Cellflow: a Parallel Application Development Environment with Run-Time Support for the Cell BE Processor

Martino Ruggiero, Michele Lombardi, Michela Milano and Luca Benini
University of Bologna, DEIS,
Viale Risorgimento, 2 - Bologna 40136 (Italy)

ABSTRACT

The Cell BE processor provides both scalable computation power and flexibility, and it is already being adopted for many computational intensive applications. 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 propose 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.

Experimental results show that in Cellflow we reduced to minimum the abstraction gap between the optimization and development phases.

1. INTRODUCTION

Cell 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 engine. Cell has already demonstrated impressive performance ratings in computationally intensive applications and kernels mainly thanks to its innovative architectural features. The heterogeneity of its computational capability, the limited, explicitly-managed on-chip memory and the multiple options for exploiting hardware parallelism, make efficient application design and implementation a major challenge. Efficiently programming requires 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 synchronizations, etc.

Moving from these considerations, the novelty of this work is the creation of a framework, called Cellflow, that can help programmers in handling these complex and critical activities and decisions. Our goal is to enable developers to quickly build multi-task applications using an explicit parallel programming model. Our key object is to give developers access to the power of Cell multi-core architecture, but at a high level. 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.

Our toolkit is made of an off-line and an on-line components. The off-line facility is a design-time software optimization infrastructure for the deployment of multi-task applications. It is made up of:
- a generic customizable application template that helps software developers to easily and quickly build their application skeleton starting from a high level task and data flow graph,
- an allocation and scheduling support, featuring both a heuristic suboptimal algorithm and an exact solver that finds the optimal mapping and scheduling on the hardware architecture.

The software framework is targeted towards statically-configured applications, where optimal allocation and scheduling settings are precomputed at design time. As far as the mapping and scheduling support is concerned, we have implemented two approaches: the first is based on traditional list scheduling and round robin allocation, which are simple and scalable heuristics that do not provide any optimality guarantee. The second is an exact optimization algorithm providing optimal solutions. It has been extensively described in [3] and it is based on a multi-stage decomposition. An extensive set of experimental results confirm that the multi-stage decomposition pays off in terms of efficiency w.r.t. a traditional approach.

The on-line runtime support is composed by software libraries and APIs which manage coordination issues, such as task allocation and scheduling, as well as task-level issues, like inter-task communication and synchronization. In this phase we tackle also the problem of the limited memory size of the SPEs, optimizing their utilization through overlaying.

With Cellflow, Cell programming becomes simpler, but at the same time it achieves high efficiency thanks to the run-time support (which is tuned to the SPE harware) and to off-line optimal allocation and scheduling.

2. RELATED WORK

The Cell architecture includes multiple, heterogeneous processor elements (PPE and SPEs) and Single-Instruction-Multiple-Data (SIMD) units on all SPEs. This kind of platform supports a wide range of heterogeneous parallelism levels. To our knowledge, prior work is mainly focused on trying to exploit fine grain parallelism of Cell, such as at instruction and function level, while our work is one of the few approaches at task level.

Authors in [14] provide a software development platform which allows to use standard C++ programming to create parallel applications, or extend existing applications to run on Cell. Some parallel programming models have been implemented and ported on the Cell processor [13, 16]. The authors in [16] have ported Streamit and its run-time environment on Cell architecture. Streamit is based on a dataflow programming language, but it needs its own compiler, while in our case we are fully compatible with the standard C-based development flow.

The authors in [17] propose a programming model based on micro-tasks communicating through massage 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 [18] describes a multicore streaming layer whose main goal is to abstract away the architecture-specific details that complicate the scheduling of computation and communication in a stream program. They propose both dynamic and static scheduling facilities, but without any requirements of optimality.

3. 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 programming enviroment. Cell is a non-homogeneous multi-core processor [15] 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) [12]. 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 compute-intensive applications. The SPE is a compute-intensive coprocessor designed to accelerate media and streaming workloads. 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.

