CellSs: a Programming Model for the Cell BE Architecture

Pieter Bellens, Josep M. Perez, Rosa M. Badia and Jesus Labarta

April 24, 2006

Abstract

In this work we present Cell superscalar (CellSs) which addresses the automatic exploitation of the functional parallelism of a sequential program through the different processing elements of the Cell BE architecture. The focus in on the simplicity and flexibility of the programming model. Based on a simple annotation of the source code, a source to source compiler generates the necessary code and a runtime library exploits the existing parallelism by building at runtime a task dependency graph. The runtime takes care of the task scheduling and data handling between the different processors of this heterogeneous architecture. Besides, a locality-aware task scheduling has been implemented to reduce the overhead of data transfers. The approach has been implemented and tested with a set of examples and the results obtained since now are promising.

1 Introduction and Motivation

The design of processors has reached a technological limitation in the recent years. Designing more performance processors is every time more and more difficult, mainly due to power usage and heat generation. Manufacturers are currently building chips with multiple processors [1]. Although in most cases each of the processors in these multi-core chips are slower than its contemporary single-core, overall they improve the performance and are more energy efficient. Examples of these chips are several dual-core processors like the AMD Opteron or Atlon, the Intel Smithfield or Montecito, or the IBM Power4 or Power5. More challenging architectures are for example the Cell processor designed by IBM, Sony and Toshiba, with nine cores (and heterogeneous) or the Niagara with eight cores, each of them being able to handle four threads.

With the appearance of these multi-core architectures, the developers are faced with the challenge of adapting their applications to be able to use threads that can make use of all the hardware possibilities.

The first generation of the Cell Broadband Engine (BE)$^{\text{TM}}$ [2] includes a 64-bit multithreaded PowerPC® processor element (PPE) and eight synergistic processor elements (SPEs), connected by an internal high-bandwidth Element Interconnect Bus (EIB). The PPE has two levels of on-chip cache and also supports IBM's VMX to accelerate multimedia applications by using VMX SIMD units.

However, the main computing power of the Cell BE is provided by the eight SPEs. The SPE is a processor designed to accelerate media and streaming workloads. 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 a DMA engine. Most SPE instructions operate in a SIMD fashion on 128 bits representing, for example, two 64-bit double-precision floating-point numbers or long integers, or four 32-bit single-precision floating-point numbers or integers, etc. The 128-bits operands are stored in a 128-bit register file. The memory instructions also addresses 128-bit operands that must be aligned at addresses multiple of 16 bytes. Data is transferred by DMA to the SPE local memory in units of 128 bytes, enabling up to 16 concurrent DMA requests of up to 16KB of data.

The Octopiler compiler [4] implements techniques for optimizing the execution of scalar code in SIMD units, subword optimization and other techniques. For example, it implements Auto-SIMDization, which is the process of extracting SIMD parallelism from scalar loops. This feature generates vector instructions from scalar source code for the SPEs and VMX units of the PPE. It is also able to overlap data transfers with computation, to allow the SPEs to process data that exceeds the local memory capacity. To our knowledge, this is also the only approach presented that allows a higher level programming model in a Cell BE based architecture. Besides the other lower level optimizations, this compiler also enables the OpenMP programming model. This approach provides the programmers with the abstraction of a single shared-memory address space. Using the OpenMP directives, the programmers can specify regions of code that can be executed in parallel. From a single body program, the compiler duplicates the necessary code and adds the required additional code to manage the coordination of the parallelization and generates the corresponding binaries for the PPE and SPE cores. The PPE uses asynchronous signals to inform each SPE that work is available or that it should terminate. The SPEs use a mailbox to update the PPE on the status of their execution. The compiler implements a software cache mechanism to allow reuse of temporary buffers in the local memory, and therefore there is no need for DMA transfers for all accesses to shared memory. Other features, like code partitioning has also being implemented, to allow applications that do not fit in the local SPE memory.

Although the OpenMP model has been demonstrated to be powerful and valid and has a growing community of users [3], we consider that other higher programming models must be offered to the Cell BE architecture programming communities that enable to exploit the heterogeneous and parallel characteristics of this architecture.

