Challenging Scheduling Problem in the field of System Design

Alessio Guerri and **Michele Lombardi** and **Michela Milano**
Department of Electronics, Computer Science and Systems (DEIS)
Alma Mater Studiorum - University of Bologna
Viale Risorgimento 2, 40136, Bologna, Italy
fax: +39 051 20 93073
emails: {aguerri, mlombardi, mmilano}@deis.unibo.it

Abstract

Scheduling problems are of primary importance in many logistical, design and industrial applications. They can be very heterogeneous and often mix with a resource allocation phase which makes the problem particularly challenging.

A number of solution techniques has been developed to deal with such problems. It is therefore important, for evaluation purposes, to dispose of relevant instances, possibly representing real problems.

In this paper we present a real world allocation and scheduling problem from the field of embedded system design; in this context, we describe four variants and devise an instance generation algorithm.

Introduction

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 few examples of the pervasivity of such devices.

Being widely employed in portable devices, these 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 task graph, 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).

In this paper we propose some variants of allocation and scheduling problems arising in the system design context. We propose four case studies we have faced so far, concerning deterministic and conditional task graphs, different objective functions and different graph structures. These problems contain important aspects to be considered: alternative resources, variable durations, alternative routes in the task graph representing the application, discrete, additive and unary resources.

We believe that the problem classes we took into account (pure scheduling problems, mixed allocation and scheduling problems, both deterministic and stochastic) could be good application candidates for a scheduling contest. In addition, we worked on both random and real world instances: we think that a benchmark for a scheduling competition should take into account both cases to effectively test the quality of a scheduling tool.

Finally, we have implemented an instance generator, described in the paper, that could be used for generating "realistic" instances[1].

Problem description

In this Section we describe the aspects common to the four case studies we considered. In particular, we describe the architecture of the MPSoC platform (and thus the hardware resources to be allocated to processes) and the characteristics of the applications to be executed on top of it.

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. Modeling

[1]The generator is available at: http://www-lia.deis.unibo.it/Staff/AlessioGuerri/Sched_LIB/

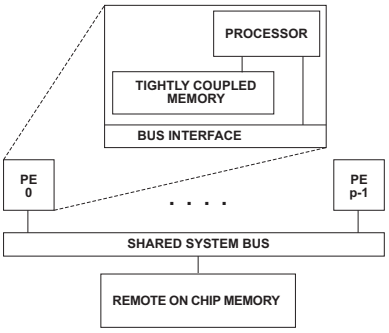

Figure 1: Single chip multi-processor architecture.

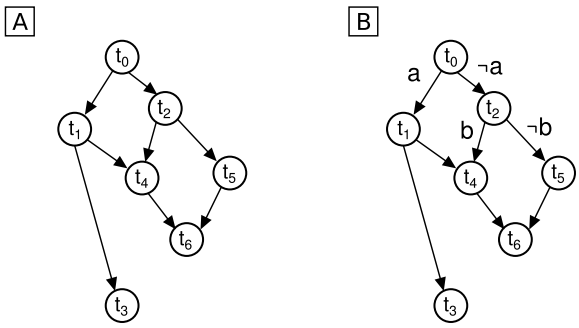

Figure 2: A task graph and a conditional task graph

the bus at such a granularity would make the problem overly complex, therefore 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.

In some platforms each PE can independently be tuned to work at different frequencies, according to the required computation workload. 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 (TG). A TG (see figure 2A) 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).

Indeed, real applications 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. Sometimes it is thus worthwhile to take into account some of those elements of uncertainty. An interesting case is that of conditional branches, since every computer program contains many of them, and they drastically affect the application behavior.

Explicitly modeling conditional branches turns a TG into a Conditional Task Graph (CTG). A CTG (see figure 2B) 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 conditions, 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.

Tasks in the application communicate by writing/reading memory buffers (communication queues) and can have internal state information. Their behaviour can be described by five phases (see figure 3): they read all input queues (one per ingoing arc; phase `INPUT`), read stored state information, if any (`RS`); then perform some computation (`EXEC`) and finally write their state information (`WS`) and all output data for successor tasks (`OUTPUT`).

Memory requirements for each phase (program data, internal state and communication queues) are annotated onto the graph and can be allocated either on the local or on the remote storage devices. In particular, communication queues can be locally allocated only if both the sender and the receiver tasks execute on the same PE.

In case of remote memory allocation, tasks need to access the bus to read/write data. As introduced above, the bus access is granted or denied by an arbitration mechanism. However, under the additive model assumption, bus requests are defined in terms of bandwidth usage. Each task phase has a different bandwidth requirement: communication and internal state access are bus intensive, and thus use a large portion of the total bandwidth. On the opposite in the execution phase bus is accessed rarely, only for refill cycles following a cache miss, and thus the bus usage is much lower.