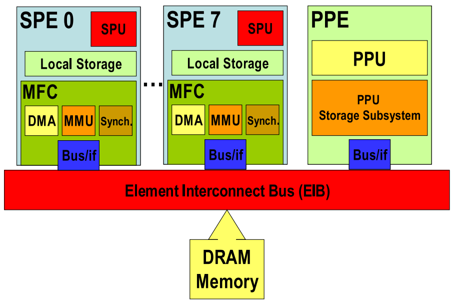


Figure 1: Cell Broadband Engine Hardware Architecture.

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

4. OFF-LINE DEVELOPMENT INFRASTRUCTURE

In this section, we describe the computational model supported by our environment, and the off-line (development time) support for the allocation and scheduling of parallel tasks on SPEs.

4.1 Application and task computational model

Our application model is a task graph with precedence constraints. Nodes of the graph represent concurrent tasks while the arcs indicate mutual dependencies due, for example, to communication and/or synchronization. Tasks communicate through queues and each task can handle several input/output queues.

Input Reading
Task Execution
Output Writing

Figure 2: Three phases behavior of Tasks.

Task execution is modeled and structured in three phases, as indicated in Figure 2: 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 2 kinds of associated memory requirements:
Program Data: storage locations are required for computation data and for processor instructions;
Communication queues: the task needs queues to transmit and receive messages to/from other tasks, eventually mapped on different SPEs.
Both these memory requirements can be allocated on the local storage of each SPE or reside in the shared memory.

4.2 Customizable Application Template

We set up a generic customizable application template allowing software developers to easily and quickly build their parallel applications starting from a high-level task and data flow graph specification compliant to the above mentioned model. Programmers can at first think about their applications in terms of task dependencies and quickly draw the task graphs, and then use our tools and libraries to translate the abstract representation into C code. This way, they can devote most of their effort to the functionality of tasks rather than the implementation of their communication, synchronization and scheduling mechanisms.

User can configure the Customizable Application Template via XML file, which will be automatically translated into C-code.

We implemented also an Eclipse plug-in graphical interface to make the configuration of the Customizable Application Template easier and less error-prone. The user can compose his/her application task graph simply dragging and dropping nodes (i.e. task) and arrows (i.e. precedence constraints). Then, our plugin will produce the XML file corresponding to the Customizable Application Template configuration.

After this configuration step, the programmer can only focus on writing the algorithms that will run on the SPEs using standard C code. Our infrastructure will automatically manage communication and synchronization between threads, exploiting at best Cell architecture features. The kernel of our customizable application template is made of a part running on PPE, that reads the configuration files and sets up all the system structures, and a part running on SPEs, that supports the run-time execution and communication. Details on how the initialization phase works and how PPE and SPE interact will be explained in section 5.1.

4.3 Allocation and Scheduling support

The problem of efficiently allocating and scheduling multi-task applications on a multi-processor in a distributed system is very challenging.

We have proposed two approaches for this task: a heuristic algorithm that provides fast but sub-optimal mapping and scheduling and an exact solver employing leading edge optimization technologies and an accurate application and platform model.

4.3.1 Heuristic approach

The suboptimal algorithm is based on a list scheduling heuristic [11], which emphasizes speed and scalability at the price of optimality loss. List scheduling keeps a list of the ready tasks (the ones whose all producers have already finished); that list is then ordered according to a priority function, and the highest-priority ready task is scheduled next. To assign priorities to tasks, ASAP (As Soon As Possible) and ALAP (As Late As Possible) start times are determined for each task, according to application task dependencies, and task mobility is calculated as the difference ALAP-ASAP. The highest mobility a task has, the highest priority it will obtain. Once the scheduling is found, the tasks are mapped on the SPEs according to a Round-Robin algorithm: proceeding in priority order, each task is mapped on a different SPE. In this way it is possible to achieve a good load balancing between all the SPEs. List scheduling and round-robin (R-R) allocation are simple and scalable heuristics, but they do not provide any optimality guarantee.

4.3.2 Exact approach

The second approach we use is aimed at producing an optimal solution for the mapping and allocation problems that minimizes the execution time of the overall application. The problem we have to solve is a scheduling problem with alternative resources and allocation dependent durations. A good way of facing these kind of problems is via Benders Decomposition, and its Logic-based extension [6]. Previous papers have shown the effectiveness of the method for similar problems. Hooker in [7] and [8] has shown how to deal with several objective functions in problems where tasks allocated on different machines are not linked by precedence constraints. Similar problems have been faced by Jain and Grossmann [5], Bockmayr and Pisaruk [4] and Sadykov and Wolsey [9], the latter comparing this approach and showing its superiority w.r.t. Integer Programming branch and cut and column generation. Many of these approaches consider multiple *independent* subproblems: that is, once the master problem is solved, then many decoupled subproblems result which can be solved in an independent fashion.

