sensors. So we can build and parameterize the *hardware model* during the initialization process automatically.

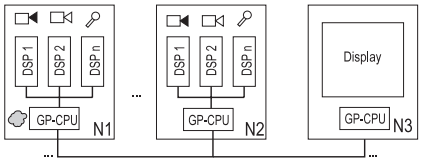


Figure 2. An exemplary hardware topology

4 Genetic Algorithms (GAs)

This section gives a short overview about GAs, always in reference to our goal of finding a near optimal allocation of n tasks on a distributed embedded system consisting of m heterogenous processors with the overall optimization objective *load balancing*.

4.1 Encoding of Chromosomes

To solve a problem via GAs, it is necessary to find a mapping of a potential candidate for a solution onto a sequence of binary digits, the so called chromosome. In our case, however, it is more efficient to represent chromosomes as strings of integers. The length of the chromosomes is given by the number of tasks that should be allocated. Every gene in the chromosome represents the processor where the task is running on. Figure 3 gives an exemplary mapping of n tasks on m processors.

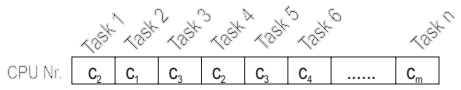


Figure 3. Mapping of tasks on processors

4.2 Implementation of Fitness Function

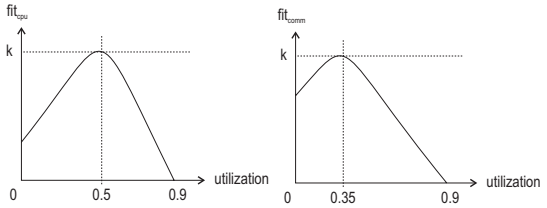
There are two criteria, processor utilization and communication bandwidth usage, which must be combined into a single fitness value $fit_{chromosome}$. A simple and feasible method is calculating two fitness scores, one for CPU utilization and one for communication bandwidth, and building a weighted sum of both partial results:

$$fit_{chromosome} = w_p \cdot fit_{processors} + w_c \cdot fit_{communication}$$

The fitness of a single processor $u(c_i)$ is calculated from its workload $load(c_i)$ via a polynomial function of second degree (4a) into the processor fitness score:

$$u(c_i) = k \cdot fit_{cpu}\left(load(c_i)\right)$$

The entire processor fitness of a chromosome is simply the sum of all $u(c_i)$.



(a) Conversion fit_{cpu} (b) Conversion fit_{comm}

Figure 4. Fitness conversion functions

In other words, a processor work load of 50% gets the highest possible fitness value - this has been determined empirically. More or less utilization causes the fitness score to decrease.

The utilization of the communication facilities is subdivided into intra- (PCI) and inter-node (Ethernet) communication. The overall communication utilization is calculated by taking the max. value of both. Since the communication between two tasks on the same CPU is implemented via shared memory, the communication cost for local messages can be neglected. The fitness of the communication is calculated similarly to the processor fitness, with the main difference, that a communication utilization of 35% (see fit_{comm} in figure 4b) results in the best possible fitness score - again determined empirically.

The fitness $e(c_i)$ of each individual processor, regarding the utilization of its communication facilities, is calculated as follows:

$$e(c_i) = k \cdot fit_{comm}\,(util(e_i))$$

The communication fitness score of the entire chromosome is again the sum of all individual scores.

4.3 Processing a GA

The typical genetic algorithm, like described in [6, 9], begins with an initial set of chromosomes which are generated randomly. Then all chromosomes are valuated by a fitness function; invalid chromosomes (i. e. a task has not sufficient hardware resources to run or overloads a processor) are removed from the population. If the optimization goal has been reached, the algorithm terminates. Otherwise some of the chromosomes are selected for reproduction, usually with a probability proportional to their fitness score. The reproduction itself consists of the crossover operation followed by a random mutation of genes. With the removal of invalid combinations and re-evaluation of the fitness score the loop is closed (Fig. 5).

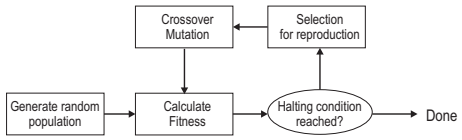


Figure 5. Principle of genetic algorithms

4.4 Required adaptions of GA

The traditional algorithm suffers from a few drawbacks and difficulties when applied to our problem.

In the simple GA, an invalid chromosome (where a task should run on a CPU but no implementation is given for this CPU type) is simply removed from the population. However, until this violation is discovered, the invalid chromosome occupies the slot of a potential good combination within the population and the check for illegal combinations consumes time. We effectively avoid this problem by randomly selecting a CPU from a list of suitable CPUs for a specific task when creating the initial population as well as inside the mutation operator.

