Seminar “Embedded Systems”

Hardware/Software Partitioning

Frederick Dufour
Advisor: Martin Radetzki

2006, July 3rd
Contents

1 Introduction 2

2 Hardware/Software Partitioning: A Key Element of Co-Design 3
  2.1 Codesign 3
  2.2 Hardware/Software Partitioning 4

3 Approaches to Hardware/Software Partitioning 5
  3.1 Definition of the Partitioning Problem 5
  3.2 The Cost Function 5

4 Partitioning Algorithms 7
  4.1 Constructive Algorithms and the Example of Hierarchical Clustering 7
    4.1.1 How Hierarchical Clustering Works 7
    4.1.2 A Very Simple Example 7
    4.1.3 Possible Extensions 9
  4.2 Iterative Algorithms: the Example of Simulated Annealing 9
    4.2.1 Overview 10
    4.2.2 How Simulated Annealing Works 10

5 The CoSyMA System 13
  5.1 Overview 13
  5.2 A Partitioning Algorithm Using Dynamically Determined Granularity 13
    5.2.1 Main Idea 13
    5.2.2 Motivation for Dynamic Granularity 13
    5.2.3 The Algorithm 15

6 Conclusions 20

Bibliography 21
Chapter 1

Introduction

This article is mainly based on chapters 8.4 and 8.5 of [Tei97], as well as on additional literature which will be mentioned as it is cited.

Abstract

Embedded Systems have reached such a complexity that it has become impossible to design them “from scratch”. It is often the case that both software (compiled code that runs on processors with generic sets of instructions) and hardware (circuits designed to accomplish a specific task or calculation, e.g. decoding multimedia information or any digital signal processing operation), can execute a given operation, but with different performance and material costs that are to be considered during the design phase. The problem is to build a partition, i.e. assigning every task to an object.

From the global context of codesign, we will define the partitioning problem, and study some popular algorithms used to solve it. Finally, we will use this knowledge to explore the COSYMA system, which is a computer-aided design tool for digital systems.
Chapter 2

Hardware/Software Partitioning: A Key Element of Co-Design

2.1 Codesign

Gupta and de Micheli ([GM93]) showed that it is possible to “achieve synthesis of heterogeneous systems which uses timing constraints to delegate tasks between hardware and software such that the final implementation meets required performance constraints.”

While pure hardware synthesis tools like Hardware Description Languages allow the implementation of systems on a chip from high-level specification, the resulting cost of implementation is a considerable drawback. On the other hand, off-the-shelf generic processors, and implementation in software, are much cheaper, but seldom suit the hard time constraints imposed on embedded systems. Thus, cost-effective designs should use a mixture of hardware and software to accomplish their goals.

The problem with the design-oriented approach, where allocation of tasks to hardware and software precedes the synthesis, is that it is not known whether the final system will meet its requirements. This suggests a synthesis-oriented approach, where constraints on performance and cost of the systems are specified, and a systematic constraint-driven exploration of the design space is done.

The model must first be given in terms of a specification for hardware (often in a Hardware Description Language) and for software. Then it is partitioned, and the specifications allow run-time estimations, simulations as well as cost estimations, to test whether the resulting system meets the constraints.

Finally, the system is synthesized. Hardware synthesis from the specification is now well-known and well implemented in CAD-tools. The software code is then implemented from the graph models, and interface circuitry is synthesized from the partitioned model.

Figure 5.1 (p. 14) gives a schematic view of the codesign process as implemented in the COSYMA system.
2.2 Hardware/Software Partitioning

As defined by Gupta and De Micheli in [GM93],

“The system-level partitioning problem refers to the assignment of operations to hardware or software.

Overall system performance is determined by the effect of hardware-software partition on utilization of the processor and the bandwidth of the bus between the processor and application-specific hardware.

Thus a partitioning scheme must attempt to capture and make use of its effect on system performance in making trade-offs between hardware and software implementations of an operation.”

Typically, the time constraints of the system won’t be met if it is all implemented as software on a generic processor, but it would be too expensive to design an application-specific hardware chip for the whole functionality.

A simple graphical representation of the problem can be given in the following way: a graph binding problem with on one side, the graph representing the application to run, which nodes are the functions, processes or instructions (depending on the granularity, see 5.2.1), and on the other side, the component architecture graph, with the processor(s), the application-specific hardware (DSP, etc.), and the communication buses between them.

