Architecture Synthesis and Partitioning of Real-Time Systems: A Comparison of Three Heuristic Search Strategies

Jakob Axelsson
Dept. of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden
Email: jakax@ida.liu.se

Abstract

This paper studies the problem of automatically selecting a suitable system architecture for implementing a real-time application. Given a library of hardware components, it is shown how an architecture can be synthesized with the goal of fulfilling the real-time constraints stated in the system's specification. In case the selected architecture contains several processing units, the specification is partitioned by assigning tasks to these. The use of three heuristic search techniques is investigated: genetic algorithms, simulated annealing, and tabu search; and it is described how these can be adapted to the architecture synthesis problem. It is concluded that tabu search is the most promising technique, but that simulated annealing is also applicable.

1 Introduction

When a *real-time system* is constructed, one of the most important design goals is to ensure that its *timing constraints* are fulfilled. These are imposed by the environment in which the system is embedded, and they typically place limits (expressed as deadlines) on the system's response time to certain events taking place in the environment. The quality of the system, i.e. its cost and its ability to reach the deadlines, depends to a large degree on certain early implementation decisions. The designer has to choose a suitable system architecture, and often, it might be advantageous to use a combination of application-specific integrated circuits (ASICs) and software running on standard microprocessors. It must also be decided how the resources of that architecture are to be shared between the system's tasks. This decision includes a *partitioning* of the functionality onto the components of the architecture.

In this paper, we study the automatic synthesis of an architecture from a real-time system's specification, and the partitioning of the system's tasks on the processors and ASICs of that architecture. It is extremely time consuming to find optimal solutions to such problems, and therefore we have to resort to *heuristic* search techniques. There are three well-known such techniques—*genetic algorithms* (GA), *simulated annealing* (SA), and *tabu search* (TS)—that have previously been used on similar problems. To determine which of these algorithms is the most suitable for our problem, we have conducted a comparative study involving all three of them.

The paper is organized as follows: The next section describes some related work and Section 3 defines the exact problem that we have studied, and proposes a suitable design environment. Section 4 shows how the three search techniques were adapted to the architecture synthesis problem and Section 5 reports the experimental results obtained. Finally, in Section 6 the conclusions are summarized, and some indications of future work are given. Due to the limited space, only a condensed account of the project can be given here, and the reader is referred to [3] for more details.

2 Related work

The synthesis of system architectures has been studied by several researchers before. Prakash and Parker [12] describe an algorithm for selecting processors to be used in a multiprocessor system, and for partitioning. It assumes that all processor nodes contain a local memory, and are connected through direct point-to-point channels. The algorithm does not consider real-time constraints.

Buchenrieder and Pyttel [4] use knowledge-based techniques for selecting components from a component library, and for deciding on how to connect them. The designer decides interactively on things like which processor to use, and what clock frequency the system should have. The design goal is unclear, but presumably it does not include satisfaction of real-time constraints.

0-8186-7895-X/97 $10.00

In hardware/software codesign, most researchers concentrate on how to partition a specification on a fixed architecture, in order to reach a maximal speedup, but some work has also been done on partitioning with respect to timing constraints, especially by Wolf's group (see e.g. [10]).

Some of the heuristic search algorithms have previously been applied to the partitioning problem. Eles *et al.* [5] compare the use of SA and TS for partitioning a graph into hardware and software parts while trying to reduce communication and synchronization between the parts, and it is concluded that TS is vastly superior to SA for this problem.

Ernst *et al.* [6] also do hardware/software partitioning using SA, but on a finer level than the previous reference. The target architecture consists of a processor and an ASIC which acts like a coprocessor. The design goal is to achieve a maximum average speedup under given cost constraints, so real-time issues are not considered.

Finally, Tindell *et al.* [13] use SA for assigning tasks of a real-time system to processors in a distributed architecture. The goal is to allocate the tasks in such a way that their deadlines are met, while not exceeding the capacity of the distribution network.

