HARDWARE/SOFTWARE PARTITIONING OF REAL-TIME SYSTEMS

Jakob Axelsson

Abstract

Many high-performance embedded real-time systems are today implemented heterogeneously, with some parts of the functionality in hardware and others in software. In this paper, we discuss how hardware/software codesign techniques can be used to improve the design of such systems, and we focus on hardware/software partitioning and its relation to other important design activities, such as system behavioural description and hardware architecture selection.

1 Introduction

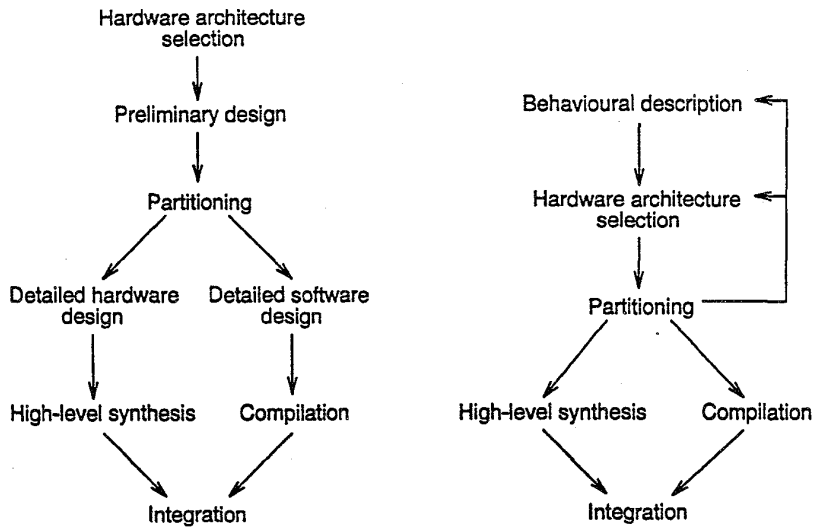
The field of hardware/software codesign has received increasing attention during the last few years, and numerous publications have appeared describing different systems and methodologies (a recent bibliography[1] lists over 250 publications related to codesign). Most approaches are based on the fact that a single executable description of a system can nowadays be compiled into either silicon or machine code, and this opens up the possibility to uniformly describe the behaviour of a system, which will be implemented in a combination of application-specific hardware and software. This description is then partitioned, with assistance from more or less automatic tools, into separate hardware and software parts, which are passed on to a high-level synthesis tool respectively a compiler to produce the final implementation.

One class of applications particularly well suited for hardware/software codesign is embedded real-time systems. These systems have well-defined timing constraints, which are sometimes so severe that they can only be met in a hardware implementation, but at the same time they might have less time-critical parts that can benefit from the lower cost and higher flexibility of a software solution. Therefore, heterogeneous implementations are often necessary for these applications, and are indeed frequently employed today. But there is a lack of design methodologies and tools that can support their implementation across the hardware/software technology barrier.

When designing a real-time system intended for heterogeneous implementation, the following activities are undertaken:

- *System behavioural description*, giving an executable specification of what the system is supposed to do (a software engineer would refer to this as a *program*).
- *Hardware architecture selection*, describing what hardware components should be used and how

Jakob Axelsson is with the Department of Computer and Information Science,
Linköping University, Sweden.



(a) Traditional design (b) HW/SW codesign

Figure 1: Traditional design (a) compared to hardware/software codesign (b).

they should be connected.

- *Task scheduler design*, forming the core of the kind of rudimentary operating system used in many embedded computer systems. The rôle of the scheduler is to control how the different computational resources of the hardware architecture are shared between the tasks of the behavioural description.
- *Hardware/software partitioning*, deciding which parts of the system behaviour should be realized by what parts of the hardware architecture.

Since we are interested in optimizing the system resulting from the combination of these four parts, we also need a quality measurement. For real-time systems, the premier goal is to meet the timing constraints, and how well this is done is measured by the ***task schedulability.***

Figure 1 illustrates how the current practice of real-time systems design (adapted from Lawrence and Mauch[2]) compares with codesign. Major differences are that the hardware architecture selection and the partitioning are performed later than in traditional design, thus allowing these design decisions to be made based on more complete and accurate information.

The following four sections of the paper each deal with one of the design concepts listed above, and clarify how they are related to each other. In section 6, we will take a look at an experimental workbench currently under development within our group, which provides a set of tools that facilitate the implementation in hardware and software of a real-time system.