The allocation is in general effectively solved through Integer Linear Programming, while scheduling is better faced via Constraint Programming. In our case, the scheduling problem cannot be divided into disjoint single machine problems since we have precedence constraints linking tasks allocated on different processors. We have implemented such an approach, called two-stage decomposition into the Cellflow infrastructure, similarly to [1], [2], and experimentally experienced a number of drawbacks. The main problem is that for the problem at hand a two stage decomposition produces two unbalanced components. The allocation part is extremely difficult to solve while the scheduling part is indeed easier. This approach scales poorly.

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.

In Figure 3 at level one the SPE assignment problem (SPE stage) 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 3) 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. Deciding the allocation of tasks and memory requirements univocally defines task

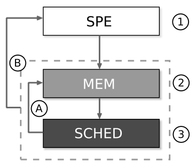


Figure 3: Solver architecture: two level Logic based Benders' Decomposition

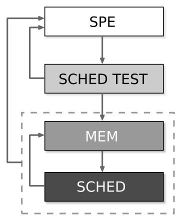


Figure 4: Solver architecture with schedulability test

durations. 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 converging 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 infeasible. Thus, after the task to processor allocation, we can perform a first schedulability test as depicted in Figure 4. In practice, if the given allocation with minimal 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.

A detailed description of the models employed at various levels can be found in [3].

5. ON-LINE RUNTIME SUPPORT

The runtime support takes care of the task scheduling and data handling between the different cores.

5.1 SPE task allocator and scheduler

Once the target application has been implemented using our generic customizable template, tasks, program data and communication queues are allocated to the proper hardware resources (SPEs or memory resources). This is done through the init task of our template which allocates and launches all the activities at booting time. More specifically, during boot the PPE creates a global configuration table that contains information about queue buffers and where to allocate local data. The table is arranged so that each table entry contains information related to a task. To populate that table, the PPE reads the configuration files (that contain allocation information) and interacts with SPEs to know the physical addresses of all structures of queues.

This interaction between PPE and SPEs uses a specific mailbox-based protocol, that supports:
- allocating local SPE data;
- allocating buffers;
- initializing semaphores;
- starting the execution (once the table is completed).

The PPE gathers information from configuration files and SPEs, and builds the global application table. When a task is scheduled, its code overlay is loaded and the task's entry from the global table is received. This means a very dynamic and memory-efficient (for both code and data) local storage management to cope with its limited size. In order to reproduce the desired scheduling behavior, we implemented a scheduling support middleware. Using this facility, programmers only have to specify the desired scheduling for every SPE, which is handled accordingly by our middleware in a transparent way. To overcome the capacity limitations of local storage, we support SPE overlay: every time a new task has to be scheduled, it is loaded into local storage by our middleware through overlay. In an overlay structure the local storage is divided into a root segment, which resides always in storage, and one or more overlay regions, where overlay segments are loaded when needed. In our framework, a scheduler is implemented on each SPE and its code is stored in the root segment of the local storage. Handling the scheduling on the SPE itself avoids additive overhead due to communication and synchronization with PPE. Our scheduling policy is non-preemptive, since the context of an SPE task is too large (it includes SPE registers, LS image, and outstanding DMA commands residing in the DMA queue) to achieve a quick context switch.