The principal contributions of this paper are:

- It uses heuristic search algorithms for concurrent architecture selection and partitioning, and compares GA, SA, and TS for this problem.
- It considers real-time applications, with the explicit goal of producing an implementation which reaches all deadlines at a minimal cost.
- It allows any mixture of processors and ASICs in the architecture.

3 Problem description

As mentioned above, the goal of this work is to find methods for providing an implementation of a real-time system. This implementation should guarantee the timing constraints stated in the specification, while incurring a minimal hardware cost. The main entities of our approach are:

Behavioural specification. The process of finding an implementation starts with a detailed *behaviour* that consists of a set $B = \{\tau_1, \ldots, \tau_m\}$ of parallel tasks, with *deadlines* and maximal *activation rates* specified.

Component library and models. The hardware components that can be used in the architecture are given in a *component library* containing microprocessors, ASICs (and other custom hardware devices such as FPGAs), memories, and instruction caches. Buses are implicit in the architecture, and therefore not included in the library. The executing components, i.e. the processors and ASICs, will be referred to as *processing units*. The library also contains estimation models for performance, size, and cost for the different components, and for this study we have used rather simple models. The microprocessor execution time and memory usage were estimated using techniques described in [1], the ASIC models were based on [11], and the caches were modeled by a single number indicating the hit ratio.

Virtual prototypes. During the design, the current status of implementation is captured using a *virtual prototype* (VP), which consists of: the behaviour; an architecture; a *partitioning* which assigns tasks to processing units; and a *schedule* which determines how processors and memories are time-shared by the tasks. In the applications we are currently studying, there is some data common to all the tasks, and the architectures must therefore always contain a shared data memory connected to all processing units.

Task scheduling. We assume that access to shared memories and processors is scheduled using a *fixed-priority* policy. The priorities can be selected either according to the *deadline-monotonic scheduling* (DMS), which is optimal for single-processor systems and which is calculated once and for all, since it does not depend on the implementation; or the optimal order can be calculated [2], which has to be done every time the design is changed. The complexity of the optimal priority assignment is $O(|B|^2)$. We will see later that the choice of priority order has a high impact on the resulting architecture. It is necessary to consider the scheduling, because it is otherwise not possible to check if the timing constraints are met in case of resource contention. The scheduling analysis can also be used to calculate the *minimal required speedup* (MRS) [2] which indicates how much the implementation would need to be accelerated in order to fulfil the timing constraints, thus it is a measurement of the distance to a satisfactory solution.

Quality criterion. During the search we need a *quality function*, which maps any VP onto a real number according to the following principles:

- A VP which fulfils the timing constraints has higher quality than one which does not;
- For two VPs that both meet the deadlines, the one with lower cost has higher quality;
- For two VPs that do not meet the deadlines, the cost and the MRS are combined into a single quality value.

A mathematical formulation is given in [3].

Transformations. The design proceeds by iteratively applying *transformations* which guarantee the VP's structural consistency. The transformations we use are:

- *Move task* from one processing unit to another.
- *Reconnect component* by connecting one of its ports to another component in the architecture.

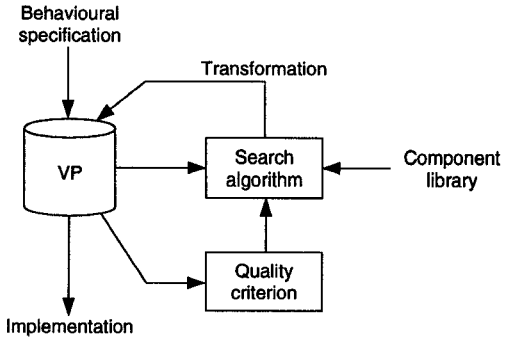

Figure 1. Overview of codesign environment.