With this goal, in this paper we present the Cell Superscalar framework (CellSs), which is based in a source to source compiler and a runtime library. The supported programming model allows the programmers to write sequential applications and the framework is able to exploit the existing concurrency and to use the different components of the Cell BE (PPE and SPEs) by means of a automatic parallelization at execution time. The only requirement we place on the programmer is that annotations (somehow similar to the OpenMP ones) are written before the declaration of some of the functions used in the application. The similarity with the Octopiler approach is that an annotation (or directive) before a piece of code indicates that this part of code will be executed in the SPEs. Therefore, similar techniques are applied to separate this part of code from the main code and generation of a manager program to be run in the SPEs that is able to call the annotated code. However, an annotation before a function does not indicate that this is a parallel region. It just indicates that it is a function that can be run in the SPE. To be able to exploit the existent parallelism, the CellSs runtime builds a data dependency graph where each node represents an instance of an annotated function and edges between nodes denote data dependencies. From this graph, the runtime is able to schedule for execution independent nodes to different SPEs at the same time. Techniques imported from the computer architecture area like the data dependency analysis, data renaming and data locality exploitation are applied to increase the performance of the application.

We would like to emphasize that OpenMP explicitly specifies what is parallel and what is not, while with CellSs what is specified are functions that are candidates to be run in parallel. The runtime will determine, based on the data dependencies, which functions can be run in parallel with others and which not. Therefore, CellSs provides programmers with a more flexible programming model with an adaptive parallelism level depending on the application input data.

In this work we focus on offering tools that enable a flexible and high-level programming model for the Cell BE, while we will rely on the Octopiler [4] or other that may appear for the code SIMDization and other lower level code optimizations.

The structure of the paper is the following: section 2 describes an overview of the system, section 3 describes the source to source compiler of the CellSs framework and section 4 the features implemented

in the runtime library. Section 5 presents some experimental results and tracefiles of real executions. Finally, section 6 concludes the paper.

2 General structure and architecture

The main objective of the environment described in this paper is to provide the users with an easy to use programming methodology which at the same time is able to produce binaries that take benefit of the Cell BE architecture. The predecessor of the work presented in this paper is the GRID superscalar environment [5, 6]. In this work, a superscalar processor is compared to a computational Grid: the processor functional units are Grid computer resources, the data that is stored in registers corresponds to files in the Grid and the assembly instructions to large (in terms of CPU time) simulations or calculations. In superscalar processors a sequential assembly code is automatically parallelized and non-dependent instructions are concurrently executed in different functional units. GRID superscalar is able to do a similar job with large sequential applications composed of coarse grain tasks, by concurrently executing non-dependent application tasks in different computing resources in a Grid. While the first generation of GRID superscalar was based on code generation and the only data dependencies that were taken into account were the ones defined by those parameters that are files, the version under development [8] is based on a source to source compiler, is able to tackle almost all type of data dependencies and besides the exploitation of the concurrency in the remote resources in a Grid, further parallelization is achieved in the local client through the use of threads.

Cell superscalar is based is this second version of GRID superscalar. The system is composed of two key components: a source to source compiler and a runtime library.

Figure 1 shows the process flow that a user application will follow in order to be able to generate an executable for the Cell BE. Given a sequential application in C language, with CellSs annotations (section 3 describes the annotations syntax) the source to source compiler is used to generate two different C files. The first file corresponds to the main program of the application, and should be compiled with a PPE compiler to generate a PPE object. The second file corresponds to the code that will be executed under request of the main program in the SPEs. This file must be compiled with an SPE compiler to obtain a SPE object, that will be linked with the SPE libraries to obtain a SPE executable. However, in order to be able to execute this program, it must be embedded in the PPE binary executed in the PPE. For this reason, the PPE embedder is used to generate a PPE object, which is then used with the other PPE objects and PPE libraries as inputs to the PPE linker, which finally generates the Cell executable.

Besides the CellSs compiler, the rest of the process is the same that must be followed to generate binaries for the Cell BE.

The main program binary is normally started from the command line and starts its execution in the PPE. At the beginning of this program the activity of the SPEs is initiated by uploading the SPE binary in the memory of each SPE used. These programs will remain idle until the main program application starts spawning work to them. Whenever the main program runs into a piece of work that can be spawned in an SPE (from here one, a *task*), a request to the runtime library is issued. The runtime will create a node representing this task in a task graph, and will look for dependencies with other tasks issued before, adding edges between them. If the current task is ready for execution (no dependencies with other tasks exists) and there are SPEs available, the runtime will make a request to an SPE to execute this task. The corresponding data transfers are done by the runtime using the DMA engines. The call to the runtime is not blocking and therefore, if the task is not ready or all the SPEs are busy, the system will continue with the execution of the main program.