2 Behavioural description

Several programming languages have been developed with the behavioural description of real-time systems in mind, e.g. Ada[4], Erlang[5] and Real-Time Euclid[6]. Common for these languages is that they use *tasks* (sometimes called processes) as the fundamental way of structuring the system. The tasks are, conceptually, executing in parallel (even though a software implementation has to sequentialize

the execution in some fashion). Since the hardware and software parts of a system also run in parallel, we find it natural to let the task be the entity that we assign either to hardware or software during partitioning. In this paper, we will not limit the discussion to any specific language, instead we just make the general assumption that the behavioural description consists of a set of (independent) tasks, $P = \{\tau_1, \ldots, \tau_m\}$.

Typical for real-time systems is that they are subject to timing constraints, and these are expressed as deadlines on each execution of the tasks, captured by the function $D(\tau)$. We also provide limitations on the minimal time $T(\tau)$ between two activations of a task. Our main concern when implementing real-time systems is to assure that these timing constraints are never violated, and we therefore analyze the behavioural description with respect to its worst-case execution time. To make this possible, we must impose certain limitations on how the description is written:

- All loops must have explicit upper bounds on the number of iterations, since it is in general impossible to derive these bounds analytically.
- Recursive functions are banned, since it is impossible to predict the maximal depth of the recursion, but also because they are difficult to implement in integrated circuits.
- No dynamic creation of tasks is allowed, since we must know the number of tasks in advance to be able to calculate the workload on the different components of the hardware architecture.

One of the key ideas of hardware/software codesign is that the behavioural description should be written as independently as possible of the hardware architecture and the partitioning. But it has been pointed out before that a total separation between specification and implementation is often impossible[3], and therefore some design activities have to be interleaved and iterated, which is captured by the backwards-pointing arrows in figure 1 (b). The particular way we choose to express the behaviour is governed by things like clarity and—not to be neglected for high-performance systems—executional efficiency. Often we can find different ways of structuring the system into tasks, and one of the factors that can be manipulated is the task size. For a hardware implementation, it is quite obvious that a large number of small tasks is advantageous, since the hardware allows true parallelism. But if we instead want to implement the same thing in software, then there is a large penalty to pay each time a task switch occurs on the microprocessor, hence we would like to reduce the number of tasks as much as possible. So clearly, the way we write behavioural descriptions depends on the technology in which we believe that different parts of the system will be implemented.

3 Hardware architecture selection

Important for the successful application of hardware/software codesign is that the intended behaviour of the system is allowed to influence the selection of a hardware architecture to a greater extent than it does in current practice. One author has compared the selection of a hardware architecture to that of chosing a motor for a vehicle: he views the hardware as an engine that provides the horsepower needed to keep the software moving at the required rate[7]. Typically, the range of choices we have when selecting a hardware architecture includes:

- Which microprocessor or -processors to use.
- Whether any application-specific integrated circuits (ASICs) are to be designed, and in that case what their characteristics (size, etc.) should be.
- What memories are needed, and in what way they should be shared by the other components.

To decide on different features of the hardware architecture, it is important to have ways of assessing the consequences of a certain architectural choice, and hence we need models that enable us to estimate the impact of a decision on the different quality criteria. We are currently studying techniques that let one specify the hardware architecture using a description language, and from the characteristics of the different components in the description, a model is derived. This can be used to estimate different factors about the design, e.g. how fast a certain task runs on a specific micro-processor, or how large the circuit will be if we instead implement the same functionality in an ASIC. In the next section, we will see how these estimations about the hardware architecture, together with predictions about the operating system kernel, make it possible to calculate a quality measurement of the implementation.

4 Task scheduling

In most systems the hardware architecture consists of several computational resources, that are shared amongst many tasks. This includes microprocessors, which only let one task execute at a time, and memories that are shared between several tasks, but which do not allow concurrent access. This means that we will inevitably get into conflict situations where several tasks want to use the same resource simultaneously. These conflicts are resolved by the *scheduler*, which makes up the kernel of the operating system.

There are many different ways to organize the scheduler, and we have decided to base our heterogeneous kernel on *fixed-priority scheduling*[8, 9], where each task gets an invariable priority number. Whenever there is a conflict over some resource, the task with the highest priority gets to access it first. The major limitation, in connection with codesign, of previous work on fixed-priority scheduling is that it is assumed that only one processor exists in the system. But a heterogeneous system should be seen as a multiprocessor, where several tasks may execute in true parallelism, and where memories can be shared between different processors and ASICs. A further complication is that the analysis methods developed for these scheduling approaches only tells whether a task set fulfils its timing obligations or not, but what one really wants is a continuous measurement that makes it possible to compare the relative merits of different implementations.

4.1 Fixed-priority scheduling for codesign

To overcome these problems, we have extended the ideas of fixed-priority scheduling to make it useful for codesign[10]. We assume that we will define the scheduler by giving a priority order relation, $\tau' <_p \tau$, pronounced "τ' has higher priority than τ". This means that whenever τ is using a resource that τ' needs, the execution of τ is pre-empted by τ', and τ's duration will thus be extended by the duration of τ'. We will call this delay the *interference* $I(\tau, \tau')$ that τ' causes τ, and it is defined as the time during which τ' uses resources of the hardware architecture that are also used by τ. It is calculated for one activation of τ'.

