A Constraint Programming Approach for Allocation and Scheduling on the CELL Broadband Engine

Luca Benini, Michele Lombardi, Michela Milano, and Martino Ruggiero

(1) DEIS, University of Bologna
V.le Risorgimento 2, 40136, Bologna, Italy
{lbenini,mmilano,mlombardi,mruggiero}@deis.unibo.it

Abstract. The Cell BE processor provides both scalable computation power and flexibility, and it is already being adopted for many computational intensive applications like aerospace, defense, medical imaging and gaming. Despite of its merits, it also presents many challenges, as it is now widely known that is very difficult to program the Cell BE in an efficient manner. Hence, the creation of an efficient software development framework is becoming the key challenge for this computational platform.

We have developed a novel software toolkit, called Cellflow, which enables developers to quickly build multi-task applications for Cell-based platform. We support programmers from the initial stage of their work, through a development-time software infrastructure, to the final stage of the application development, proposing a safe and easy-to-use explicit parallel programming model.

A fundamental component of the software toolkit is the off-line allocator and scheduler that manages hardware resources while optimizing performance metrics such as execution time, allocation costs, power. The optimization engine receives as input a task graph representing an application, the hardware resources and produces an optimal allocation and scheduling. We have developed various approaches, either based on decomposition [5] or based on pure Constraint Programming, this latter being the core of this paper. We have identified instance features that guide toward the choice of the best solver for the instance at hand.

Experimental result show that Constraint Programming (possibly combined with Integer Programming) is a proper tool for dealing with this kind of applications achieving very good performance.

1 Introduction

Single-chip multicore platforms are becoming widespread in high-end embedded computing applications (networking, communication, graphics, signal processing). The Cell Broadband Engine is probably one of the highest-volume multicore platforms in use today, targeting interactive graphics and advanced signal processing[1]. It is a heterogeneous multi-core architecture composed by a standard general purpose microprocessor (called PPE), with eight coprocessing units (called SPEs) integrated on the same chip. The SPE is a processor designed for streaming workloads, featuring a local memory, and a globally-coherent DMA (DIrect Memory Access) engine [15], [28].

[1] Sony's Playstation 3, powered by Cell BE, had sold more than 10M pieces at the end of 2007.

P.J. Stuckey (Ed.): CP 2008, LNCS 5202, pp. 21–35, 2008.

The heterogeneity of its processing elements and, above all, the limited explicitly-managed on-chip memory and the multiple options for exploiting hardware parallelism, make efficient application design and implementation on the Cell BE a major challenge. Efficient programming requires one to explicitly manage the resources available to each SPE, as well the allocation and scheduling of activities on them, the storage resources, the movement of data and synchronization. As a result, even with the help of APIs and advanced programming environments, programming Cell in an efficient fashion is a daunting task. Therefore, significant effort is being focused on the development of software optimization tools and methods to automate the mapping of complex parallel applications onto the Cell BE platform.

The final goal of this work is to enable developers to quickly build multi-task applications using a high-level explicitly parallel programming model. Low-level compilers and hardware-optimized core functions are provided by the the SDK from IBM [12]. However, the basic SDK does not offer any facility for optimizing the resource utilization in terms of both allocation and scheduling, memory transfers and utilization. We want to set programmers free from the issue of managing allocation and scheduling tasks, so they can focus on developing the core algorithms of the application.

The allocation and scheduling problems that are at the core of the mapping task are quite large and extremely challenging, and they are usually tackled using incomplete approaches. Even though incomplete approaches can be computationally efficient, they generally produce sub-optimal solutions. This is a significant shortcoming especially for demanding applications with tight execution time constraints, as incomplete optimizers may fail to find a feasible solution even when it does exist. Hence, efficient complete approaches are of great practical interest: not only they help programmers in taking hard design decisions, but also they can significantly extend the size and complexity of applications that can be run on the target hardware platform while meeting performance constraint.

For the problem at hand we have developed two approaches. One is based on Logic Based Benders Decomposition [8], and in particular on a recursive application of the technique. This approach has been proposed in [5] and will be recalled here for making the paper self contained. The second approach, which is the core of the present paper, is a pure CP model targeting both allocation and scheduling. We have experimentally compared the two approaches and identified instance features that guide toward the choice of the best solving strategy.

2 The Problem

The current design methodology for multicore systems on chip is hampered by a lack of appropriate design tools, leading to low efficiency and productivity. Software optimization is a key requirement for building cost- and power-efficient electronic systems, while meeting tight real-time constraints and ensuring predictability and reliability, and is one of the most critical challenges in today's high-end computing.