Figure 3: Execution model of a task

The worst case execution time (WCET) is specified for each node/task and plays a critical role whenever application real time constraints are to be met. Tasks duration depends both on memory allocation choices and frequency assignments on processing elements (in case DVS is supported). Communication and state access phases can last up to three or four times more if memory is allocated in the remote device rather than on the local scratchpad. An analogous difference, although much lower, affects also the execution phase. Frequency assignment influences the duration in an intuitive way.

There is a complex interaction between bus usage and completion time (makespan): to avoid bus congestion one would be tempted to pack as many tasks as possible on the same PE, in order to allocate communication queues on local memories. Unluckily, in this way the application parallelism is not exploited and the completion time increases, despite local queue allocation tends to reduce task durations. In practice the presence of real time constraints often forces to strongly parallelize the application: this decreases the makespan, but increases the bus usage as well.

The basic problem is to allocate hardware resources (processing elements, storage devices, working frequencies, bus

bandwidth) to tasks and to provide a schedule meeting real time constraints. We considered several variants of this hardware design problem, with different graph structure, platform features and cost function.

Case studies

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 a scheduling contest.

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.

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 processing elements 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 5×10^2 activities.

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

The C-ASP case study is similar to the D-ASP, but we consider generic TGs and we explicitly take into account the presence of conditional branches in target application.

Given a CTG with generic structure and a MPSoC platform, we want to compute an optimal allocation of processing elements and storage devices 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, local memory 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 D-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 activity, regardless it executes or not (see figure 4): this is a common practice in the related literature (Brunnbauer *et al.* 2003; Shin & Kim 2003). Unlike in the D-ASP, tasks are executed only once.

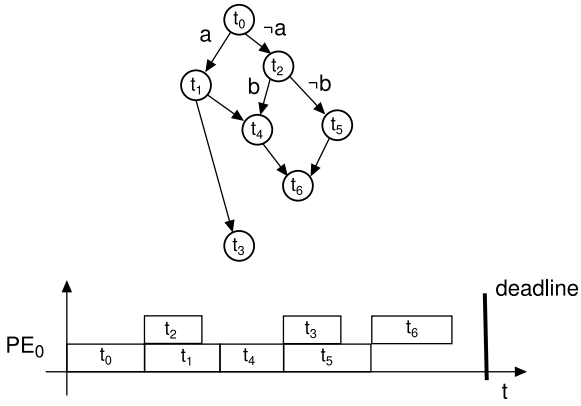

Figure 4: Reference Conditional Task Graph

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 tasks, split into 64 activities.

On the opposite, instances of the second group are completely 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 tasks, 95 activities.

Finally, we also considered the case when the objective function to be minimized is the expected 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 problems up to 6 PEs, 105 activities.

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

As introduced in the Problem description section, recent MPSoC platforms can tune the working frequency of each processing node separately. In this context, the Dynamic Voltage Scaling Problem arises. In 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 frequency. 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 execution, meeting all the real-time constraints. The objective is to minimize the total energy consumption: energy is consumed 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.

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 energy 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 affects 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 PEs (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 3×10^2 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 PEs 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, has been studied, for example, in (Benini *et al.* 2005), (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), (Jejurikar & Gupta 2005), (Andrei *et al.* 2004), (Andrei *et al.* 2006), (Benini *et al.* 2006), 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, &

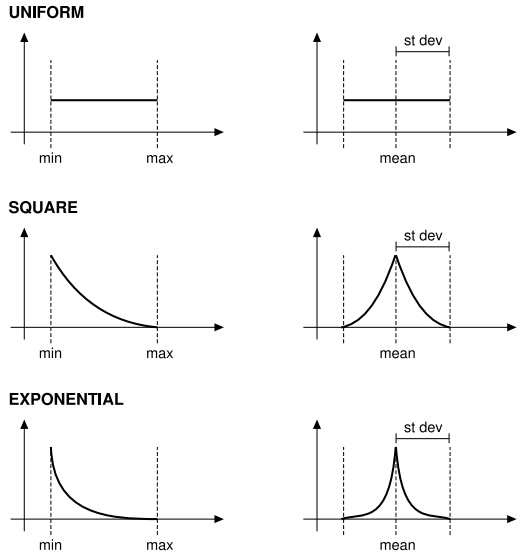

Figure 5: Possible probability distributions of a random parameter

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 functions 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). Optimization of bus access in the context of CTG scheduling is considered in (Lombardi & Milano 2006) and (Pop, Eles, & Peng 2000).

Among related approaches, (Beck & Wilson 2005; 2004) consider stochastic activity durations; (Vilím, Barták, & Cepek 2005) and (Beck & Fox 2000; 1999) deal respectively with optional or alternative activities.