5.2 SPE Communication and Synchronization support

Software support for efficient messaging is also provided by our set of high-level APIs. The communication and synchronization library abstracts low level architectural details to the programmer, such as memory maps or explicit management of hardware semaphores or interrupt signaling. The infrastructure for the communication between a producer/consumer pair is composed by a data queue, two counters and a series of semaphores. The data queue is composed by several data slots. The data queue can be allocated either in shared memory or in local memory of SPE. A couple of semaphores associated to each slot by means of synchronization between producer/consumer pairs is implemented. Semaphores and counters are distributed and allocated in local storage to SPEs. When a producer task generates a message, it locally checks the private counter which contains the identifier of the free slot in the queue and starts to poll the slot's semaphore. When producer acquires the semaphore, it starts writing the message. If the data queue is allocated remotely (either in shared memory or in local memory to consumer) a DMA transfer is issued. When the message is ready, the producer signals this by releasing consumer's semaphore. If producer and consumer reside on different SPEs, this is the only bus access for the entire synchronization process. We set up a communication and synchronization library abstracting away low level architectural details to programmers, such as memory maps or explicit management of semaphores, DMA transfers and shared memory.

As previously mentioned, all the information about queues (i.e. structure physical addresses, ids, etc.) are stored in the task table which is filled at boot time: this brings to more efficient both communication and syncronization since the hand-shaking address negotiations are done only once and not every time a task is scheduled.

6. EXPERIMENTAL RESULTS

6.1 Case study

In this section we show an example of how to use Cellflow to build a parallel application. The selected case study is an instance of a software radio application. A software radio receives its input from a data source (the digitized antenna output), while its output is connected to a digital audio output device. As Figure 5 shows, the main dataflow is a pipeline with a band-pass filter for the desired

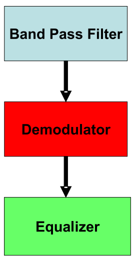
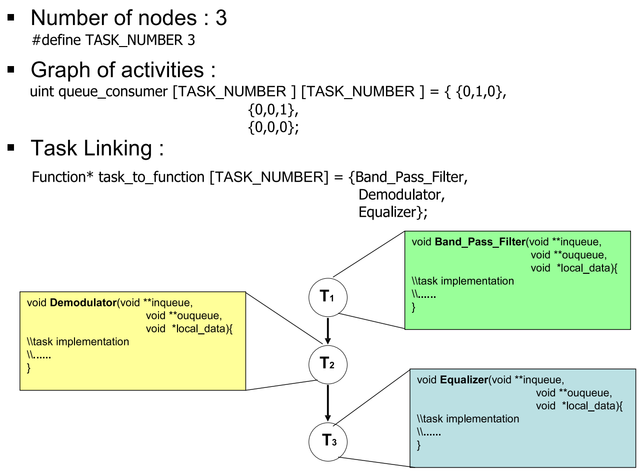


Figure 5: Data flow graph for a software FM radio

frequency, a demodulator, and an equalizer. The most intuitive and quick way to translate this data-flow into code using Cellflow is to map every node in the chart to a task. From the developer prespective, he/she will just implement the kernel code of these functional nodes and configure the application template to build the overall task graph (i.e. to specify to the run-time environment both communication and synchronization constraints between tasks). Figure 6 gives a simple view of the flow-graph and of the final implementation of the target software radio application.



- Number of nodes : 3

```
#define TASK_NUMBER 3
```

- Graph of activities :

```
uint queue_consumer [TASK_NUMBER ] [TASK_NUMBER ] = { {0,1,0},
                                                      {0,0,1},
                                                      {0,0,0};
```

- Task Linking :

```
Function* task_to_function [TASK_NUMBER] = {Band_Pass_Filter,
                                            Demodulator,
                                            Equalizer};
```

```
void Band_Pass_Filter(void **inqueue,
                      void **ouqueue,
                      void  *local_data){
\\task implementation
\\......
}
```

```
void Demodulator(void **inqueue,
                 void **ouqueue,
                 void  *local_data){
\\task implementation
\\......
}
```

```
void Equalizer(void **inqueue,
               void **ouqueue,
               void  *local_data){
\\task implementation
\\......
}
```

Figure 6: Simple dataflow graph of a software FM radio versus C code.