Embedded devices are not general purpose, but run a set of predefined applications during the entire system lifetime. Therefore software compilation can be optimized once for all at design time thus improving the performance of the overall system. Thus,

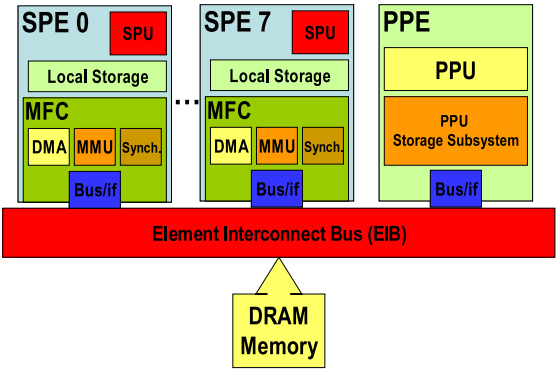

Fig. 1. Cell Broadband Engine Hardware Architecture

optimization is a critical component in the design of next-generation, highly program mable, intelligent embedded devices.

We focus on a well-known multicore platform, namely the IBM Cell BE processor (described in section 2.1), and we address the problem of allocating and scheduling its processors, communication channels and memories. The application that runs on top of the target platform is abstracted as a task graph (described in section 2.2). Each task is labelled with its execution time, memory and communication requirements. Arcs in the task graph represent data dependencies and communications between pairs of tasks. The optimization metric we take into account is the application execution time that should be minimized.

2.1 Cell BE Hardware Architecture

In this section we give a brief overview of the Cell hardware architecture, focusing on the features that are most relevant for our optimization tools. Cell is a non-homogeneous multi-core processor [32] which includes a 64-bit PowerPC processor element (PPE) and eight synergistic processor elements (SPEs), connected by an internal high bandwidth Element Interconnect Bus (EIB) [29]. Figure 1 shows a pictorial overview of the Cell Broadband Engine Hardware Architecture. The PPE is dedicated to the operating system and acts as the master of the system, while the eight synergistic processors are optimized for computation-intensive applications. The PPE is a multithreaded core and has two levels of on-chip cache. However, the main computing power of the Cell processor is provided by the eight SPEs. The SPE is a computation-intensive coprocessor designed to accelerate media and streaming workloads [27]. Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE.

Efficient SPE software should heavily optimize memory usage, since the SPEs operate on a limited on-chip memory (only 256 KB local store) that stores both instructions and data required by the program. The local memory of the SPEs is not coherent with

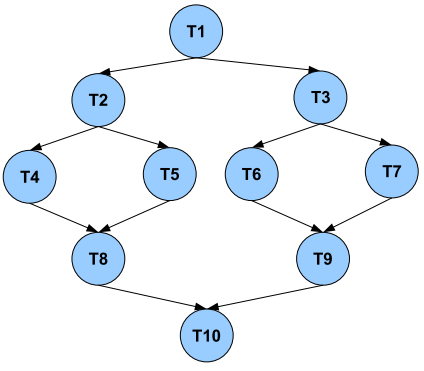

Fig. 2. Example of task graph

the PPE main memory, and data transfers to and from the SPE local memories must be explicitly managed by using asynchronous coherent DMA commands.

2.2 The Target Application

The target application to be executed on top of the hardware platform is input to our methodology, and for this purpose it must be represented as a task graph. This latter consists of a graph pointing out the parallel structure of the program. The application workload is therefore partitioned into computation sub-units denoted as tasks, which are the nodes of the graph. Graph edges connecting any two nodes indicate task dependencies due to communication and/or synchronization. Tasks communicate through queues and each task can handle several input/output queues. For example task $T9$ in Figure 2 reads two input queues from tasks $T6$ and $T7$ and writes an output queue for task $T10$.

Task execution is modeled and structured in three phases (see Figure 3): all input communication queues are read (Input Reading), task computation activity is performed (Task Execution) and finally all output queues are written (Output Writing). Each phase consists of an atomic activity. Each task also has two kinds of associated memory requirements:

1. Program Data: storage locations are required for computation data and for processor instructions;
2. Communication queues: each task needs queues to transmit and receive messages to/from other tasks, eventually mapped on different SPEs.

Both these memory requirements can be either allocated on the local storage of each SPE or in the shared memory (DRAM in Figure 1).

Fig. 3. Three phases behavior of Tasks

Durations are linked to the allocation choices: the duration of an execution phase in case of remote allocation of program data ($dmax^{ex}$) is greater than in case of a local allocation $dmin^{ex}$. Writing (and reading) operations have their minimum possible value ($dmin^{wr}, dmin^{rd}$) if the communication queue is on the local memory of the producer (resp. consumer) tasks, a higher value ($dmed^{wr}, dmed^{rd}$) if it is allocated on the local memory of the consumer (resp. producer) task, an even higher value ($dmax^{wr}, dmax^{rd}$) in case of remote allocation (on the on-chip DRAM memory).

3 Why CP

The main goal of this paper is to apply software optimization for maximizing the exploit of the hardware resources of the CELL BE architecture.

Scientific literature related to our problem explores many directions: we here recall the main research trends:

- exploitation of heterogeneous parallelism provided by the CELL architecture possibly performing automated scheduling and allocation;
- software optimization for other (yet similar) multicore platforms.