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 multiprocessor 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 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 MPSoCs 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[2].

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, for evaluation and research purposes (e.g. to perform complexity characterization) it is very important to be able to generate instances with certain desired properties (e.g. with different level of parallelism, branching factors, number of precedence relations); 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 that can be modelled in our generator.

The algorithm we use is described step by step in the following, 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. An example of the graph generation steps is depicted in figure 6.

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.

[2] A repository of allocation and scheduling problem instances can be found at http://www-lia.deis.unibo.it/Staff/AlessioGuerri/Sched_LIB/

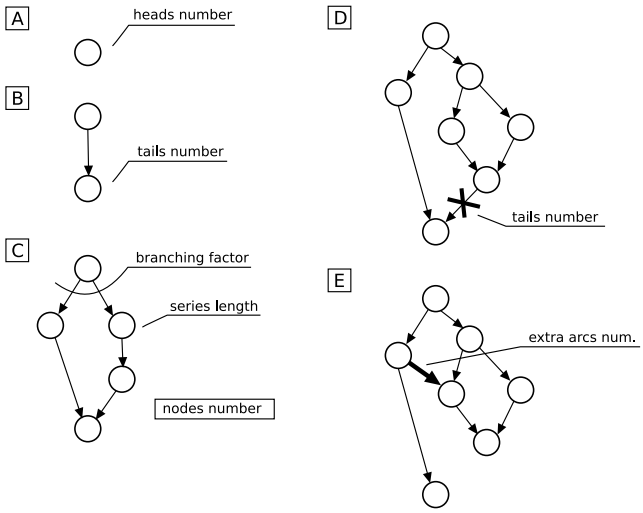

Figure 6: Task graph generation algorithm

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. At the present time the attribute sets it generates are those of the case studies we considered, but we are currently working to make it able to generate any node and arc attribute.

A deadline value is computed by multiplying the length of the longest path for a specified factor. The file format of the generated instances is compliant with that of TGFF, an instance generator used in the system design community (Dick, Rhodes, & Wolf 1998). 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 MPARM[3], a MPSoC platform simulator, 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.

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.

Considerations on the instance collection

We worked on several kinds of problem instances: random, real and realistic, both for the deterministic and the stochastic case; we believe all these problem classes should be taken into account when selecting a benchmark for a scheduling competition.

In fact, on one hand the performance of a scheduling tool has to be tested against problems with different characteristics, graph structure, resources, etc. Randomly generated instances enable us to perform this kind of tests. At the same time a scheduler, in order to be any useful, must be able to propose executable solutions to real world instances: it is therefore important to test it on real or at least realistic instances.

Finally, given that real world is quite never deterministic, a scheduling tool must not only tackle deterministic instances, but also be able to deal with stochastic scenarios without loosing efficiency.

We believe that the kind of instances proposed make possible to test the strength of a scheduling tool when facing a large variety of cases that can possibly happen in the real world.

Conclusions

We presented an allocation and scheduling problem arising in the design flow of modern embedded systems. In particular we considered four problem variants, with different cost function (bus traffic and energy consumption), graph structure (pipeline, generic, conditional), platform features.

We analyzed their peculiarities and we realized a flexible instance generator; we characterized a number of generated task graphs by means of synthetic benchmarks and we collected a small number of instances representing real applications.

[3]See the project homepage at http://www-micrel.deis.unibo.it/sitonew/research/mparm.html

We believe the problem we described to be of great interest, both for its complexity and for its practical uses.

Planned future developments include the improvement of the instance generation procedure with two main purposes: to build graphs with any kind of structure and to easily incorporate different application specific attributes (e.g. duration, memory requirements, available working frequencies).

We are also starting to tackle new problem variants: for example the case when the exact value of task durations is not know, but only a lower and an upper bound, and possibly a probability distribution; the goal is to provide a flexible schedule with guaranteed feasibility, or maximizing some robustness measure.

Another interesting case is that of platforms running multiple applications/task graphs at the same time. In this case an optimal allocation and schedule must be provided for each possible groups of TGs, together with a transition table describing how to switch from a configuration to another.

References

Andrei, A.; Schmitz, M.; Eles, P.; Peng, Z.; and Al-Hashimi, B. 2004. Overhead-conscious voltage selection for dynamic and leakage power reduction of time-constraint systems. In *Procs. of DATE2004*, 518–523.

Andrei, A.; Benini, L.; Bertozzi, D.; Guerri, A.; Milano, M.; Pari, G.; and Ruggiero, M. 2006. A cooperative, accurate solving framework for optimal allocation, scheduling and frequency selection on energy-efficient mpsocs. In *Procs. of SOC-2006*.