- *Split processing unit* by inserting a new processing unit and moving some tasks to it from the old.
- *Merge processing units* by moving all tasks from one processing unit to another.
- *Add storage*, i.e. insert a new cache or memory.
- *Replace component* by an equivalent one (i.e. a processor by a different processor etc.) without changing the topology.
- *Cleaning*, by which unused components (such as processing units with no tasks) are removed. This transformation is always applied after one of the others, thereby avoiding the need for a remove transformation.

By restricting ourselves to such a limited set of operations, it becomes easy to verify the correctness of the search algorithms. Fig. 1 shows a codesign environment based on iterative transformation of virtual prototypes.

4 Applying the search algorithms

We will now briefly describe the three search algorithms, and show how they were adapted to our problem. To make the comparison fair, we have tried to use common parts whenever possible, and this includes the transformations as well as a procedure for generating random initial solutions.

4.1 Simulated annealing

Simulated annealing [9] is inspired by an annealing process, where matter is first melted, and then slowly cooled in a controlled way to obtain a certain arrangement of the atoms. When the temperature is high, atoms can occasionally move to states with higher energy, but then, as the temperature drops, the probability of such moves is reduced.

In the optimization procedure, the energy of the state corresponds to its inverse quality function value, and the temperature becomes a control parameter which is reduced during the execution of the algorithm. A neighbour of the current solution (i.e. in our case a VP which can be reached by applying one of the transformations) is randomly chosen in each iteration, and in case it is better than the current, the algorithm accepts to move to it. If it is worse, then the move is still taken with a probability determined by the difference in quality and the current temperature. During early iterations, there is a high probability to accept worse solutions, but as the temperature drops, acceptance becomes less and less likely. In this way, the algorithm behaves much like a random walk during early stages, while it performs almost a hill-climbing as the temperature drops towards zero.

4.2 Genetic algorithms

Genetic algorithms [8] are inspired by how changes in the chromosomes are made in nature to adapt a species to changes in the environment. Like its natural counterpart the GA is defined using the concepts of *crossover*, *mutation*, and *selection* based on survival of the fittest. The algorithm keeps a population of individuals, which represent a subset of the search space, and in each iteration, some of the individuals are replaced by new ones. These are created by selecting pairs of individuals from the previous population, where the probability of being selected increases proportionally with the individual's quality. The two selected individuals are cut up in two at a random point, and then one part of them is swapped between the individuals to create two new ones. For VPs, the crossover consists of selecting a number of processing units from each parent, and exchanging these to create two new solutions. Caches, separate memories, and the allocated tasks might follow the processing units in the swap. (In practice, the exact rules for doing this without creating any inconsistencies become rather intricate.) Fig. 2 illustrates the crossover of two VPs (the upper ones), where the dotted rectangles indicate the parts that are exchanged. (Since no component is connected to Mem2 after the swap, it is removed by the cleaning.) Finally, some random changes are done to the new individuals, which corresponds to a mutation. For the VPs, this is implemented as a random application of some of the transformations, e.g. task movement, or component replacement. (A GA which works on arbitrary data structures, rather than just bit-string chromosomes, is sometimes called an *evolution program*.)

4.3 Tabu search

Tabu search [7] is an iterative heuristic where long and short term memories are used to make the search more efficient. In this work, we only used the short term memory, which ensures that a recent step is not nullified while traversing the search space. The memory is implemented as

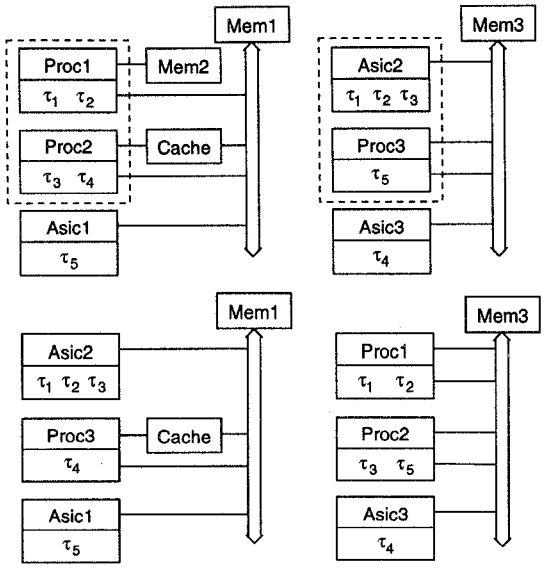

Figure 2. Crossover between two VPs.