The Cell architecture supports a wide range of heterogeneous parallelism levels. To our knowledge, prior work is mainly focused on trying to exploit fine grained parallelism of Cell, such as at instruction and functional level, while our work is one of the few approaches at task level. In [14] authors present a framework for the automatic exploitation of the functional parallelism of a sequential program through the different SPEs. Their work is based on annotation of the source code of target application. A runtime library deals with generating threads, scheduling them on the SPEs, and transferring data to/from them. The authors in [30] present a realtime software platform for the Cell processor. It is based on the virtualization of the processing resources and a real-time resource scheduler which runs on the PPE. The compiler described in [20] implements techniques for optimizing the execution of scalar code in SIMD units, subword optimization and other techniques. Authors in [19] describe several compiler techniques that aim at automatically generating high-quality code over a wide range of heterogeneous parallelism available on the CELL processor.

At task level, the authors in [33] propose a programming model based on micro-tasks communicating through message passing interface. The micro-task represents a unit of computation that causes communication at its beginning and end. They tackle the mapping and scheduling problem by a suboptimal heuristic solver. The work in [34] describes a multicore streaming layer whose main goal is to abstract away the architecture-specific details that complicate the scheduling of computation and communication activities in a stream program. They propose both dynamic and static scheduling facilities, but without any optimality guarantee.

The literature on optimization of other multicore architectures uses heuristic approaches for mapping and scheduling task graphs onto the target platforms. In [16] a re-timing heuristic is used to implement pipelined scheduling, that optimizes the initiation interval, the number of pipeline stages and memory requirements of a particular design alternative. Pipelined execution of a set of periodic activities is also addressed

in [17], for the case where tasks have deadlines larger than their periods. Palazzari et al. [31] focus on scheduling to sustain the throughput of a given periodic task set and to serve aperiodic requests associated with hard real-time constraints. Mapping of tasks to processors, pipelining of system specification and scheduling of each pipeline stage have been addressed in [18], aiming at satisfying throughput constraints at minimal hardware cost. A comparative study of well-known heuristic search techniques (genetic algorithms, simulated annealing and tabu search) is reported in [21]. Eles et al. [22] compare the use of simulated annealing and tabu search for partitioning a graph into hardware and software parts while trying to reduce communication and synchronization between parts. More scalable versions of these algorithms for large real-time systems are introduced in [23]. Many heuristic scheduling algorithms are variants and extensions of list scheduling [24], a scheduling algorithm coming from the real time literature.

Heuristic approaches provide no guarantees about the quality of the final solution. On the other hand, complete approaches which compute the optimum solution (possibly, with a high computational cost), can be attractive for statically scheduled systems, where the solution is computed once and applied throughout the entire lifetime of the system.

Our previous work [3], [4] was aimed at optimally solving task graphs allocation and scheduling on a different multicore platform (called MPARM and based on ARM processors) using a Logic Based Benders Decomposition approach. The allocation part is solved through Integer Programming and the scheduling problem via Constraint Programming. We have applied and extended this approach for the CELL BE platform in [5]. We will summarize this paper in section 4. In this paper we propose a pure Constraint Programming approach for this problem.

CP has been previously used to solve similar, yet simplified, problems. The work in [25] is based on Constraint Logic Programming to represent system synthesis problem, and leverages a set of finite domain variables and constraints imposed on these variables. Optimal solutions can be obtained for small problems, while large problems require the use of heuristic algorithms. The proposed framework is able to create pipelined implementations in order to increase the design throughput. In [26] the embedded system is represented by a set of finite domain constraints defining different requirements on process timing, system resources and interprocess communication. The assignment of processes to processors and interprocess communications to buses as well as their scheduling are then defined as an optimization problem tackled by means of constraint solving techniques.

4 How CP

For the problem of allocating and scheduling task graphs onto the CELL BE platform we have implemented two approaches. One is based on a recursive application of Logic Based Benders Decomposition [8] and is extensively described in [5]. We recall here the main structure of the solution technique, while we refer to [5] for modeling details and extensive comparison with a traditional (two-stage) decomposition approach.

The second model we propose is the core of this paper and is a pure CP model where both allocation and scheduling are solved using a single monolithic model.

We describe in detail this second approach and propose an experimental evaluation in section 5 along with a comparison with the decomposition approach.

4.1 Decomposition Based Approach

The problem at hand can be solved using a Logic Based Benders decomposition approach similarly to [3], [4], [7], [6], [9], [10], and [11], where the allocation is modelled and solved in the master problem (usually using Integer Programming) while the scheduling problem is tackled as a subproblem (possibly via Constraint Programming). This approach does not scale well and in [5] we have shown that the reason is the poor balancing between the allocation and the scheduling components, as the first is much more complicated.

Therefore, we have experimented a multi-stage decomposition, which is actually a recursive application of standard Logic based Benders' Decomposition (LBD), that aims at obtaining balanced and lighter components. The allocation part should be decomposed again in two subproblems, each part being easily solvable.