The other problem is, that a randomly generated initial population consists almost entirely of invalid combinations; either the CPU utilization of a processor in the system is exceeded or a communication link is overloaded. As proposed by Jens Gottlieb [3], a penalty in the fitness score for those violations has proven in our work be a good approach. Good penalty functions unite the following characteristics:

- A penalty must be greater than the best possible fitness value. This guarantees that no invalid chromosome dominates a valid one.
- The penalty value must be proportional to the serverity of the violation.
- Penalty functions should not create local maxima in the invalid area. This guides the population into valid areas.

In reference to our problem, task allocations which overload a CPU ($load(c_i) > 90\%$) get a punishment value, proportional to the severity of the violation:

$$u(c_i) = -p \cdot (w_p + w_c) \cdot (\#cpu) \cdot (1 + load(c_i) - 0.9)$$

The same approach is applied to the communication usage as well; too high communication requirements ($util(c_i > 90\%)$) are punished:

$$e(c_i) = -p \cdot (w_p + w_c) \cdot (\#cpu) \cdot (1 + util(e_i) - 0.9)$$

In both cases, p has to be choosen so that the punishment value is higher than the highest possible score of a perfect chromosome. If more violations occur, the punishment value is subtracted repeatedly.

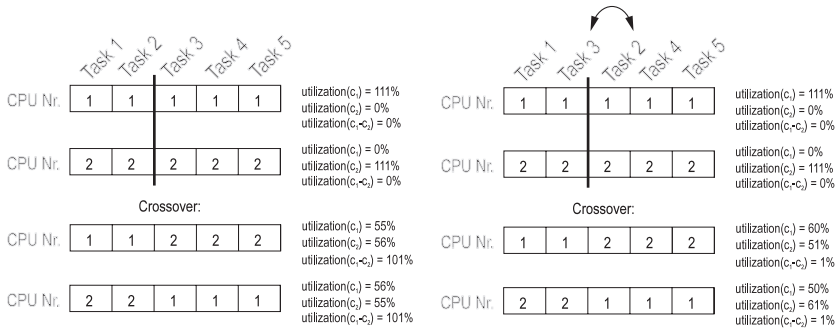


Figure 6. Improvement of the crossover operator when ordering the genes cleverly

5 GA with domain specific heuristics

With the modification just described, it is possible to implement a GA to find a solution for our task allocation problem. However, with increasing problem size this approach performs worse. As problem can be identified, that there is no correspondence of the genes and genetic operations in the real world problem. Neither the order of the genes, nor the point of intersection in the crossover operator, has a meaning in the task allocation problem. This leads to unrestricted search in the entire solution space with the advantage that even unfeasible task allocations are taken into account. On the other hand, many invalid allocations have to be checked and ruled out which slows down the search.

We propose an improvement that accelerates the search but does not restrict the search space and which can be implemented very efficiently. First, tightly coupled tasks should be mapped on adjacent genes in the chromosome. By doing so, the genetic crossover operator gets a real world meaning: Rather than individual tasks without relation among each other, entire coupled clusters of tasks are exchanged by the genetic crossover operator.

Figure 6 shows an example for this problem and the proposed improvement. Lets assume that fusion task 1 is coupled with task 3 tightly as is fusion task 2 and task 4. Task 5 is connected loosely to the other tasks. In the left example the tasks are mapped to genes in the order they appear. The top most chromosome has mapped all tasks to processor #1, and the next chromosome all tasks to processor #2. Both chromosomes have the good characteristic that should be retained: Task 1 and task 3 is placed on the same processor as is task 2 and task 4. Anyhow, both chromosomes are invalid, because they cause a too large processor utilization. Now those chromosomes are elected for a crossover operation at the location indicated by the thick line. The resulting two chromosomes utilize the available processors quite balanced, unfortunately the communication bandwidth between both CPUs is exceeded. Though, when applying our idea, the genes are ordered slightly different, like shown in the right example of figure 6. The initial condition is exactly the same, but now the crossover operator does not divide tasks which belong together. This results in two chromosomes that are both valid and have a good balance. In the general case, this approach does not prevent that highly interconnected tasks are separated. But it is much more likely that good sub-clusters remain together.

An easy and effective way to achieve a proper order of the genes is described in the following: In the beginning a set of clusters is created, each node of the task graph n_i is assigned its own cluster. The data rate between the cluster containing n_i and cluster containing n_j equals initially the edge of the task graph e_{ij}. Now the two strongest connected clusters are merged into one cluster. It is important that the order of the tasks inside the cluster remains unchanged when merging them. After that, the inter cluster communication is updated. Again the two most interconnected clusters are merged. This procedure is repeated until we end up in a single cluster. The order of the tasks in this cluster is the order of the genes we are aiming for.

A second enhancement is accomplished, if the clustering idea is introduced to the initial population as well. A good trade-off between complexity and speed up of the GA is obtained by creating the chromosomes of the initial population according to the following procedure: The tasks are grouped into classes, depending on their required connection bandwidth. For our hardware topology we decided to use four classes. Class 3 is composed of tasks which must reside on the same processor because of their extreme high data exchange rate. Tasks which must be coupled via PCI are combined in class 2. Tasks which can be connected via Ethernet are put into class 1. Class 0 summarizes all very loosely coupled tasks. Instead of placing the tasks in the order they appear in the fusion tree, tasks in the highest class 3 are placed first at a random processor together with their heavily coupled tasks. Next, the tasks in class 2 are placed randomly on any processor and their strongly connected tasks are fit on the same node and so forth until all tasks of all classes have been placed on the hardware.