a *tabu list*, which indicates whether a move is disallowed. It has a fixed length n, which is also the number of iterations a tabu remains in effect. In some situations it might be sensible to override a tabu, and allow the move. This is determined by an *aspiration* criterion, and we have used a very simple such: a suggested solution which breaks a tabu is accepted if it is better than the best solution found so far.

In each iteration, k elements of the neighbourhood of the current solution is examined, and the best non-tabu element in that set is selected as the next element. The neighbourhood list must be constructed using heuristics which consciously try to find those elements in the entire neighbourhood that are most likely to lead to the optimum. There are no random factors involved in the selection of the move. We picked the k neighbours to consider using two approaches depending on the current solution:

- If the deadlines are not met, the algorithm tries neighbours which increase the capacity, by moving tasks, splitting processing units, introducing caches, etc.
- If the deadlines are met, the algorithm tries to reduce the cost, by merging processing units, replacing components with cheaper ones, etc.

After the selection is made, the tabu list is updated to prevent the move from being undone.

4.4 Comparison

The three search algorithms traverse the search space in rather different ways. TS and SA move sequentially, always selecting a neighbour of the current solution, whereas GA covers a large number of points in parallel, and makes long leaps between them. On the other hand, SA and GA progress randomly, while TS is totally deterministic.

The effort to implement GA was considerably larger than for SA and TS, due to the complexity of the crossover operation. GA also has a disadvantage in that it needs to store information about a large number (sometimes several hundreds) of solutions, whereas SA and TS only store a few.

5 Experimental results and discussion

To compare the three search techniques, we conducted a large set of experiments with all three of them. First, we had to determine suitable parameter values for the algorithms. This includes things like starting temperature and temperature decrease factor for SA; probability of crossover and mutation for GA; and size of tabu list and neighbourhood for TS. Since the algorithms are partly random, the same experiments had to be repeated many times to determine their average results. For the comparison to be fair, the algorithms should be granted approximately the same execution time, and we therefore limited the number of generated VPs to 3,000 in each run. The component library we used contained three microprocessors, three ASICs (differing in maximal area), three caches, and one memory, which gives room for many different cost/performance trade-offs.

To investigate the importance of the behavioural specification's structure, we used a set of *synthetic benchmarks*, where the tasks were described as a vector which captured its important characteristics (size, execution time on different processing units, etc.). In this way, the experiments could be controlled to make sure that sufficiently different behaviours were covered. To validate the entire approach, including estimators, we also conducted a case study, in which architectures were generated for a *packet switch* (see [1]). All the examples were large enough to call for non-trivial architectures, i.e. more than one processing unit.

Table 1 summarizes the experiments for the synthetic benchmarks using optimal scheduling, and for the switch using both optimal scheduling and DMS. The two figures in each cell indicates the average cost of the best feasible solutions found (i.e. solutions that do not break any constraints), and the average number of iterations needed to find them. Feasible solutions were found in all runs, except for synt2, where GA succeeded in 20% of the runs, TS in 90%, and SA in all. The main findings of the experiments were:

- All the algorithms managed to find feasible solutions in almost all the experiments, even when the constraints were very tight.
- SA and TS found solutions with similar quality, with SA slightly better for the synthetic benchmarks and TS superior for the switch. GA was always clearly worse.