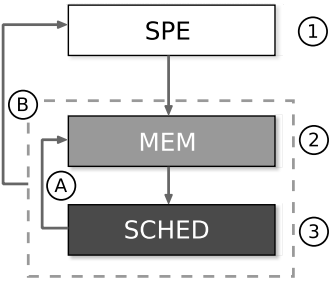

Fig. 4. Solver architecture: two level Logic based Benders' Decomposition

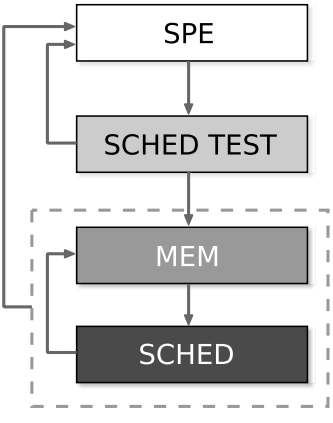

Fig. 5. Solver architecture with schedulability test

In Figure 4 at level one the SPE assignment problem (SPE stage) that computes task to processor assignment acts as the master problem, while memory device assignment and scheduling as a whole are the subproblem. At level two (the dashed box in Figure 4) the memory assignment (MEM stage) is the master and the scheduling (SCHED stage) is the correspondent subproblem. The first step of the solution process is the computation of a task-to-SPE assignment; then, based on that assignment, allocation choices for all memory requirements are taken. Finally, a scheduling problem with fixed resource assignments and fixed durations is solved. When the SCHED problem is solved (no matter if a solution has been found), one or more cuts (labeled A) are generated to forbid (at least) the current memory device allocation and the process is restarted from the MEM stage; in addition, if the scheduling problem is feasible, an upper bound on the

value of the next solution is also posted. When the MEM & SCHED subproblem ends (either successfully or not), more cuts (labeled B) are generated to forbid the current task-to-SPE assignment. When the SPE stage becomes infeasible the process is over, and converges to the optimal solution for the problem overall.

We found that quite often SPE allocation choices are by themselves very relevant: in particular, a bad SPE assignment is sometimes sufficient to make the scheduling problem unfeasible. Thus, after the task to processor allocation, we can first check whether the SPE allocation is schedulable, as depicted in Figure 5 (SCHED TEST). In practice, if the given allocation with minimal task durations is already infeasible for the scheduling component, then it is useless to complete it with the memory assignment that cannot lead to any feasible solution overall.

4.2 Pure CP Model

In alternative to the decomposition approach, we have implemented a pure CP model that is solved using the commercial tool ILOG Scheduler/Solver 6.3.

Let n be the number of tasks, m the number of arcs and p the number of processing elements.

The possible allocation choices are modeled by means of the following variables:

$$\begin{array}{ll} TPE_i \in \{0, \ldots, ...p-1\} & \forall i = 0, \ldots, n-1 \\ M_i \in \{0, 1\} & \forall i = 0, \ldots, n-1 \\ APE_r \in \{-1, \ldots, ...p-1\} & \forall r = 0, \ldots, m-1 \end{array}$$

TPE_i is the processing element assigned to task t_i. Similarly, if $APE_r = j$ then the communication buffer related to arc a_r is on the local memory of the processing element j, while if $APE_r = -1$ the communication buffer is allocated on the remote memory. Finally, M_i is 1 if program data of task t_i are allocated locally to the same processor of task t_i.

Due to architectural restrictions, a communication buffer can be allocated either on the local memory of the source task, or that of the target task, or on the remote memory; therefore for the arc r connecting nodes representing tasks t_h and t_k:

$$APE_r = TPE_h \vee APE_r = TPE_k \vee APE_r = -1$$

From a scheduling standpoint, each task is modeled as a set of non preemptive activities a, each with a start variable $start(a)$ and an end variable $end(a)$. In particular, a task t_i is split into an activity modeling its execution phase ex_i, and a set of activities modeling each one the reading and writing of a communication buffer, i.e., wr_r for each outgoing arc r and rd_r for each incoming arc r:

$$\begin{array}{ll} ex_i(ED_i) & \forall t_i \\ wr_r(WD_r) & \forall a_r = (t_i, t_k) \\ rd_r(RD_r) & \forall a_r = (t_h, t_i) \end{array}$$

The duration of each activity is defined by the proper variable and is reported between round brackets after its name. It depends on the related memory allocation choices; hence we define a variable for each execution and communication task:

$$\begin{array}{ll} ED_i \in \{0,\ldots,...eoh\} & \forall i = 0,\ldots,n-1 \\ WD_r \in \{0,\ldots,...eoh\} & \forall r = 0,\ldots,m-1 \\ RD_r \in \{0,\ldots,...eoh\} & \forall r = 0,\ldots,m-1 \end{array}$$

ED_i is the duration of the communication phase of task t_i, WD_r and RD_r respectively are the time needed to write and read buffer r. Their range is the whole temporal horizon (eoh is the end of horizon).

