Abstract

To be done

Introduction

To be done

Problem description

With the term “embedded system” we refer to any information processing device embedded into another product. In the last years their diffusion has been growing at a surprising rate: automotive control engines, cellular phones, multimedia electronic devices are only a few examples of the pervasively of such devices.

Being widely employed in portable devices, this systems need energy efficient platforms with real time performance: Multiprocessor Systems on Chips (MPSoCs) are among the most appealing solutions to these issues.

MPSoCs are multi core, general purpose, architectures developed on a single chip; each core has low energy consumption and limited computational power: real time level performance is thus achieved by extensive parallelization.

Given a target application, usually described as a set of interdependent processes, to design a system amounts to allocate hardware resources to processes and to compute a schedule (Xie & Wolf 2000). Since these devices always run the same application in a well-characterized context, it is possible to spend a large amount of time for finding an optimal allocation and scheduling off-line and then deploy it on the field, instead of using on-line, dynamic (sub-optimal) schedulers (Culler 1999; Compton & Hauck 2002).

The architecture

The MPSoC model we consider consists of a pre-defined number of distributed Processing Elements (PE) as depicted in figure 1. All nodes are assumed to be homogeneous and composed by a processing core and by a low-access-cost local scratchpad memory.

The local memory is of limited size, therefore data in excess must be stored externally in a remote on-chip memory with higher latency, accessible via the bus.

Scratchpad memories, unlike caches, are managed at application level and statically partitioned before the execution starts. The bus for state-of-the-art MPSoCs is a shared communication resource, and serialization of bus access requests of the processors (the bus masters) is carried out at transaction basis by a centralized arbitration mechanism.

In some platforms each PE can independently be tuned to work at different frequencies, according to the computation workload required. This feature is referred to as Dynamic Voltage Scaling (DVS) and allows dramatic improvements in energy efficiency.

The application

The target application to be executed on top of the hardware platform is represented as a Task Graph (fig. 2). A TG is a directed acyclic graph \( \langle T, A \rangle \), where \( T \) is the set of nodes modeling generic tasks (e.g. elementary operations, subprograms, ...) and \( A \) the set of arcs modeling precedence constraints (e.g. due to data communication).

Each task has a five phase behavior (see fig. 3): it reads all communication queues (one per ingoing arc; phase INPUT), reads stored state information, if any (RS); then performs some computation (EXEC) and finally writes its state information (WS) and all output data for successor tasks (OUTPUT).

Computation, storage and communication memory requirements are annotated onto the graph and can be allocated either on the local or on the remote storage devices. Depending on the modelled application, some memory allo-
The inputs of the problem are the target platform description.

Case study 1: Allocation and Scheduling Problem

Simultaneously access the bus requiring a portion of the overall 

4

validated in (Benini et al. 2005) where each task can simul-

10

over, since modeling the bus at the granularity described in 

n

In the context of the hardware design problem introduced in the 

previous section, we consider four case studies coming from real scenarios. Although they are similar, each of them has peculiarities which set different issues.

In the following, we will describe our case studies, pointing out the characteristics making them interesting benchmarks for scheduling problems.

As a general remark, for scheduling purpose in all cases tasks were split into several activities to take into account the different bandwidth usage of each execution phase, and to allow a more realistic model of precedence relations. Moreover, since modeling the bus at the granularity described in section would make the problem overly complex, a more abstract additive bus model was devised, explained and validated in (Benini et al. 2005) where each task can simultaneously access the bus requiring a portion of the overall bandwidth.

Case study 1: Allocation and Scheduling Problem with deterministic task graphs (D-ASP)

The inputs of the problem are the target platform description (number of processing nodes, size of storage devices and bus bandwidth), and the task graph. Each task is annotated with a duration and memory requirements for program data, internal state and communication data.

The problem is to allocate resources (processing elements and memory slots) to tasks, and to compute a schedule; all the real-time constraints and all the capacity constraints on the resources have to be met.

The objective is to minimize the total amount of data transferred on the bus, being the communication resources one of the major bottlenecks in modern MPSoCs. The bus is used when a task allocates data on the remote memory and when two communicating tasks are allocated on different processors.

In this case study, we consider task graphs representing pipelines, in the sense that each task $T_i$ reads input data from task $T_{i-1}$ and writes output data for task $T_{i+1}$.