It is important to emphasize that all this (task submission, data dependence analysis, data transfer) is transparent to the user code, which is basically a sequential application with user annotations that indicates which parts of the code will be run in the SPE. The system can dynamically change

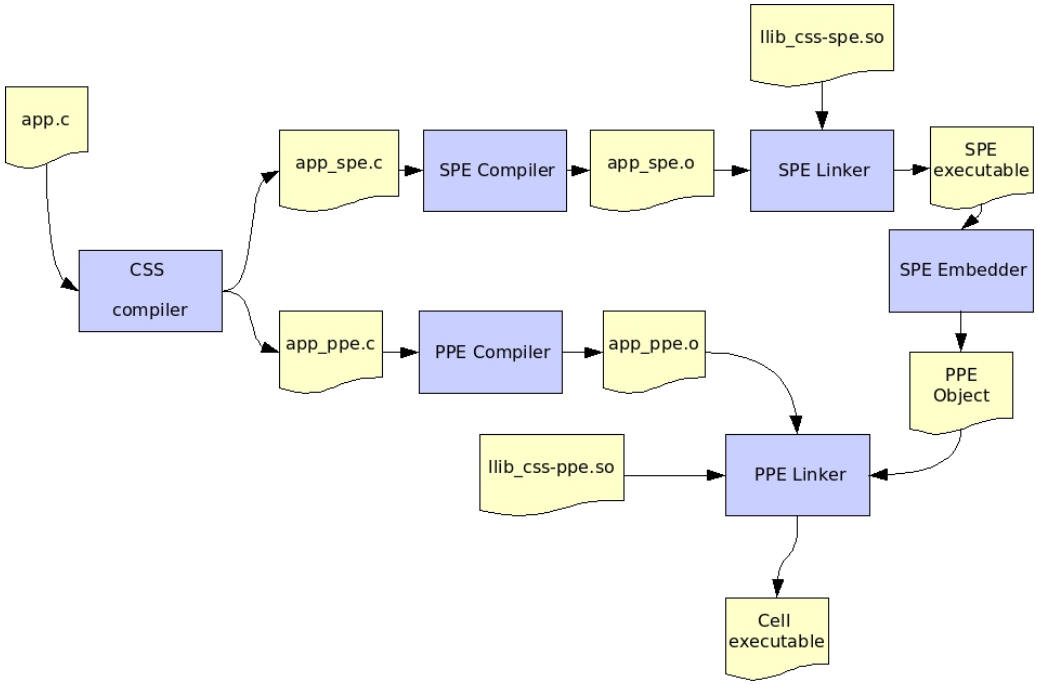

Figure 1: Steps to generate a Cell executable

the number of SPEs used by the application, taking into account the maximum concurrency of the application at each stage of the same.

3 Compiler

The current version of the CellSs environment is based on source to source C compiler that is able to process user annotations in the code. What is required from the user is to indicate those parts of the code that are to be executed in the SPEs. Since the PPE is slower than the SPEs, the candidates to be annotated are the CPU consuming parts of the application.

The current version of the source to source compiler supports the following features:

1. Provide the ability of specifying that a function is a task to be executed in the SPEs.
2. Provide the ability of specifying function parameter directions.
3. Provide the ability of specifying parameters that are 1-D arrays and their lengths.

What is to say, the compiler supports the annotation of parts of code encapsulated in a function (or procedure), that have scalar parameters or 1-D arrays. The user must indicate the direction of the parameters (input, output or input/output) in order to allow the runtime to find the data dependencies.

Figure 2 shows an example of code with annotations. Let's assume `array_op` is a computation intensive function in a user application. The annotation is indicating that this function can be executed in the SPEs and the direction of the parameters. The function has four input parameters, namely: `a`, `b`, `index_i` and `index_j`; and an output parameter, `c`. The ({ }) after the parameter names (`a`, `b` and `c`) indicate that those parameters are arrays. In this example, the user is not providing the size of the input or output arrays since it is already specified in the function interface. An equivalent annotation,

```
#pragma css task input(a{}, b{}, index_i, index_j) output(c{})
void array_op(float a[N], float b[N], float c[N], int index_i, int index_j);
main(){
    ...
    array_op(A, B, C, i, j);
    ...
}
```

Figure 2: Example of annotated code in CellSs

from the compiler point of view is shown in Figure 3, where the size of the arrays is specified. As it can be seen, the annotation of the code is very simple.