As stated in section 2.2, durations are linked to the allocation choices; the duration of an execution phase in case of remote allocation of program data ($dmax^{ex}$) is greater than in case of local allocation. Writing (and reading) operations have their minimum possible value ($dmin^{wr}, dmin^{rd}$) if the communication queue is on the local memory of the producer task (resp. consumer), a higher value ($dmed^{wr}, dmed^{rd}$) if it is allocated on the local memory of the consumer (resp. producer) task, an even higher value ($dmax^{wr}, dmax^{rd}$) in case of remote allocation of communication queue in DRAM. All those properties are enforced by means of the following constraints:

$$\begin{array}{ll} \forall i = 0,\ldots,n-1 & ED_i = dmin_i^{ex} + \\ & (dmax_i^{ex} - dmin_i^{ex})(1 - M_i) \\ \forall r = 0,\ldots,m-1, a_r = (t_h,t_k) & WD_i = dmin_r^{wr} + \\ & (dmax_r^{wr} - dmin_r^{wr})(APE_r = -1) + \\ & (dmed_r^{wr} - dmin_r^{wr})(APE_r = TPE_k) \\ \forall r = 0,\ldots,m-1, a_r = (t_h,t_k) & RD_i = dmin_r^{rd} + \\ & (dmax_r^{rd} - dmin_r^{rd})(APE_r = -1) + \\ & (dmed_r^{rd} - dmin_r^{rd})(APE_r = TPE_h) \end{array}$$

All reading operations are performed immediately before the execution, and all writing operations start immediately after. Let $r_0,\ldots,r_{h-1}$ be the indices of the ingoing arcs of task t_i and $r_h,\ldots,r_{k-1}$ those of the outgoing arcs; then:

$$\begin{array}{ll} end(rd_{r_j}) = start(rd_{r_{j+1}}) & \forall j = 0, h-2 \\ end(rd_{r_{h-1}}) = start(ex_i) & \\ end(ex_i) = start(wr_{r_h}) & \\ end(rd_{r_j}) = start(rd_{r_{j+1}}) & \forall j = h, k-2 \end{array}$$

All resource constraints are triggered when the TPE allocation variables are assigned; in particular if $TPE_i = j$, all reading, writing and execution activities related to task t_i require processing element j. The resource capacity constraint is enforced by a timetable constraint and a precedence graph constraint available in ILOG Scheduler 6.3 [13].

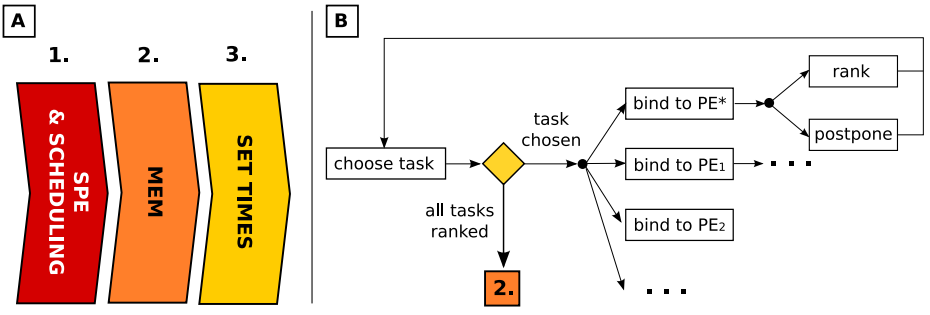

Fig. 6. A: Structure of the dynamic search strategy; B: Operation schema for phase 1

Search Strategy

The model is solved by means of a dynamic search strategy where resource allocation and scheduling decisions are interleaved.

We chose this approach since most resource constraints are not able to effectively prune start and end variables as long as the time windows are large and no task (or just a few of them) has an obligatory region: in particular it is difficult, before scheduling decisions are taken, to effectively exploit the presence of precedence relations and makespan bounds. In our approach, tasks are scheduled immediately after they are assigned to a processing element: this results in immediate updates of the time windows for all tasks linked by precedence relations.

The main drawback with this method is that an early bad choice is likely to lead to thrashing, due to the size of the search space resulting from the mixture of allocation and scheduling decisions; a pure two phases allocation and scheduling approach, like the decomposition based one presented in the previous section, would be able to recover faster from such a situation.

Intuitively, the presence of many precedence constraints strongly shrinks the set of good allocation choices and is likely to guide the allocation toward promising choices, whereas if the graph mostly contains independent or loosely related tasks a two stages approach is probably to be preferred.

A considerable difficulty is our specific case is set by the need to assign each task and arc both to a processing element and to a storage device: this is makes the number of possible allocations too big to completely define the allocation of each task right before it is scheduled. Therefore we chose to postpone the memory allocation stage after the main scheduling decisions are taken, as depicted in Figure 6A.