We are now ready to define a measurement of how well an implementation fulfils the timing requirements, and this metric is based on the observation that every implementation could, in theory, reach the deadlines, if only the underlying hardware were executing fast enough. So then there is of course a minimal speedup $S(\tau)$ which is needed for a task τ to reach its deadline $D(\tau)$, and this can be calculated by the following equation:

$$S(\tau) = \frac{1}{D(\tau)} \left(C(\tau) + \sum_{\tau' <_p \tau} \left\lceil \frac{D(\tau)}{T(\tau')} \right\rceil I(\tau, \tau') \right)$$

Here, the function $C(\tau)$ gives the worst-case execution time of the task τ. We use the model of the hardware architecture described in the previous section to derive the values of the C and I functions. The term $\sum\lceil D(\tau)/T(\tau')\rceil \cdot I(\tau,\tau')$ gives the total interference caused by all higher-priority tasks τ'. The total duration is $C(\tau)$ plus the sum, and it is divided by the specified maximal duration $D(\tau)$, to get the minimal required speedup for the task.

Assuming that we have decided upon a certain priority order, we can easily check if the deadlines are met by calculating the minimal required speedup of each task, and if all these values are less than 1, we can assert that the deadlines will be guaranteed. We can also use the speedup values to compare the merits of two different priority orders: the one which gives the lowest value to $S^* = \max_{\tau \in P} S(\tau)$ is better since its deadlines will be guaranteed by every implementation that also guarantees those of the order with the higher S^*. Based on this, we have developed a procedure which calculates the optimal priority order in $O(m^2)$ time, for m tasks and relative to a specific hardware architecture and partitioning.

5 Partitioning

The partitioning is a way of deciding for each part—task—of the behavioural description, which part—microprocessor or ASIC—of the hardware architecture should be used to implement it. Therefore the partitioning can be thought of as a mapping, whose domain is the set of tasks in the behavioural description, and whose range is the set of components with processing capabilities in the hardware architecture (see Figure 2). Similarly, all variables in the behavioural description have to be mapped to elements in the architecture with storage capacity, i.e. memories or ASICs. The problem of constructing this mapping has an exponential complexity: if there are k components and m task, there are $O(k^m)$ different ways to partition. But fortunately, the problem lends itself well to a solution using a branch-and-bound algorithm, which only needs to evaluate a small fraction of all combinations in order to find the optimum.

The goal function that we try to optimize for partitioning is the schedulability of the system, captured by the minimal required speedup S^* of the task set. In this way, the partitioning is dependant on the performance estimations which are based on the models of the hardware architecture and scheduler. But often there might be other criteria besides performance that are relevant to the partitioning, but that can be difficult to quantify. As an example, the designer may know that one task in the behavioural description is likely to change in future versions of the system, and it is therefore preferably implemented in software. This, together with other similar arguments, leads us to the conclusion that the designer must be given the opportunity to manually guide the partitioning process.

Another consideration is the partitioning grain size. We have chosen tasks as the entities to assign to hardware or software, but one could of course use smaller parts, e.g. single instructions. The advantage of a smaller grain size is that we can adjust the partitioning more exactly than with a coarser grain. But on the other hand, the data dependencies between instructions are often much stronger than between tasks, so if we move some instructions to hardware, it can be problematic to execute hardware and software parts in parallel. We can also expect to get a larger runtime overhead with a smaller grain. Partitioning on task level on the other hand gives the user some control over the grain size, since he can often choose, to some extent, how big the tasks should be. On these grounds, we feel that a task-level partitioning is justified.

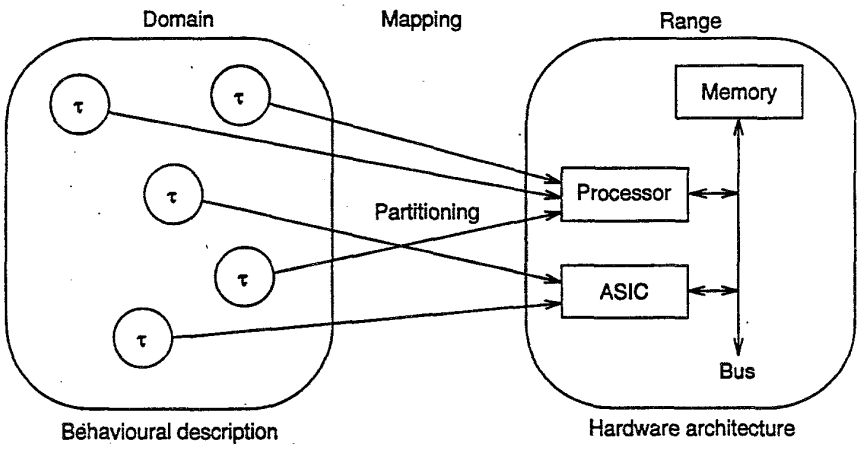


Figure 2: Hardware/software partitioning maps parts of the behavioural description onto parts of the hardware architecture.