From the annotated source code, the CellSs compiler is able to generate two different sets of files: the main program of the application to be executed in the PPE and the tasks code to be executed in the SPEs. In the main program files, the compiler inserts the following code:

- Calls to CellSs runtime initializing and finalizing functions.
- Calls for registering the annotated functions. This creates a table of functions that can be later indexed by the SPE generated code.
- Substitutes de original calls to annotated functions by calls to the `Execute` function from the CellSs runtime

Besides, the original code of the annotated tasks can be eliminated, since it is not going to be executed in the PPE and therefore is not necessary in this binary. Figure 4 shows an example of generated code for the PPE for example of figure 3.

The compiler generates an adapter for each of the annotated tasks. These adapters can then be called (under request of the PPE program) from the SPE main program (tasks program), which is hard-coded in the CellSs SPE runtime library. Figure 5 shows an example of the generated code for the SPEs for the example of Figure 3.

The current implementation of this compiler is based on the Mercurium compiler [7]. This compiler was originally designed for OpenMP but since its infrastructure is quite general it has been reasonable easy to port it to enable Cell superscalar.

4 Runtime

As it has been mentioned before, CellSs environment shares most of the GRID superscalar next generation version infrastructure. Besides the compiler, the other important piece which is shared is

```
#pragma css task input(a{N}, b{N}, index_i, index_j) output(c{N})
void array_op(float *a, float *b, float *c, int index_i, int index_j);
main(){
    ...
    array_op(A, B, C, i, j);
    ...
}
```

Figure 3: Another example of annotated code in CellSs

```
main(){
    ...
    CellSs_RegisterLocalFunction ("array_op");
    Execute ("array_op", 5, IN_DIR, ARRAY_T, 1, A, N, FLOAT_T,
        IN_DIR, ARRAY_T, 1, B, N, FLOAT_T, OUT_DIR, ARRAY_T, 1, C, N, FLOAT_T,
        IN_DIR, INT_T, 0, i, IN_DIR, INT_T, 0, j);
    ...
}
```

Figure 4: Example of generated code for the PPE

```
void css_array_op_adapter (int *params, char *data_buffer)
    {
    array_op_adapter(data_buffer[params[0]], data_buffer[params[2]],
        data_buffer[params[4]], data_buffer[params[6]], data_buffer[params[8]]);
    }
```

Figure 5: Example of generated code for the SPE

the runtime library. Although a rough overview of what the runtime is able to do will be described in this section, the reader is referred to [8] for a deeper explanation. The most important change in the original user code that the CellSs compiler inserts are the calls to the `Execute` function whenever a call to an annotated function appears.

At runtime, these calls to the `Execute` function will be the responsible for the intended behavior of the application in the Cell BE processor. At each call to *Execute*, the runtime will do the following actions:

- Addition of a node in a task graph that represents the called task.
- Data dependency analysis of the new task with other previously called tasks. The data dependency analysis is based on the assumption that two parameters are the same if they have the same address. The system looks for RaW, WaR and WaW data dependencies [1].
- Parameters renaming: similarly to *register renaming*, a technique from the superscalar processor area, we do renaming of the *output* and *input/output* parameters. For every function call that has a parameter that will be written, instead of writing to the original parameter location, a new memory location will be used, that is, a new instance of that parameter will be created and it will replace the original one, becoming a renaming of the original parameter location. This allows to execute that function call independently from any previous function call that would write or read that parameter. This technique allows to effectively remove all WaW and WaR dependencies by using additional storage, greatly simplifying the dependency graph and thus improving the chances to extract more parallelism.
- Additionally, under certain conditions, the task maybe submitted for execution. This process is described in the next paragraph.

During the execution of the application the runtime maintain a list of *ready* tasks. A task is labeled as *ready* whenever no data dependencies exist between this task and the others or whenever all the data dependencies between this task and the others have been solved (i.e., the predecessor tasks have

[1] RaW, WaR and WaW stand for *Read after Write*, *Write after Read*, and *Write after Write* respectively [12]

finished their execution). The task dependency graph and the ready list are updated each time a new task appears (when calling the `Execute` function) and each time a task finish. When a task finishes, the runtime is notified (section 4.1 specifies how this is implemented for the Cell processor case) and the task graph will be checked to establish which data dependencies have been satisfied and add those tasks that now have all data dependencies solved to the ready list.

Given a list of ready tasks and a list of available resources, the runtime will choose the best matching between tasks and resources and will submit the tasks for execution. By *task submission* it is meant to perform all the necessary actions in order to execute that task: parameters transfer and request for task execution. Section 4.1 describes how this is performed for the Cell BE processor case. This resource selection is tailored to exploit the data locality between the tasks. In this sense, the runtime will try to assign tasks that have a data dependency to the same resource. Then, the data which is shared between both tasks is kept in the resource, reducing the application time devoted to data transfer. Section 4.2 describes how this is implemented for the Cell BE processor case.