The pipeline workload is typical, for example, of multimedia streams encoding/decoding, where the same sequence of tasks is repeated an unknown number of times. To analyze the pipeline behaviour at working rate, several repetitions of the same task must be scheduled. In particular, if $n$ is the number of tasks in the pipeline, after $n$ repetitions the pipeline is at full rate; therefore, in the D-ASP $n^2$ repetitions of each task must be scheduled. This leads to additional precedence constraints stating that tasks must be executed in order: in other words, the $j$ − $th$ execution of a task $T_i$ must execute before the $(j + 1) − th$ execution and after the $(j − 1) − th$.

In a pipeline, the real-time requirements translate into throughput constraints: the time between two consecutive data supplied in output must be lower than a given value. This posts a constraint on the maximum time between two consecutive executions of the same task, and in particular of the last one in the pipeline.

In the D-ASP we deal with alternative unary resources (the processing elements), shared resources (the memories) and the bus.

The main difficulty of the problem is that the objective function and the scheduling constraints are conflicting one each other: while the objective function aims at packing as much tasks as possible on the same processing element (so as to minimize the communications), the real-time constraints suggest to parallelize the tasks execution on different processors (to reduce the pipeline throughput).

In addition, we can see that the objective function does not depend on the schedule, being the communications on the bus completely defined once tasks and memory requirements allocation are decided.

We have generated and solved D-ASP instances with a number of tasks up to 10 and a number of processors up to 9. We remind that, since each task is split into multiple activities (input data reading/writing, internal state reading/writing, execution) and we analyze a pipeline, we actually scheduled up to $5 \times 10^2$ activities.

Case Study 2: Allocation and Scheduling Problem with conditional task graphs (C-ASP)

Real applications to be implemented on top of MPSoCs platforms never exhibit a deterministic behavior: task durations...
are not known in advance, conditional branches (like if-then-
else statements) can be present and so on. It is thus worth-
while to take into account some of those elements of uncer-
tainty.

An interesting case is that of conditional branches, since
every computer program contains many of them, and they
drastically affect the application behavior. Moreover, tasks
on alternative paths can access the same resources at the
same time (see $l_1$ and $l_2$ in the schedule of fig. 4): therefore
taking into account branches allows better resource usage.

Explicitly modeling conditional branches turns a Task
Graph into a Conditional Task Graph. A CTG (see figure 4)
is a triple $\langle T, A, C \rangle$, where $T$ is the set of nodes/tasks, $A$ the
set of arcs/precedence constraints, and $C$ is a set of condi-
tions, each one associated to an arc, modeling what should
be true in order to choose that branch during execution. We
assume to know for each condition the probability $p(c_i)$ that
$c_i$ is true: code profiling or other techniques can be used to
estimate such probabilities.

Given a CTG with generic structure (not necessarily a pipe-
line) and a MPSoC platform, we want to compute an
optimal allocation of processing elements and storage de-
vices minimizing the expected value of the bus traffic; we
also want to provide a schedule guaranteed to meet a global
deadline constraint in all scenarios.

In the allocation (see fig. 4 for an example), local mem-
ory capacities cannot be exceeded; if the remote memory is
used to store program data, internal state or communication
queues, tasks generate bus traffic exactly as in the P-ASP.

As in the previous case tasks are split into activities,
whose duration depends on the memory allocation; in the
schedule we chose to assign a unique start time to each activ-
ity, regardless it executes or not (see fig. 4): this is a common
practice in the related literature (Brumbauer et al. 2003;
Shin & Kim 2003). Unlike in the P-ASP, tasks are executed
only once.

We solved the problem by devising deterministic formula-
tions for the stochastic elements. The complexity issues are
similar to those of P-ASP, stressed by the generic structure
of the graph and by the presence of conditional branches.

We ran experiments on two test sets. The first contains
slightly structured instances, i.e. with very short tracks
or even singleton nodes: due to this lack of precedence
relations computing a feasible schedule appears to be the
most challenging problem component. We were able to
solve slightly structured instances up to 6 PEs, 34 nodes,
64 scheduling activities.

On the opposite, instances of the second group are com-
pletely structured (one head, one tail, long tracks). Here the
high number of arcs, and thus of communication queues,
sets the allocation as the core problem. We were able to
solve instances in this group up to 6 PEs, 25 nodes, 95
scheduling activities.

Finally, we also considered the case when the objective
function to be minimized is the expected completion time
(makespan), instead of the communication cost. This made
the problem very difficult to solve with our approach, to the
point that we had to dismiss the allocation phase and focus
only on computing an optimal schedule for a fixed PE and
memory mapping.

We performed experiments on a third set of instances
with characteristics somehow in-between those of the other
groups. We were able to solve these “pure” scheduling prob-
lems up to 6 PEs, 105 activities.