Beck, J. C., and Fox, M. S. 1999. Scheduling alternative activities. In *Procs. of AAAI/IAAI-99*, 680–687.

Beck, J. C., and Fox, M. S. 2000. Constraint-directed techniques for scheduling alternative activities. *Artificial Intelligence* 121(1-2):211–250.

Beck, J. C., and Wilson, N. 2004. Job shop scheduling with probabilistic durations. In *Procs. of ECAI-04*, 652–656.

Beck, J. C., and Wilson, N. 2005. Proactive algorithms for scheduling with probabilistic durations. In *Procs. of IJCAI-05*, 1201–1206.

Benini, L.; Bertozzi, D.; Guerri, A.; and Milano, M. 2005. Allocation and scheduling for mpsocs via decomposition and no-good generation. In *Procs. of CP-2005*, 107–121.

Benini, L.; Bertozzi, D.; Guerri, A.; and Milano, M. 2006. Allocation, scheduling and voltage scaling on energy aware mpsocs. In *Procs. of CPAIOR-2006*, 44–58.

Brunnbauer, W.; Wild, T.; Foag, J.; and Pazos, N. 2003. A constructive algorithm with look-ahead for mapping and scheduling of task graphs with conditional edges. In *DSD*, 98–103. IEEE Computer Society.

Chatha, K. S., and Vemuri, R. 2002. Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10(3):193–208.

Coll, P. E.; Ribeiro, C. C.; and de Souza, C. C. 2002. Test instances for scheduling unrelated processors under precedence constraints. Technical report. http://www-di.inf.puc-rio.br/ celso/grupo/readme.ps.

Compton, K., and Hauck, S. 2002. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys* 34(2):171–210.

Culler, D.A., S. J. 1999. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann.

Davidovic, T., and Crainic, T. G. 2006. Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multiprocessor systems. *Computers & Operations Research* 33:2155–2177.

Dick, R.; Rhodes, D.; and Wolf, W. 1998. Tgff task graphs for free. *codes* 00:97.

Fohler, G., and Ramamritham, K. 1997. Static scheduling of pipelined periodic tasks in distributed real-time systems. In *Procs. of EUROMICRO-RTS97*, 128–135.

Grossmann, I. E., and Jain, V. 2001. Algorithms for hybrid MILP/CP models for a class of optimization problems. *INFORMS Journal on Computing* 13(4):258–276.

Hall, N. G., and Posner, M. E. 2001. Generating experimental data for computational testing with machine scheduling applications. *Operations Research* 49:854–865.

Hooker, J. N. 2004. Planning and scheduling by logic-based Benders decomposition. Technical report. http://web.tepper.cmu.edu/jnh/planning.pdf.

Jejurikar, R., and Gupta, R. 2005. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Procs. of DAC2005*, 111–116.

Kwok, Y. K., and Ahmad, I. 1999. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing* 59(3):381–422.

Lombardi, M., and Milano, M. 2006. Stochastic allocation and scheduling for conditional task graphs in mpsocs. In *Procs. of CP-2006*, 299–313.

Palazzari, P.; Baldini, L.; and Coli, M. 2004. Synthesis of pipelined systems for the contemporaneous execution of periodic and aperiodic tasks with hard real-time constraints. In *Procs. of IPDPS04*, 121–128.

Pop, P.; Eles, P.; and Peng, Z. 2000. Bus access optimization for distributed embedded systems based on schedulability analysis. In *Procs. of DATE-00*, 567.

Prakash, S., and Parker, A. 1992. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing* 16(4):338–351.

Shin, D., and Kim, J. 2003. Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. In Verbauwhede, I., and Roh, H., eds., *ISLPED*, 408–413. ACM.

Thorsteinsson, E. S. 2001. A hybrid framework integrating mixed integer programming and constraint programming. In *Procs. of CP2001*, 16–30.

Tobita, T., and Kasahara, H. 2002. A standard task graph set for fair evaluation of multi-processor scheduling algorithms. *Journal of Scheduling* 5(5):379–394.

Vilím, P.; Barták, R.; and Cepek, O. 2005. Extension of $O(n \log n)$ filtering algorithms for the unary resource constraint to optional activities. *Constraints* 10(4):403–425.

Xie, Y., and Wolf, W. 2000. Co-synthesis with custom asics. In *Procs. of ASP-DAC-00*, 129–134. ACM.

Xie, F.; Martonosi, M.; and Malik, S. 2005. Bounds on power savings using runtime dynamic voltage scaling: an exact algorithm and a linear-time heuristic approximation. In *Procs. of ISPLED05*, 287–292.

Yao, F.; Demers, A.; and Shenker, S. 1995. A scheduling model for reduced CPU energy. In *Procs. of FOCS95*, 374–382.