Finally, the runtime has been provided with a tracing capability. When running an application, a post-mortem tracefile can be optionally generated. This tracefile can be afterwords analyzed with the performance analysis toolset Paraver [10]. Section 4.3 describes this feature.

4.1 Middleware for Cell

CellSs applications are composed of two type of binaries: the main program, which will be run in the PPE and the tasks program, which will run in the SPE. These binaries are obtained by compilation of the files generated by the CellSs compiler with the CellSs runtime libraries as described in section 2.

When starting the main program in the PPE, the tasks program will be launched in each of SPEs used for this execution. The tasks programs will wait for requests from the main program.

To be able to execute the annotated functions is the slave SPEs, the runtime should be able to: prepare all the necessary data to be transferred to the SPEs, request the SPE to start a task and, synchronize with the SPEs to know when a task finishes.

When the scheduling policy selects a task from the ready list and an SPE as resource for executing this task it builds a data structure, the *task control buffer*, which stores all the information required by the SPE to locate the necessary data for the task. The *task control buffer* contains information such as: task type identifier, location of each of the parameters and control information.

The task type identifier is just an identifier that allows the SPE to know which is the task between all the annotated that have to be executed. Regarding the input parameters, it may happen that some of them are already located in the local memory of the SPE (as a result of a previous execution) or it may be that they are in the main memory and therefore should be DMAed in before executing the task. Regarding the output parameters, it may be that some of them must be kept in the local memory after task execution or it may be that some of them must be copied back to the main memory when the task finishes (this operation will be done by DMA also). All this information is stored in the fields of the *task control buffer*.

The request from the main program to execute a task in a given SPE is done through a mailbox structure. One entry of this mailbox exists for each of the SPEs. When there is a task ready for execution for a given SPE, the main program places in the corresponding entry the request, the address of the *task control buffer* and the size of this buffer.

The behavior of the tasks program run in the SPE, is the following: if the tasks program is idle, it polls its corresponding entry of the mailbox until a task request is detected. Then, it DMAs in the *task control buffer*. From this buffer is able to understand where it is all the data that is required for the requested task, even if it is in the main memory and should be DMAed in to the local memory, or even it is already in the local memory. Once all the necessary input data has been DMAed in, the task is executed in the SPE by calling the annotated task through the generated adapter. When the

task is finished, the tasks program DMAs out some of the output data, according to the *task control buffer* and keeps in local memory the rest of data.

Besides, the *task control buffer* may contain instructions to compact the data located in the local memory. Since some of the data located in the local memory is kept from task to task and other is not, the local memory can suffer from fragmentation. The CellSs PPE runtime library keeps a map of the SPEs local memories and implements a compaction policy to reduce the fragmentation level.

Regarding the requirement in the SPEs of data alignment, the tasks program takes care of this by aligning all data in positions multiple of 16 bytes.

Regarding the synchronization between the PPE and the SPE, two different options have been implemented and tested. The first implementation is based on a MUTEX mechanism provided by the IBM CBE SDK [11]. However, when we tested this version, it turn out that the main program was missing some of the SPE callbacks to indicate that the task has finished. This was considered a bug of the IBM CBE SDK and a bug was reported. However, in the meanwhile we need a working version. Then, a second synchronization mechanism was implemented using event signaling. The tasks program communicate in two different cases with the main program: when all data required for a task has been DMAed in and when a task execution is finished (and all output data has been DMAed out). The type of event is written in a position of the mailbox and then the tasks program signals the main program using an intrinsic (`spu_hcnpeq`).

The implementation described in this section is based on the SPE Threads provided with the Cell BE system libraries.

4.2 Locality exploitation

As described above, when executing the main program in the PPE, the CellSs runtime builds a task data dependency directed graph where each node is an instance of the annotated functions. If two tasks are linked with an edge in this graph it means that at least one result of the source node is used as input by the sink node.

The objective is to reduce the amount of data that is transferred between the PPE and the SPEs and between the SPEs. The current CellSs implementation is able to keep the results of a computation in the SPE instead of transferring them back to the main memory. Then, if the task or tasks that need this result are scheduled in the same SPE, no data transfers are required. A locality aware scheduling policy has been implemented in the CellSs PPE library to tackle this problem.