Since task durations directly depend on memory assignment, scheduling decisions taken in phase 1 of Figure 6 had to be relaxed to enable the construction of a *fluid* schedule with variable durations. In practice we adopted a Precedence Constraint Posting approach [1,2], by just adding precedence relations to fix the order of tasks at the time they are assigned to SPEs: they will be given a start time only once the memory devices are assigned. Note this time setting step is done in polynomial time. Figure 7A shows an example of fluid schedule where tasks have variable durations and precedence relations have been added to fix the order of the tasks on each SPE;

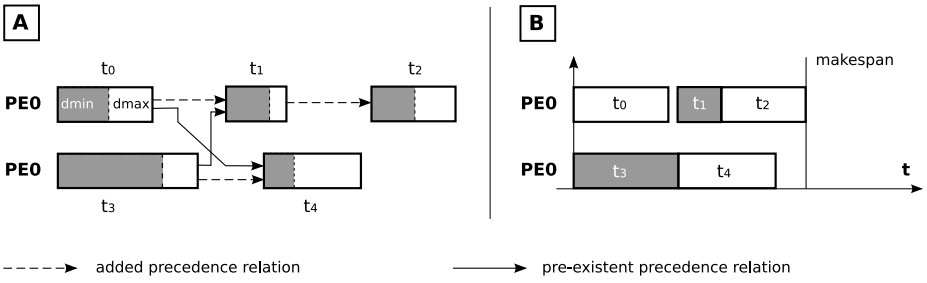

Fig. 7. A: A fluid schedule; B: A possible fixed schedule

Figure 7B show a corresponding schedule where all durations are decided (a grey box means the minimum duration is used, a white box means the opposite).

In deeper detail, the SPE allocation and scheduling phase operates according to the schema of Figure 6B: first, the task with minimum start time is selected – ties are broken looking a the (least) maximum end time and than at the task index. Second, the SPE where the task can be allocated at its minimum start time is identified (let it be SPE^*), then a choice point is open, with a branch for each SPE. Along each branch the task is bound to the corresponding resource and a *rank or postpone* decision is taken: we try to rank the task immediately after the last activity on the selected resource, otherwise the task is postponed and not considered ready until its minimum start time changes due to propagation (this is analogous to the standard schedule or postpone strategy in ILOG). The process is reiterated as long as there are unranked tasks.

In phase 2, memory requirements are allocated to storage devices, selecting at each step the variable with the smallest domain; in phase 3 a start time is assigned to each task. Finally, since the processing elements are symmetric resources the procedure embeds quite standard symmetry breaking techniques to prevent the generation of useless branches in the search tree.

5 Computational Efficiency

The decomposition based approach has been implemented using the state of the art solvers ILOG Cplex 10.1 and Scheduler/Solver 6.3, while the pure CP model has been implemented on Scheduler/Solver 6.3.

Since the main goal of the paper is to study and compare the performance of the two approaches it would be not realistic to assume the availability of such a large benchmark set that would allow us to sample a large variety of problem instances. Therefore we resorted to synthetic benckmarks as follows.

A first group of 90 instances is coming from the actual execution of multi tasking programs on a CELL BE architecture. These benckmarks have been created by synthesising code (matrix multiplication) tuning the computation vs. communication effort which is related to matrix size. For the instances in the first group the duration variability is very small or even null depending on memory allocation (i.e., $dmin^{ex}$ and $dmax^{ex}$ are very close or equal, and analogously durations of reading and writing activities are similar).

Table 1. Results on the set of instances where task durations are not strongly influenced by allocation decisions

		CP			TD		
Number of tasks	**Number of arcs**	**time (sec.)**	**SbB**	**> TL**	**time (sec.)**	**SbB**	**> TL**
15	9-13	0.01	0.01	0	0.31	0.31	0
15	14-26	0.02	0.02	0	0.62	0.62	0
25	30-55	0.10	0.11	0	369.66	369.66	2
25	56-65	0.05	0.05	0	530.96	530.96	2
30	47-71	1.25	0.82	2	620.13	620.13	11
30	73-82	0.12	0.09	0	834.45	834.45	8

A second group of instances has been generated by using the same task graph structure of the first group and by changing randomly the durations of communication activities depending on the allocation choices; we chose to generate 200 instances instead of 90 to increase the reliability of the evalutation. Compared to the previous ones, instances of this second group have a higher variability of minimal and maximal task durations.

The first set of instances is representative of high computational intensive applications in general, like many signal processing kernels. In this scenario the overall task duration is dominated by the data computation section, while the variability induced by different memory allocations is negligible. On the other hand, the second set is representative of more communication intensive applications. In this case, the overall task duration can be drastically affected by different memory allocations. Several video and image processing algorithms are good examples of applications which fit in this category. The Cell configuration we used for the tests has 6 available SPEs.