Each node of the application graph is connected to every node of the architecture graph on which it is able to run, and the edge is weighted with the runtime or the cost (which, however, often depends on the whole system rather than on simple pairwise values), as well as the edges that stand for the buses. The problem is then to find a matching with minimal cost. Of course, more elaborate models can take into account more specific limitations, like the maximum number of gates or the maximum chip area for the architecture nodes.

The difficulty of scheduling problems (they are NP-hard, see [LKB77]) makes hardware/software partitioning a weak link in the chain of computer-aided co-design.
Chapter 3

Approaches to Hardware/Software Partitioning

3.1 Definition of the Partitioning Problem

Teich ([Tei97]) gives the formal definition of the hardware/software partitioning problem as follows:

Given a set \( O = o_1, o_2, \ldots, o_n \) of functional objects, a partition \( P = p_1, p_2, \ldots, p_m \) of system components is sought, that satisfies:

- \( p_1 \cup p_2 \cup \ldots \cup p_m = O \),
- \( p_i \cap p_j = \emptyset \) for all \( i, j, i \neq j \), and
- the cost \( f(P) \) is minimal

All functional objects \( o_i \) in a system component \( p_j \) will be implemented on this same component. A partitioning algorithm matches every functional object with a system component object and looks for partitions with the lowest possible cost. With \( n \) functional objects and \( m \) possible system components, there are obviously \( O(m^n) \) possibilities. As a consequence, it is impossible to use any kind of exhaustive search. The algorithms presented have various approaches to the problem, but they all do a selection on the solutions they explore and evaluate. Figure 3.2 gives a schematic view of the general partitioning problem.

3.2 The Cost Function

The cost function expresses the factors the designer wants to minimize (mostly, time constraints and hardware costs). It can involve execution times, chip area, number of gates, needed memory, communication costs, etc.

By fine-tuning the factors, one can express the priorities of the different constraints. Thus, absolute constraints (like time constraints, or impossible matchings) will often have exponential factors (see 5.2.3), whereas optimization
Functional objects → System components

Figure 3.1: A schematic view of the general partitioning problem

Constraints (often the case of hardware costs) will be expressed by terms with a slower growth, e.g., linear. This cost function is of prime significance because the algorithms use it to select “good” partitionings. The main difficulty is to estimate the influence of a partition on timing constraints, as timing behaviours are often global by nature. This makes incremental computations of the cost function not trivial, but the latter are essential to make efficient partition algorithms. Approximation techniques have been suggested to take into account the effect of a partition on overall latency.
Chapter 4

Partitioning Algorithms

There are two main families of algorithms to solve the hardware/software partitioning problem: constructive algorithms and iterative or transformative algorithms. We will see how these categories of algorithms function, and study an example of each category.

4.1 Constructive Algorithms and the Example of Hierarchical Clustering

This range of algorithms build a partitioning in bottom-up fashion by grouping (clustering) nodes, using so-called closeness-functions to estimate the costs, as the other “realistic” criteria cannot be computed due to the lack of a complete partition during computation.

4.1.1 How Hierarchical Clustering Works

This algorithm (in fact, a family of algorithms) builds a partition by pairwise grouping of partition objects, until a certain condition is reached (the number of partition objects has decreased to the wished level, or the overall closeness has reached a given level). It starts with assigning every functional object (that is, every node of the functional graph) to a different partition object. The algorithm then computes the closeness between \( p_i \) and \( p_j \) for every pair \((p_i, p_j)\) of partition objects, and merges the pair with the highest closeness into a single partition object. The closeness is often based on the communication costs between the two components.

The algorithm thus builds a tree, the leaves of which are functional objects. Cutlines (sets of vertices on a same depth level) are partition candidates. A proper partition is obtained when conditions on the number of partition objects and the closeness have been reached.

4.1.2 A Very Simple Example

Figure 4.1 shows the different steps of the process on a very simple example with four objects. The pairwise closenesses of objects are represented as edge
Figure 4.1: An example run of the Hierarchical Clustering algorithm. The nodes selected in red have the highest closeness, they will be merged in the following step.
weights on the left subschemata. The right subschemata show the current tree as built by the algorithm.