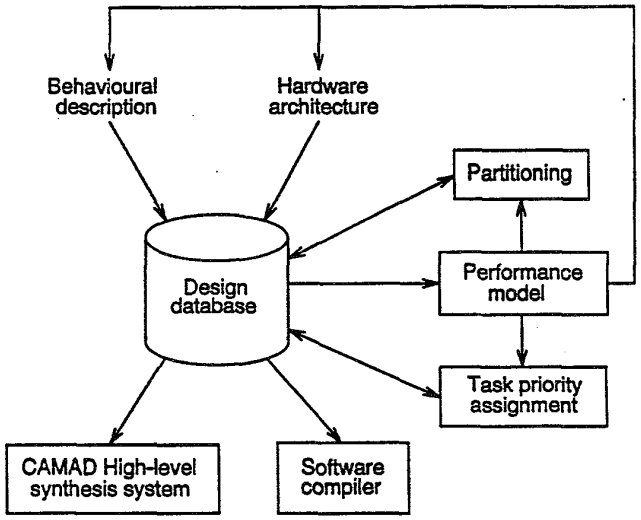


Figure 3: A codesign workbench for embedded real-time systems.

6 A codesign workbench

In this section, we will sketch out a workbench for codesign of embedded real-time systems, which we are currently developing. At the heart of the system is a design database containing the behavioural description and hardware architecture, and a performance model, which is used to estimate different characteristics (primarily schedulability) of a proposed implementation. Using this model, the designer will iteratively develop the behavioural description and the hardware architecture, and then the system derives an optimal partitioning and priority order automatically or with user assistance. A high-level synthesis system is needed to take the hardware part from behavioural specification to silicon, and we intend to use the CAMAD system, which has been developed within our group[11]. For the software part, any standard compiler could in principle be used.

The workbench will also provide possibilities to simulate a design, and to visualize different aspects of it. In the future, further tools can easily be added, for instance some support could be given to improve the hardware architecture.

7 Conclusions

In this paper, we have discussed an approach to hardware/software codesign which is based on a careful analysis of the design process which the embedded systems' engineer goes through, and we have suggested a workbench with tools aiding the search for an implementation that gives the desired performance. We have tried to show that hardware/software partitioning of real-time systems is intimately connected to three other parts of the design:

- The behavioural description structures the system as a collection of concurrent tasks that can be implemented in hardware or software. But by changing the task structure into another, functionally equivalent, structure, we might find new, and improved, ways to partition the system.
- The hardware architecture gives us the entities to which different tasks in the behavioural description may be assigned. If a satisfactory partitioning cannot be found, the solution may very well be to modify the hardware architecture, by removing the bottlenecks that cause the performance problem.
- The task schedulability provides a measurement of how well the timing requirements of the system are fulfilled, if we choose a certain way of partitioning the system.

It is our belief that to reach the design goals, the designer must find a well-balanced combination of a behavioural description, a hardware architecture and a partitioning, and this has to be achieved through an iterative process, where all three components are adjusted continuously. By providing good models of the performance of the hardware architecture and scheduler, different design alternatives can be rapidly evaluated, and thereby pave the way for a successful accomplishment of later design stages.

References

1. Klaus Buchenrieder. *Hardware/Software Co-Design: An Annotated Bibliography.* IT Press, 1994.
2. Peter D. Lawrence and Konrad Mauch. *Real-Time Microcomputer System Design: an Introduction.* McGraw-Hill, 1988.
3. William Swartout and Robert Balzer. On the inevitable intertwining of specification and implementation. In Narain Gehani and Andrew D. McGettrick, editors, *Software Specification Techniques*, pages 41–45. Addison-Wesley, 1986.
4. United States Department of Defence. *Ada Programming Language*, 1983.
5. Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang.* Prentice-Hall, 1993.
6. Eugene Kligerman and Alexander D. Stoyenko. Real-Time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, September 1986.
7. Wayne H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.
8. C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.
9. Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

10. Jakob Axelsson. Analysis and improvement of task schedulability in hardware/software codesign. Accepted to the 7th Euromicro Workshop on Real-Time Systems, June 14-16th, 1995.

11. Zebo Peng and Krzysztof Kuchcinski. Automated transformation of algorithms into register-transfer level implementation. *IEEE Transactions on Computer-Aided Design*, 13(2):150–166, 1994.