Results on the first set of instances, where task duration is not much influenced by memory allocation, are reported in table 5. Every row reports results on 15 instances. Each instance is characterized by the number of tasks and a variable number of arcs in the interval reported in the table. We recall that arcs in the task graph represent communications and should be modelled with two communication activities (writing and reading). For each solver the computation time is reported in seconds and is the average execution time on instances solved to optimality (in which case the two approaches yield the same solution quality). In the column SbB the time computation is restricted to instances solved by both methods; finally column > TL reports the number of timed out instances (out of 15). The time limit has been set to 1800 seconds.

As we can see the CP approach achieves significant speed ups with respect to the decomposition approach and the number of timed out instances is significantly smaller in this case. The produced schedules were validated on the same platform used for characterization of the instances.

On the other hand, results on the second set of instances where tasks have high duration variability due to allocation choices are reported in table 2. Every row reports results on 20 instances. Each instance is characterized by the number of tasks (variable in the range reported in table) and the number of arcs. The time is reported in seconds and is the average execution time on instances solved within the time limit; as in the previous table in the colmn SbB the time computation is restricted to instances solved

Table 2. Result on the set of instances where task durations are strongly influenced by allocation decisions

		CP			TD		
Number of tasks	**Number of arcs**	**time (sec.)**	**SbB**	**> TL**	**time (sec.)**	**SbB**	**> TL**
10-11	4-11	16.70	16.70	0	3.67	3.67	0
12-13	8-14	116.92	116.92	2	11.19	4.59	0
14-15	8-15	81.50	81.50	8	10.25	7.67	0
16-17	11-17	34.66	34.66	11	29.53	18.17	0
18-19	13-19	66.47	66.47	15	72.56	33.92	1
20-21	16-22	400.41	400.41	16	248.00	82.50	2
22-23	19-26	30.78	30.78	18	355.15	395.00	3
24-25	20-29	—	—	20	200.00	—	9
26-27	23-29	—	—	20	425.00	—	6
28-29	25-35	—	—	20	742.73	—	9

by both approaches. In the column > TL we report the number of timed out instances (out of 20). Also in this case the time limit has been set to 1800 seconds.

As we can see, the performances of the pure CP approach now start decreasing. For the difficult instances (last three rows), all 20 instances have achieved the time limit while the decomposition approach is still able to produce optimal results for half of the instances.

It appears that the CP solver, during the initial PE assignment and scheduling phase, has difficulties in computing good makespan bounds taking into account the impact of memory allocation choices. On the other hand those choices are anticipated, and thus better managed, by the decomposition based solver, at the price of a weakness in exploiting resource constraints to compute makespan bounds. Benders' cuts seem to be a quite robust device to partially overcome the limitations of the decomposition approach: perhaps they could be introduced as well in the CP solver to give to it the ability to handle memory allocation.

These results give a clear indication about the type of solver we have to use depending on the instance structure. If the allocation part is predominant since it greatly influences task durations, the decomposition approach should be used. On the contrary, if choosing resource assignments should respect resource capacity constrains but it does not influence significantly task durations, the pure CP approach greatly outperforms the (more complex) decomposition approach.

6 Conclusions

The work presented in this paper is part of a wider project aimed at developing a software development infrastructure, called Cellflow to help programmers in software implementation on the Cell Broadband Engine processor. Although an off-line development framework and an on-line runtime support are needed in Cellflow, the optimization engine is a fundamental component. We are designing an algorithm portfolio and a selection algorithm based on the instance structure.

Acknowledgement

The work described in this publication was supported by the PREDATOR Project funded by the European Community's 7th Framework Programme, Contract FP7-ICT-216008.