At the beginning, all processes are leaves, and there is no intermediate node. The blocks \( p_1 = \{o_1\}, p_2 = \{o_2\} \) show the highest closeness, and they are regrouped in a single block. We thus have at step b) a new block \( p_5 = \{o_1, o_2\} \), so the nodes 1 and 2 are now connected on the tree. The closeness values are also computed as the average of weights between the objects that form them. For instance, \( c_{3,5} = \frac{c_{1,3} + c_{2,3}}{2} = 20 \).

Then, blocks 3 and 5 have the highest closeness, so they are grouped in a single block 6. The algorithm terminates, as the initial constraint numblocks = 2 is fulfilled.

4.1.3 Possible Extensions

Multi-Stage Clustering

The algorithm can be extended to multi-stage clustering where the whole history (the tree built by the algorithm) is considered instead of just the previous step when computing the closeness and selecting the pair to be merged.

Given a tree, one can use a target function to determine the locally best created cut. Then, taking this cut as starting partition, one can use the clustering process again, this time using a different closeness metrics. This process increases the actual amount of design space explored.

Ratio-cut

If the closeness function used is the sum of the weights of all edges between vertices in different blocks, then the “best” partition would have to be the one where there is only one block, regrouping all functional objects. To avoid this, a restriction on the maximum number of functional objects in a group can be added to the model. However, this alters the computed quality of the partitions. To avoid this, we can introduce a new metrics called Ratio-cut.

Let \( P = p_i, p_j \) and \( \text{cut}(P) \) the sum of weights of all edges between objects that lie in different blocks \( p_i \) and \( p_j \). Let \( \text{size}(p_i), \text{size}(p_j) \) be the block size (number of objects) of \( p_i, p_j \). The ratio of \( P \) is defined as such:

\[
\text{ratio} = \frac{\text{cut}(P)}{\text{size}(p_i)\text{size}(p_j)}
\]

The enumerator acts in favor of a high regrouping of objects, while the denominator ensures that the blocks are balanced in size. A heuristic using this metrics would be called Ratio-cut.

4.2 Iterative Algorithms: the Example of Simulated Annealing

This range of algorithms start with a given partition (often a randomly generated one) and improve it iteratively (that is, modify it to minimize a given cost function). We will study the example of Simulated Annealing, which is used in the COSYMA system (see 5), and was invented by Kirkpatrick et al. ([KGV83]). [Tei97] gives the following explanation of the algorithm:
“Starting from an initial partition, a simulated temperature is slowly decreased. For every temperature, random moves are generated. The algorithm, under certain assumptions, can reach a global optimum. Its complexity strongly depends on the implementation, varying from low-order polynomial to exponential.”

The following is a simple explanation of the algorithm. The reader might want to refer to the article on the Wikipedia (http://en.wikipedia.org/wiki/Simulated_Annealing) for further details.

4.2.1 Overview

The Origins: Annealing in Metallurgy

This algorithm is inspired from annealing, a method in metallurgy that consists in cooling a material with a variable speed to favor the formation of crystals or other regular, stable structures. The material is first melted down, and its temperature lies high over the fusion temperature of the element. Then, it is progressively cooled down, more and more slowly as it approaches the fusion temperature of the material (which in this case is actually the solidification temperature). The underlying physical reason for this method to work is that the melting down of the metal/material lets its atoms move freely and randomly, thus providing a fair distribution in the movement space (position and speed vectors). But then, the atoms need to move slowly to attach to each other to form regular structures with higher density (in other words, the material needs to lower its temperature). The spatial distribution of the atoms at the stage of solidification (when the material reaches the corresponding temperature) has a strong impact on the internal energy, and thus stability of the resulting structures. For this reason, the material should be cooled down slowly when approaching this temperature.

Relation to the Partitioning Problem

When using this algorithm to solve the partitioning problem, we can use the same paradigm, with a simple analogy: the cost function defined in the partitioning problem could be seen as the internal energy of a system, which we try to minimize. A possible partition could be seen as the material, and any elementary modification of the partition (moving one functional object to another system component), as movement of its “atoms”. Just as a higher temperature allows greater moves of the atoms, we could use an external parameter (still called “temperature” for the sake of the comparison) that would influence the amplitude of the modifications the algorithm executes on the solution.

4.2.2 How Simulated Annealing Works

The algorithm starts with a given partition (unlike a constructive algorithm) which might be generated randomly, or by a constructive algorithm, or determined externally. Then, using the temperature as an external parameter that decreases slowly (compared to the iterations), it repeats the following iteration on the solution, where a single modification is considered, until the solution is considered
suitable for the given purpose (most often, when the cost has reached a certain value, or a given number of iterations have been executed).