The above described pipeline implementation of the software radio application is the simplest way to map the data-flow graph into code. In order to increase the parallelism, the same benchmarch can be implemented splitting tasks in several sub-tasks, making the data-flow graph parallelism more explicit. More in detail, the equalizer task can be viewed as a more complex sub-graph composed by different filters: it is made up of a split-join, where each child adjusts the gain over a particular frequency range, followed by a filter that adds together the outputs of each of the bands. As Figure 7 shows, the equalizer is composed by a series of band-pass filters running in parallel, with their outputs added together. The band-pass filter can be viewed in turn as the subtraction between two low-pass filters which work at different frequency, with the overall result feeded to an amplifier. In the overall implementation through Cellflow (see Figure 8(a)), the translation of this more complex data-flow graph will only reflect a different configuration of the application template (i.e. with more tasks and a different communication and synchronization tree) and the implementation of more fine-grained kernel tasks (i.e. for the implemetation of the low-pass filter, the subtracer block and the amplifier).

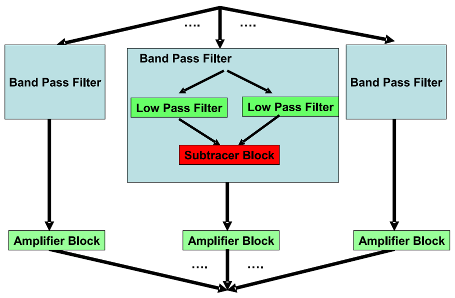


Figure 7: Flow graph for an equalizer.

6.2 Performance Analysis

In this section we analyze the speedup achieved by Cellflow on of three real-life applications, namely Mat-mult, FFT and Software Radio.

Mat-mult is a block matrix multiplication: each task executes a matrix multiplication between an input matrix and a private operand matrix, and then feeds its output to the following task. The platform receives a continuous flow of input matrices and produces a continuous flow of output matrices. This benchmark is representative of a wider class of applications for embedded systems with high data parallelism, like image and sound filters. The FFT benchmark is an implementation of the Fast Fourier Transform. Conceptually it is a single pipeline, but the main path is duplicated into a split-join to expose parallelism (see Figure 8(b))

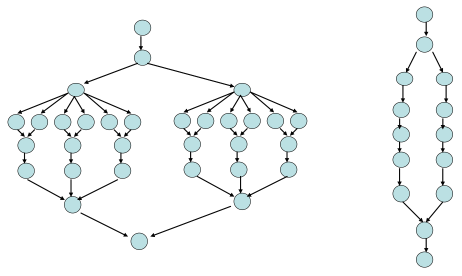


Figure 8: (a)Complex data-flow graph for the Software Radio. (b)FFT-benchmark flow graph.

The Software Radio implementation has been described in section 6.1.

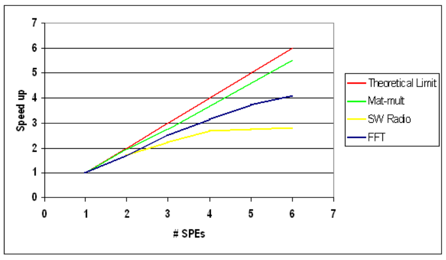


Figure 9: Benchmark results. Speedup is normalized against the execution with 1 SPE.

We carried on our analysis on a Sony PlayStation 3, which represents an inexpensive solution to work with Cell processor. The performance results for the three examples are shown in Figure 9. The figure presents the performance results obtained when running an increasing number of SPEs scaled to the case when 1 SPE is used. The Mat-mult benchmark scales almost perfectly w.r.t. the theoretical speed-up limit, thus proving the efficiency of our run-time environment and its almost negligible overhead. Also in the case of FFT an increasing number of SPEs brings to perceptible speed-ups. The software radio benchmark instead shows good speedup until only three SPEs: there is a path in the graph which duration bounds the speed up. More performance improvement can be reached in this case using software pipeline optimization.

6.3 Validation of optimizer solutions

To analyze the quality of our optimizer allocator, we performed experiments on a large set of synthetically generated task graphs. A task-graph generator has been implemented, so that it is possible to obtain a large number of pseudo-random test cases. To explore applications with different characteristics, the generator can be configured to produce task graphs with specific features, such as:
- Number of tasks;
- Average number of communication arcs between tasks;
- Average queue buffer size;
- Buffer and program data location;
- Average task execution time.

For test purposes, we produced three sets of task graphs: one with 15-task instances, one with 25-task instances and one with 30 tasks per instance. The test instances have then been processed by both our optimal and heuristic (list scheduling algorithm and Round-Robin allocator) solvers.