6 Results

We've applied the straight-forward implementation of the GA and the enhanced algorithm for task allocation in distributed embedded systems to some realistic example data fusion trees of different sizes. The enhanced algorithm took less time to find an almost optimal solution on all tested examples. To obtain the results for the diagrams and tables, a mutation rate of 0.4 has been used and the chromosomes to pair have been selected by the roulette wheel method. Because of scattering, all presented values were averaged over 200 runs of the GA.

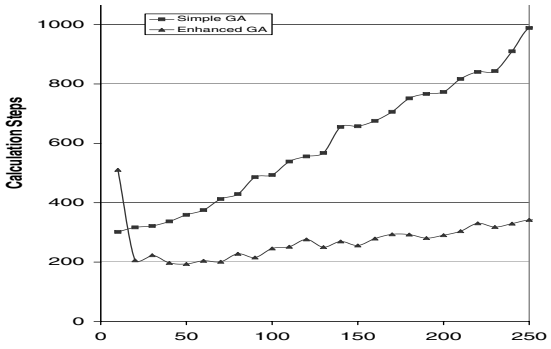


Figure 7. Impact of population size

In figure 7 the behavior of the simple implementation and the improved version of the algorithm is visualized depending on the population size. An example data fusion tree, comprised of 15 fusion tasks and a network consisting of 10 processors on 4 nodes, has been used as input. The population size is assigned to the x-axis; the required number of fitness function evaluations is applied to the y-axis. Except of the undersized population, the proposed enhancements accelerate the search noticeable.

Tasks	CPUs	Complex.	Normal	Enhanced
15	10	easy	4.63	3.00
26	10	medium	13.93	6.79
15	14	medium	20.29	12.90
20	14	hard	81.20	24.00
26	14	hard	58.40	38.93
20	12	very hard	203.86	25.79

Table 1. Iterations for a good configuration

Table 1 presents the number of iterations of the GA necessary to find an almost optimal configuration for many different *fusion-* and *hardware models*. The population size was fixed at 80 elements. All examples have been classified by its complexity. Problems classified as 'easy' have plenty valid solutions while in 'hard' problems the fraction of valid combinations is very small. For the example classified as 'very hard', a special *hardware model* has been constructed which utilizes all processors close at their limit.

On our development computer (a Pentium 4 running at 1.9 GHz) the enhanced algorithm takes on the average 115 ms to find a valid configuration for the example classified as *very hard*. After approximately 300 ms an almost optimal configuration is found.

7 Conclusion

We presented an enhanced genetic algorithm for task allocation in a distributed embedded system which scales quite well with problem complexity. Another advantageous property of the presented algorithm is the fact, that a valid configuration can be found quite fast, while it takes much longer to find an (almost) optimal solution.

References

[1] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[2] S. Dey and S. Majumder. Task allocation in heterogeneous computing environment by genetic algorithm. In *IWDC '02: Proceedings of the 4th International Workshop on Distributed Computing, Mobile and Wireless Computing*, pages 348–352, London, UK, 2002. Springer-Verlag.

[3] J. Gottlieb. On the feasibility problem of penalty-based evolutionary algorithms for knapsack problems. In E. J. W. Boers, S. Cagnoni, J. Gottlieb, E. Hart, P. L. Lanzi, G. R. Raidl, R. E. Smith, and H. Tijink, editors, *Applications of evolutionary Computing: Proc. EvoWorkshops 2001*, pages 50–59, Berlin, 2001. Springer.

[4] D. L. Hall and J. Llinas. *Handbook of Multisensor Data Fusion*. CRC Press, 2001.

[5] H. Jung, K. Lee, and S. Ha. Efficient hardware controller synthesis for synchronous dataflow graph in system level design. In *ISSS*, pages 79–84, 2000.

[6] M. Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, 1996.

[7] A. T. P. and C. S. R. Murthy. Optimal task allocation in distributed systems by graph matching and state space search. 46(1):59–75, Apr. 1999.

[8] B. Rinner, B. Ruprechter, and M. Schmid. Rapid Prototyping of Multi-DSP Systems Based on Accurate Performance Estimation. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing ICASSP 2001*, Salt Lake City, U.S.A., May 2001. IEEE.

[9] S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall International Series in Artificial Intelligence. Prentice Hall, 2003. RUS st 03:1 1.Ex.

[10] A. Tengg, A. Klausner, and B. Rinner. I-SENSE: A Light-Weight Middleware for Embedded Multi-Sensor Data-Fusion. In *Proceedings of the 5th IEEE International Workshop on Intelligent Solutions in Embedded Systems (WISES)*, Madrid, Spain, June 2007.