It can be noted that the behaviour of the temperature parameter depends on the implementation, and plays a significant role on the overall efficiency of the algorithm.

The Iterated Loop

A random move is selected: transferring a randomly selected partitioning object (the size of which depends on the granularity chosen, from single instruction to more complex task) to another system component (also randomly chosen if there are more than one). Of course this is not exactly true, and depending on the problem, moves that change the solution too much, or cause a great variation of the cost function, will be avoided.

Then, it will compute the new value of the cost in this new state (if possible incrementally, that is, using the old value and only adapting the factors that need to) and decide whether the move is accepted or not, in which case the next iteration will start with the same solution and consider another move. If the move is from state \( s \) with cost value \( F \) to state \( s' \) with cost value \( F' \), then the probability that the move is allowed is:

\[
P(F, F', T) = \begin{cases} 
1 & \text{if } F' < F \text{ (i.e., downhill moves are always valid)} \\
\frac{1}{e^\frac{F-F'}{T}} & \text{otherwise.}
\end{cases}
\]

As a consequence, even moves that lead to an increase of the costs will be allowed during the first iterations of the algorithm, and they will be less and less likely to happen as the temperature is decreased. At the end, when the temperature is low enough, almost only downhill moves will be allowed. As downhill moves are always allowed, the algorithm will eventually reach at least a local minimum. But the peculiarity of the algorithm lies in the fact that even uphill moves are allowed, at least at the beginning. This ensures a good exploration of the design space, and prevents the solution considered from stopping at a local minimum which might be good compared to its neighbourhood but not to the whole design space.

A Very Simple Example

Figure 4.2 shows different stages of an example run of the Simulated Annealing algorithm. The design space, the dimensions of which could be the different functional objects, is simplified as the X-axis. The curve stands for the cost function, and the black dot for the current solution. Between each subfigure, the temperature has decreased, and the algorithm has gone through a number of basic iterations described in the last paragraph. In only one iteration, the solution can only “move” in its close neighbourhood, but eventually it could reach any portion of the curve in blue. On the other hand, portions of the curve in red are solutions that are too high relatively to the current solution, so the heuristic will reject moves to these areas. The color gradient symbolizes the decreasing probability that moves to the corresponding areas can be allowed. As the temperature decreases, the gradient gets steeper: at a certain stage, almost only downhill moves are allowed.
Figure 4.2: A possible run of the Simulated Annealing algorithm. The dot stands for the current solution. It slowly moves in the design space. In the second and third steps, the algorithm allows moves although they go “uphill”, in the fourth, the low temperature will only allow “downhill” moves, which will eventually yield the minimum.
Chapter 5

The CoSyMA System

5.1 Overview

The CO-SYnthesis for eMbedded micro A rchitectures (COSYMA) system is a platform for exploration of the co-synthesis process. It was developed mainly at the University of Braunschweig, Germany, and is described in [ÖBE+97]. Figure 5.1 (p. 14) gives an schematical view of the system.

5.2 A Partitioning Algorithm Using Dynamically Determined Granularity

5.2.1 Main Idea

While traditional schemes determine a fixed granularity, i.e., the size of the partitioned “objects” (functions, instructions, processes), J. Henkel and R. Ernst have designed an algorithm that determines the granularity dynamically, during the partitioning. The algorithm is described in [HE97]. It consists of three main steps:

Base granularity: A control flow graph of the application is built, which nodes are basic blocks (typically, single instructions).

Partitioning objects: Using only structural information of the graph, basic blocks are grouped into sets called partitioning objects, which are not mutually exclusive.

Macro-instruction: Using behavioural information, an optimization algorithm is used to find a solution to the hardware/software partitioning problem.

5.2.2 Motivation for Dynamic Granularity

Consider the simple case of two tasks and an architecture consisting of a standard processor (SW) and an application-specific chip (HW). There are only four possibilities for implementing, only two of which really are HW/SW systems. This means that the hardware chip will have to be able to execute a whole task, and its synthesis will therefore be expensive. But it might have been possible to
Figure 5.1: A schematical view of the COSYMA system
meet the time constraints only by executing a single loop or group of instructions on hardware. This example shows a drawback of coarse granularity.