Name / #tasks	GA	SA	TS
synt1 / 6	788.6 1510	546.2 1875	716.9 422
synt2 / 12	1450.0 2710	1334.3 2199	1398.5 1377
synt3 / 6	1147.6 662	937.8 1993	989.8 528
synt4 / 9	1370.7 1518	1231.0 2335	1255.3 1113
switch / 12 (opt. sched.)	843.4 535	792.6 1902	759.2 752
switch / 12 (DMS)	841.1 245	824.9 1995	804.2 584

Table 1. Summary of experimental results.

- TS came to good solutions much faster than SA, and had the maximal number of iterations been more limited, the TS would have been superior in quality.

For the switch, a more detailed study was performed, comparing the results when using optimal scheduling to those for DMS, and it can be seen that using the optimal scheduling generated results that were clearly superior (for SA and TS). A closer study revealed that this was due to the fact that the optimal scheduling can (in practice, and not only in theory) meet the deadlines for many partitions on architectures with several processing units, for which the DMS was not feasible. This motivates the use of optimal scheduling during architecture synthesis, even though it implicates a longer algorithm execution time.

We also compared the algorithms' results with the optimal implementation for the switch, which was determined through exhaustive search. This implementation consisted of the cheapest processor, the most expensive ASIC, and the shared memory, and had a cost of 752.7. It was found no less than 13 out of 15 times by TS using optimal scheduling (but never by SA), which indicates that TS also was able to find very good partitions for the architectures it generated.

6 Conclusions and future work

In this paper, we have studied the use of three heuristic algorithms for automatic synthesis of architectures for real-time systems. Our experiments show that tabu search and simulated annealing can both be applied to this problem, whereas genetic algorithms are less suitable, due to the difficulty in defining a reasonable crossover operation.

The tabu search algorithm that we used was very simple, and it is our intention to include more advanced features of the algorithm, such as a long term memory, which can be used to *intensify* and *diversify* the search, thereby making it even more efficient. Due to these possibilities of further improvement, and its superior search speed, we judge TS to be the most promising technique for this problem, although SA is also applicable.

Another possibility of improvement is the estimation models, which are not very accurate today, in particular for ASICs. Since estimators are crucial for any work in codesign, we expect to see considerable progress in this area within the near future. Finally, we will also remove some restrictions from the behavioural specifications, to better handle inter-task communication, and this will entail development of new schedulability analysis models.

References

[1] J. Axelsson. Schedulability-driven partitioning of heterogeneous real-time systems. Licentiate Thesis No. 517, Linköping University, 1995.

[2] J. Axelsson. Hardware/software partitioning aiming at fulfilment of real-time constraints. *Journal of Systems Architecture*, 42(6–7):449–464, 1996.

[3] J. Axelsson. Three search strategies for architecture synthesis and partitioning of real-time systems. Technical Report LiTH-IDA-R-96-32, Dept. of Computer and Information Science, Linköping University, 1996. (Available from `http://www.ida.liu.se/publications/techrep/`)

[4] K. Buchenrieder and A. Pyttel. System zur wissensbasierten Konfigurierung von Leiterplatten. *CADS*, 92(1):52–59, 1992.

[5] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. To appear in *Design Automation for Embedded Systems*, 1997.

[6] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, 10(4):64–75, Dec. 1993.

[7] F. Glover, E. Taillard, and D. de Werra. A user's guide to tabu search. *Annals of Operations Research*, 41:3–28, 1993.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[9] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[10] C. Lee, M. Potkonjak, and W. Wolf. System-level synthesis of application specific systems using A* search and generalized force-directed heuristics. In *Proc. 9th International Symposium on System Synthesis*, pages 90–95, 1996.

[11] S. Narayan and D. D. Gajski. Area and performance estimation from system-level specification. Technical Report ICS-92-16, University of California, Irvine, 1992.

[12] S. Prakash and A. C. Parker. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16:338–351, 1992.

[13] K. W. Tindell, A. Burns, and A. Wellings. Allocating hard real-time tasks: An NP-hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.