References

1. Policella, N., Cesta, A., Oddi, A., Smith, S.F.: From precedence constraint posting to partial order schedulesA CSP approach to Robust Scheduling. AI Communications 20(3), 163–180 (2007)
2. Laborie, P.: Complete MCS-Based Search: Application to Resource Constrained Project Scheduling. In: Proc. of IJCAI 2005, pp. 181–186 (2005)
3. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for MPSOCs via decomposition and no-good generation. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 107–121. Springer, Heidelberg (2005)
4. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation, Scheduling and Voltage Scaling on Energy Aware MPSoCs. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, pp. 44–58. Springer, Heidelberg (2006)
5. Benini, L., Lombardi, M., Mantovani, M., Milano, M., Ruggiero, M.: Multi-stage Benders Decomposition for Optimizing Multicore Architectures. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 36–50. Springer, Heidelberg (2008)
6. Bockmayr, A., Pisaruk, N.: Detecting infeasibility and generating cuts for MIP using CP. In: Int. Workshop Integration AI OR Techniques Constraint Programming Combin. Optim. Problems CP-AI-OR 2003, Montreal, Canada (2003)
7. Grossmann, I.E., Jain, V.: Algorithms for hybrid milp/cp models for a class of optimization problems. INFORMS Journal on Computing 13, 258–276 (2001)
8. Hooker, J.N., Ottosson, G.: Logic-based benders decomposition. Mathematical Programming 96, 33–60 (2003)
9. Hooker, J.N.: A hybrid method for planning and scheduling. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 305–316. Springer, Heidelberg (2004)
10. Hooker, J.N.: Planning and scheduling to minimize tardiness. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 314–327. Springer, Heidelberg (2005)
11. Sadykov, R., Wolsey, L.A.: Integer Programming and Constraint Programming in Solving a Multimachine Assignment Scheduling Problem with Deadlines and Release Dates. INFORMS Journal on Computing 18(2), 209–217 (2006)
12. Ibm CELL Broadband Engine software development kit, `http://www.alphaworks.ibm.com/tech/cellsw/download`
13. Laborie, P.: Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. Journal of Artificial Intelligence 143, 151–188 (2003)
14. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: Cellss: a programming model for the cell be architecture. In: SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, p. 86. ACM Press, New York (2006)
15. Chen, T., Raghavan, R., Dale, J., Iwata, E.: Cell broadband engine architecture and its first implementation. In: IBM White paper (2005)
16. Chatha, K.S., Vemuri, R.: Hardware-software partitioning and pipelined scheduling of transformative applications, vol. 10, pp. 193–208 (2002)
17. Fohler, G., Ramamritham, K.: Static scheduling of pipelined periodic tasks in distributed real-time systems. In: Procs. of the 9th EUROMICRO Workshop on Real-Time Systems - EUROMICRO-RTS 1997, Toledo, Spain, pp. 128–135. IEEE, Los Alamitos (1997)

18. Bakshi, S., Gajski, D.D.: A scheduling and pipelining algorithm for hardware/software systems. In: Proceedings of the 10th international symposium on System synthesis - ISSS 1997, Washington, DC, USA, pp. 113–118. IEEE Computer Society, Los Alamitos (1997)
19. Eichenberger, A., et al.: Optimizing compiler for the cell processor. In: PACT 2005: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, pp. 161–172. IEEE Computer Society, Los Alamitos (2005)
20. Eichenberger, A.E., et al.: Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. IBM Syst. J. 45(1), 59–84 (2006)
21. Axelsson, J.: Architecture synthesis and partitioning of real-time synthesis: a comparison of 3 heuristic search strategies. In: Procs. of the 5th Intern. Workshop on Hardware/Software Codesign (CODES/CASHE 1997), Braunschweig, Germany, pp. 161–166. IEEE, Los Alamitos (1997)
22. Eles, P., Peng, Z., Kuchcinski, K., Doboli, A.: System level hardware/software partitioning based on simulated annealing and tabu search. Design Automation for Embedded Systems 2, 5–32 (1997)
23. Kodase, S., Wang, S., Gu, Z., Shin, K.: Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers. In: Procs. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), Toronto, Canada, pp. 181–188. IEEE, Los Alamitos (2003)
24. Eles, P., Peng, Z., Kuchcinski, K., Doboli, A., Pop, P.: Scheduling of conditional process graphs for the synthesis of embedded systems, Paris, France, pp. 132–139 (1998)
25. Kuchcinski, K., Szymanek, R.: A constructive algorithm for memory-aware task assignment and scheduling. In: Procs of the Ninth International Symposium on Hardware/Software Codesign - CODES 2001, Copenhagen, Denmark, pp. 147–152. ACM Press, New York (2001)
26. Kuchcinski, K.: Embedded system synthesis by timing constraint solving. IEEE Transactions on CAD 13, 537–551 (1994)
27. Flachs, B., et al.: A streaming processing unit for a cell processor. In: IEEE International Solid-State Circuits Conference, 2005 (ISSCC 2005). Digest of Technical Papers, pp. 134–135 (2005)
28. Hofstee, H.: Cell broadband engine architecture from 20,000 feet. In: IBM White paper (2005)
29. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: Built for speed. IEEE Micro. 26(3), 10–23 (2006)
30. Maeda, S., Asano, S., Shimada, T., Awazu, K., Tago, H.: A real-time software platform for the cell processor. IEEE Micro. 25(5), 20–29 (2005)
31. Palazzari, P., Baldini, L., Coli, M.: Synthesis of pipelined systems for the contemporaneous execution of periodic and aperiodic tasks with hard real-time constraints. In: 18th International Parallel and Distributed Processing Symposium - IPDPS 2004, pp. 121–128 (2004)
32. Pham, D., et al.: The design and implementation of a first-generation cell processor. In: IEEE International Solid-State Circuits Conference ISSCC 2005, vol. 1, pp. 184–592 (2005)
33. Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: MPI microtask for programming the Cell Broadband Engine processor. IBM System Journal 45(1) (2006)
34. Zhang, D., Li, Q.J., Rabbah, R., Amarasinghe, S.: A Lightweight Streaming Layer forMulticore Execution. In: Proceedings of Workshop on Design, Architecture and Simulation of Chip Multi-Processors, dasCMP 2007 (2007)