**Case Study 3: Dynamic Voltage Scaling Problem
with pipelined deterministic task graphs (P-DVSP)**

As introduced in Section 3, recent MPSoCs platforms can tune the working frequency of each processing node separately.
In this context, the Dynamic Voltage Scaling Problem arises.
The P-DVSP we consider energy-aware MPSoCs. The
problem input is the description of the platform and the task
graph. In particular, the platform is described through the
number of processing elements and a set of frequencies each
processor can run, with the energy consumption at each fre-
quency. In addition, we have a time overhead and an energy
cost for switching from each frequency to each other.

For this case study, the task graph is a pipeline. Each
task is annotated with the duration (in clock cycles) and the
size of communication data. Memory capacity constraints
are not considered, assuming that each memory slot is large
enough to contain all the data necessary for the execution.

The problem is to allocate tasks to processors, decide a
running frequency for each task and schedule the task exe-
cution, meeting all the real-time constraints. The objective
is to minimize the total energy consumption: energy is con-
sumed when a task executes, when two tasks executing on
different processors communicate using the bus and when a
processor switches its frequency. The power minimization
is important, for example, when the MPSoC is employed in
a battery-operated device such as a mobile phone.

In the P-DVSP the number of alternative resources is
higher that the D-ASP, because here each processor at each
frequency is an alternative.

To fulfill the real-time requirements longer tasks must
usually execute at higher speeds, and this gives space for the
shorter tasks to execute at a lower speed reducing the en-
ergy consumption. In this manner, it can be the case that a
number of tasks execute on the same processor at different
speeds: scheduling two tasks running at different speed one
just after the other causes an energy consumption that af-
facts the objective function and a time overhead that affects
the makespan. As for the D-ASP case study, allocation and
scheduling are somehow conflicting.
We have generated and solved P-DVSP instances with up to 10 processors (able to run from 3 to 6 different speeds) and 10 tasks. Each task is composed of the execution and the communication data reading/writing activities and is scheduled several times, thus we scheduled up to $3 \times 10^4$ activities.

Case Study 4: Dynamic Voltage Scaling Problem with generic deterministic task graphs (G-DVSP)

We address the same problem described as case study 3 considering generic task graphs. The main difference is that a task can possibly read data from more than one preceding task and possibly write data that will be read by more than one following task. The number of reading and writing activities can become considerably higher, being higher the number of edges in the task graph. This leads to a higher parallelism between tasks and thus the problem becomes much harder.

In the G-DVSP (as in the C-ASP) we consider a single repetition of the task graph. Real-time constraints are imposed on processors and tasks deadline: each task must end the execution before a given time and the computational workload of each processor must be carried out before a given time.

We generated and solved G-DVSP instances with up to 6 processors and 76 activities (14 execution tasks and 62 communication activities between them). As one can see, the size of the instances we can solve is lower than the other case studies. In addition, we experimentally found that, even if in the mean case the search time is comparable with the other case studies, some instances are extremely hard to solve. Typically these instances have task graphs with high parallelism or an optimal solution where the task end times are very close to the deadlines.

Related work

The mapping and scheduling problems on multi-processor systems have been studied extensively in the past. An early example is represented by the SOS system (Prakash & Parker 1992). SOS considers processor nodes with local memory, connected through direct point-to-point channels. The algorithm does not consider real-time constraints.

All the case studies introduced in the previous section have been considered in other works. The pipelined workload, typical of several real applications, as been studied, for example, in (Chatha & Vemuri 2002) and (Fohler & Ramamritham 1997). Energy-aware platforms have been studied in several works; the first DVS approach for single processor systems which can dynamically change the supply voltage over a continuous range is presented in (Yao, Demers, & Shenker 1995). More recent works on the argument can be found in (Xie, Martonosi, & Malik 2005), (Jeurikar & Gupta 2005), (Andrei et al. 2004), to cite few.

Different platforms and task graphs have been considered: (Thorsteinsson 2001) considers a multi-processor platform where the processors are dissimilar; (Palazzari, Baldini, & Coli 2004) consider a task graph of periodic tasks with some aperiodic tasks; (Grossmann & Jain 2001) work on a scenario similar to our case study 4.

Different objective functions can be also taken into account. A good survey of several objective function typical of scheduling problems can be found in (Hooker 2004).

Many researchers have also worked extensively on the problem of allocating and scheduling conditional, precedence-constrained tasks on processors in a distributed real time system, extremely important in the system design community (Xie & Wolf 2000). Different models have been considered: for example, (Beck & Wilson 2005; 2004) consider stochastic activity durations; (Vilím, Barták, & Cepek 2005) and (Beck & Fox 2000; 1999) deal respectively with optional or alternative activities; optimization of bus access in the context of CTG scheduling is considered in (Pop, Eles, & Peng 2000).