At this point, we had to profile the behaviour of applications (we were mainly interested in task execution times). Application profiling can be easily done running the applications on IBM Full-System Simulator, or using the profiling tool that comes with the Cell SDK, but the slowness of the former and the inaccuracy of the latter would prevent to run computationally intensive and precise tests. The best choice was to run the code on real hardware. Thus, all our experimental tests have been conducted using all the available SPEs (i.e. six for PlayStation 3): this is the worst case in terms of synchronization, communication and bus usage, as well as complexity of scheduling and allocation problem.

We compared for each instance the heuristic allocation and scheduling with the optimal ones. Figure 10 shows the percentage difference (normalized on 100 instances) of heuristic solutions with respect to the optimal ones. For the 37% of the instances, the heuristic and the exact solutions matched, for the remaining instances the heuristic optimizer produced sub-optimal results, with up to a 35% optimality gap.

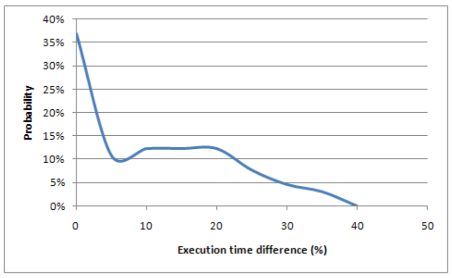


Figure 10: Histogram of the optimality gap on 100 instances.

Figure 11 represents the average performance (application execution time) for each set of tests (15, 25 and 30 tasks). This proves that the error does not grow too rapidly with the number of tasks,

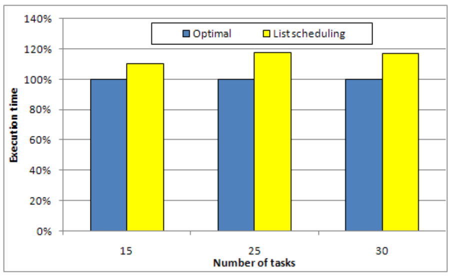


Figure 11: Comparison of average application execution times.

but remains around 15%. These experiments confirm that the optimal solver achieves significant better results, but also that the list scheduler with round-robin allocator provides resonable solutions for critical or very large instances.

6.4 Detailed comparison between optimal and heuristic approaches

In this section we perform a detailed comparison, on a meaningful case study, of the solution generated by the complete solver with that provided by the heuristic one. Figure12 shows the task graph of the selected instance. It is a complex graph, composed by 25 tasks, connected through 63 communication edges. The Gantt charts depicted in Figure13 shows a pictural overview of the solutions found by the optimal (top) and the heuristic (bottom) solvers

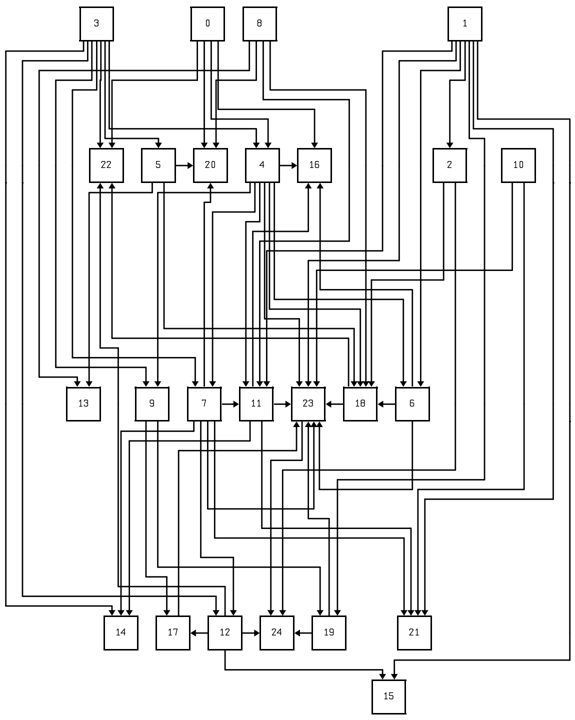


Figure 12: Task graph of the case study bench.