Assume the same example as above is considered, but this time, each task is described at the level of basic blocks. There could be 1000 blocks per task, and supposing each architecture could possibly implement any block, there are \(2^{1000}\) possible implementations. Even with a good heuristic, the computational time required to find a global optimum would be totally unacceptable. This example shows a drawback of fine-grain granularity.

5.2.3 The Algorithm

Base Granularity

The functional graph of the application is built, with additional information as to the nesting levels regarding control constructs. Thus, vertices standing for a block embedded in nested control constructs are marked with the corresponding level. This process is shown with an example on figure 5.2.3 page 15.

![Functional Graph](image)

(a)

![Functional Graph](image)

(b)

Figure 5.2: Example of a functional graph of an application. The vertices (single operations) have been marked with their nesting levels.
Partitioning Objects

The idea is to build partitioning objects regrouping whole control constructs, using information about the nesting level. This step only uses structural information. This is shown for our example on figure 5.2.3 (p. 18). This provides the next step with a selected number of partitioning objects that are consistent, while covering the whole span of sizes possible. This is the basis of dynamic granularity.

Macro Instructions

The optimization algorithm used here is Simulated Annealing (see 4.2). At each step of the algorithm, a move is generated by the annealing schedule, depending on the current partition. It is then accepted or rejected (this depends on the difference of cost between the current and the would-be partition, as well as on the current temperature). At each step, exactly one partitioning object is moved from hardware to software, or vice-versa. Figure 5.2.3 (p. 19) shows a panel of base configurations and some of the associated moves. For each iteration, it randomly chooses a partitioning object \( o_r \). Then, there are three cases:

1. The selected object is in software. Then, its move to hardware is considered valid.
2. The selected object \( o_r \) is in hardware. Then, if \( o_r \) is contained in another partitioning object \( o_j \) that contains a node that is in hardware and not in \( o_r \), the move is rejected, otherwise it is considered valid.
3. The selected object \( o_r \) is partly in hardware and partly in software. Then, moving all nodes of \( o_r \) that are in software to hardware is considered a valid move.

Cost function

Then, the cost function determines whether a move is accepted or rejected. The cost function can be summarized as follows:

\[
\text{cost} = A \cdot \text{cost}_T(T_{HW/SW}) + B \cdot w_{\text{area}} \times \frac{A_{\text{area}}}{A}
\]

The first term evaluates the meeting of time constraints, whereas the second term evaluates the amount (area) of hardware, tending to minimize the hardware effort. Factors \( A \) and \( B \) are chosen so that the time constraint always has priority over the optimization constraint (an invalid, although optimized, design is of little use). \( A_{\text{area}} \) is the gate count of all partitioning objects currently implemented in hardware, and \( \overline{A} \) is the average gate count of a partitioning objects. Both parameters are evaluated using a high-level estimation tool.

The time cost is computed as follows:

\[
\text{cost}_T = e^{\frac{T_{HW/SW} - T_{\text{est}}}{T_0}} \cdot \|T_{\text{est}} - T_{HW/SW}\| \cdot \frac{1}{T_N}
\]

The exponential factor eliminates invalid designs that do not meet the time constraints, thus accelerating the convergence of the algorithm. \( T_0 \) and \( T_N \) are
chosen heuristically. $T_{\text{cost}}$ is the time constraint, $T_{\text{HW/SW}}$ should not have a higher value than $T_{\text{cost}}$.

Important is that the cost can be computed incrementally, to save calculation effort during execution of the algorithm.

Results

Due to the variable size of the partitioning objects that can cover the whole application or only a single instruction, the computation time could almost be kept independent from the size of the benchmark.

However, the model could be extended to take into account the possibly concurrent execution of functions on hardware and software. Only then would it enable designers to make the most of hardware/software systems.
Figure 5.3: How the partitioning objects are generated
Figure 5.4: An example of possible base configurations and associated moves.
Chapter 6

Conclusions

Co-synthesis of Hardware/Software systems is beginning to take full advantage of Computer Aided Design, as some tools integrate most of the needed features. However, some issues like hardware/software partitioning are still topics of ongoing research, and therefore the tools still lack powerful solutions to these problems. In most cases, human experience is needed in the process. However, new, innovative approaches like the ones used in the COSYMA system give hope that the tools may become reliable and efficient enough to allow a completely automatised, optimized synthesis of complex systems.
Bibliography