Whatever platform or problem description is considered, it is important to test the quality of the solving tool on appropriate instances, able to describe hard problems that turns out to be also realistic. In (Davidovic & Crainic 2006) the authors analyze in deep the characteristics of multi-processor scheduling problems with communication delays (MSPCD) proposing an accurate instance generator able to create benchmarks with realistic platforms and task graphs. (Kwok & Ahmad 1999) propose a set of MSPCD instances to compare 15 scheduling algorithms described in literature. A set of benchmarks is also proposed in (Coll, Ribeiro, & de Souza 2002) and (Tobita & Kasahara 2002), but communication delays are not accounted; in particular, (Coll, Ribeiro, & de Souza 2002) consider smaller instances, but consider platforms with dissimilar processors. (Hall & Posner 2001) present an instance generator independent of the problem characteristic, but useful to evaluate and compare different solving algorithms.

About the Instances

The study of allocation and scheduling problems on MP-SoCs required to collect and build a large amount of instances, for many different purposes.

In particular, we performed our tests on three type of instances: random graphs and platforms, randomly generated synthetic benchmarks and real applications.

Random instances

To verify the effectiveness of a solution method, as well as to get some insight on the problem structure it is crucial to be able to perform tests on a large number of relevant, possibly well known instances.

As for the relevance, real world instances are the best choice: unluckily they are difficult and time consuming to build. To overcome this problem we realized a random instance generator.

Since some real world instances were available, we chose to devise a graph generation algorithm able to mimic their peculiarities; in particular, since real applications are often quite structured, the generator was primarily designed to build graphs of that sort.

At the same time, it is very useful for evaluation and research purpose to have instances with specified properties; therefore, each step of the generation procedure has to be
controlled with high precision: in our case this is done by means of randomized input parameters.

We used randomized parameters of two types, respectively described by their span (min, max) or their mean and variance. In both cases a probability distribution law can be specified (uniform, square, exponential; Gaussian probability is to be added). Figure 5 shows the distribution functions which can be modelled.

The algorithm we use is described step by step in the following (fig. 6), where we refer as “fork” to any node with more than one outgoing arc, and as “join” to any node with more than one ingoing arc. As an example, the generation of the graph in figure 2 is considered.

A. Head generation: A random number of head nodes is generated, based on a heads number parameter

B. Tails generation: The minimum specified number of tail nodes (tails number parameter) is generated and connected 1 to 1 to the head nodes.

C. Expansion loop: A random arc with a non fork source and non join destination is chosen. The arc is replaced with a random number of series. The operation is controlled by a branching factor and series length parameter. If no suitable arc exists a single 1 length series is generated. The process is repeated until the maximum number of nodes (nodes number parameter) is reached.

D. Tails completion: A random number of tail nodes is picked, and arcs are consequently removed to turn join nodes into tails.

E. Arc insertion: Extra arcs are added according to another generation parameter (extra arcs number)

Each time a node or arc is created the attributes of the associated task/communication queues are computed; again the operation is controlled by random parameters. A deadline value is computed by multiplying the length of the longest path for a specified factor. Finally, the algorithm can also produce conditional graphs.

A second, much simpler, generator was also realized to build suitable platforms for a given graph. The computed number and attributes of hardware resources (PEs, memory devices, etc.) are based on the graph and can be controlled (again) by means of random parameters.

Randomly generated synthetic benchmarks
Since we defined the problem by making some simplifying assumptions, there is a certain misalignment between our model and the real embedded system. If the solutions we provide are too far from reality they are of no use: specific instances and tests had to be devised to evaluate the entity of such misalignment.

We therefore generated a large set of instances with the same algorithm described above; such instances were turned into synthetic applications by mapping nodes to real processes performing some computation and communicating some data.

Such applications were then executed on a MPSoC platform simulator (MPARM) to compute all non structural graph properties (such as task durations, memory requirements, etc...).

Finally, the instances were solved: the optimal allocation and schedule was implemented and its real cost, resource usage and execution time compared against the theoretical one. In the comparison the model we provided resulted very accurate.

Realistic applications
When dealing with a real problem the most interesting thing is always to tackle real instances. Therefore, a limited number of the instances we used represented applications of practical use, such as MPEG or GSM encoding/decoding.

In these cases a careful parsing step was needed to recognize and extract a task graph representation from the application code. Using MPARM, each task has been run alone on a processor a large number of times, collecting the mean and
the worst execution and communication times. These values were then re-injected into the graph. Finally, the resulting instances were solved and evaluated as in the previous case.

References