The heuristic solver finds a significantly slower solution than the complete solver, with a 20% optimality gap. There are two main reasons for this results, originating from sub-optimal choices in both scheduling and allocation:
- the list scheduling algorithm gives low priority to task 10, therefore it can not start very soon as done in the optimal solution. This

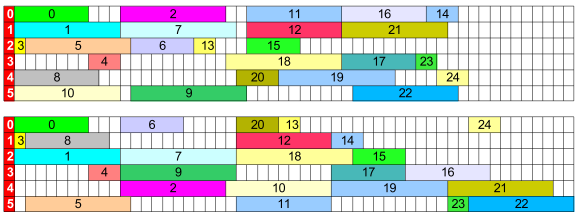


Figure 13: Comparison between optimal (top) and heuristic (bottom) schedule.

forces 24 to end later, since tasks 24, 23, 19 and 10 are dependant from each other.
- the Round-Robin allocation algorithm tries to load balance the number of tasks per SPE, so it allocates task 16, 21 and 22 on SPEs 3, 4 and 5, without exploiting the opportunity to use SPEs with lower utilization.

7. CONCLUSIONS

We propose a complete framework, called Cellflow, to help programmers in software implementation on the Cell Broadband Engine processor. Cellflow is composed by an off-line development framework and an on-line runtime support, and experimental results demonstrate the efficiency and viability of our solution.

8. 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.

9. REFERENCES

[1] Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for MPSOCs via decomposition and no-good generation. In: Proc. of the Int.l Conference in Principles and Practice of Constraint Programming (CP 2005).

[2] Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation, Scheduling and Voltage Scaling on Energy Aware MPSoCs. In: Proc. of the Int.l Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming (CPAIOR 2006).

[3] L.Benini, M. Lombardi, M. Mantovani, M. Milano and M. Ruggiero Multi-stage Benders Decomposition for Optimizing Multicore Architectures. In: Proc. of the Int.l Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming (CPAIOR 2008), to appear.

[4] Bockmayr, A., N. Pisaruk. Detecting infeasibility and generating cuts for MIP using CP. Int. Workshop Integration AI OR Techniques Constraint Programming Combin. Optim. Problems CP-AI-OR03, Montreal, Canada, 2003.

[5] I. E. Grossmann and V. Jain. Algorithms for hybrid milp/cp models for a class of optimization problems. INFORMS Journal on Computing, 13:258–276, 2001.

[6] Hooker, J.N., Ottosson, G.: Logic-based benders decomposition. Mathematical Programming 96 (2003) 33–60

[7] J. N. Hooker. A hybrid method for planning and scheduling. In: Proc. of the 10th Intern. Conference on Principles and Practice of Constraint Programming - CP 2004, pages 305–316, Toronto, Canada, Sept. 2004. Springer.

[8] J. N. Hooker. Planning and scheduling to minimize tardiness. In: Proc. of the 11th Intern. Conference on Principles and Practice of Constraint Programming - CP 2005, pages 314–327, Sites, Spain, Sept. 2005. Springer.

[9] R. Sadykov, L.A. Wolsey. Integer Programming and Constraint Programming in Solving a Multimachine Assignment Scheduling Problem with Deadlines and Release Dates. INFORMS Journal on Computing Vol. 18, No. 2, 2006, pp.209-217.

[10] A. Eichenberger and et al. Optimizing compiler for the cell processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[11] H. El-Rewini, H. H. Ali, and T. Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27–37, 1995.

[12] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[13] D. Kunzman, G. Zheng, E. Bohm, and L. V. Kale. Charm++, offload api, and the cell processor. In *Proceedings of PMUP Workshop at PACT'06*, 2006.

[14] M. D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In *GSPx Multicore Applications Conference*, 2006.

[15] D. Pham and et al. The design and implementation of a first-generation cell processor. *IEEE International Solid-State Circuits Conference ISSCC. 2005*, pages 184–592 Vol. 1, 2005.

[16] X. D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. In *Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, 2007.

[17] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, T. Nakatani MPI microtask for programming the Cell Broadband Engine processor In *IBM System Journal*, vol 45, N. 1, 2006

[18] D. Zhang, Q. J. Li, R. Rabbah, S. Amarasinghe A Lightweight Streaming Layer for Multicore Execution In *Proceedings of Workshop on Design, Architecture and Simulation of Chip Multi-Processors, dasCMP 2007*