At given moments of the execution (scheduling step), the policy considers the subgraph composed of the tasks in the ready list and the subsets of nodes in the task dependency graph which are connected to the ready nodes up to a given depth. This subgraph is then partitioned in as many groups as available SPEs, guided by the parental relations of the nodes. In this sense, the policy will try to group source with sink nodes, trying to reduce the number of transfers but also not reducing the concurrency of the execution. Each partition is then initially assigned to one SPE. The tasks in a partition are sent for execution independently (not as a whole). The static scheduling can be dynamically changed to solve workload unbalance, at a cost of a data transfer in some cases. This will happen when it is detected that a SPE is idle and there are no tasks left in its corresponding partition, while other SPEs has ready tasks waiting for execution in their partition. Some of the tasks will then be reassigned dynamically to other partitions. This algorithm resembles the dominant sequence clustering algorithm (DSC) [9], but the algorithm we have implement is more dynamic and adds work stealing.

Figure 6 shows an example of behavior. At a given point in the execution of an application, a task graph has been generated by the CellSs runtime (figure 6.a). Each node in the graph represent a call to a annotated function in the original code, and the edges are added by the CellSs runtime whenever a data dependency exist between them. The number indicated in the node reflects the creation order of the graph (for example, the first annotated function that was called by the application is represented

by node 1 and the node labeled with a 5 represents the fifth annotated function that was called). The tasks that do not have data dependencies with other tasks are included in the *ready list*.

At this scheduling step, the subgraph considered for partition consist of: tasks in the *ready list*, and successors of these tasks until depth 2 (i.e., direct successors, and direct successors of its successors). Consider that we have 4 SPEs available for the execution of this application. The subgraph is then partitioned in 4, and one partition is assigned to each SPE. A possible partition is shown in figure 6.b). Then, the scheduler will select one task from each partition to be spawned in each SPE. For some partitions, the selection is obvious, as for the `spe0` partition, where only task 1 is in the *ready list*. In other cases, the decision will be based on the length of the longest path from the node to the leaves of the task graph (i.e., for partition `spe1` task 2 will be selected). Finally, in other cases, as for partition `spe3` the scheduling will just pick the first node in alphabetical order (task 23). Each time a task finishes, the CellSs runtime will be notified. After identifying the SPE that is idle, the scheduler will select another ready task from the partition and spawn it for execution in the SPE. The CellSs runtime will control the required data transfers in each case. For example, for this partition, when task 1 and task 2 have finished, task 3 will be spawned for execution. The *task control buffer* will indicate that the data obtained by task 1 is already in the SPE0 local memory but the data obtained by task 2 will transferred [2].

Another possible partition is the one described in figure 6.c). In terms of data transfers, figure 6.c) is better since the data transfer of the result generated by task 2 can be kept in the local memory of SPE0, and therefore whenever task 3 is executed, no data transfers will be required. However, in terms of execution time may be inferior, since task 1 and task 2 cannot be executed at the same time in a single SPE and therefore these two tasks will be serialized.

Figure 6.d describes the dynamic behavior of the scheduling policy. The nodes in light blue (darker color when printed in b&w), represent tasks that are currently being executed in the corresponding SPE. The nodes that have disappeared from the graph represent tasks that have finished their execution. As it can be observed, partition `spe3` is now empty, and therefore an SPE will be idle. In this situation the policy will try to balance the workload by stealing ready tasks from other partitions. In this case, task 4 from partition `spe1` is ready for execution and waiting. The scheduling policy selects this node and move it to partition `spe3`, and task 4 is started immediately (see figure 6.e). In some cases this may require some data transfers from a local memory of an SPE to another. The policy will take these transfers into account trying to minimize them.

4.3 Tracing

A tracing mechanism has been implemented in the CellSs runtime which generates Paraver [10] conforming post-mortem tracefiles of the applications. A Paraver tracefile is a collection of records ordered by time where information about the events and states that the application has passed through is stored. These traces can then analyzed with the Paraver graphical user interface which allows performance analysis at different levels (i.e., task level, thread level), filtering and composing functions that allow different views of the application and a set of modules to calculate various statistics.

Although Paraver has its own tracing packages for MPI, OpenMP and other programming models, in this case a Tracing Component has been embedded in the CellSs runtime. The Tracing Component records events as they are signaled throughout the library. For example, it records when the main program enters or exits any function of the CellSs API (as for example CellSs_On, CellSs_Off, CellSs_RegisterLocalFunction), it records when an annotated function is called in the main program (and therefore a node is added to the graph), when a task is started or finished, etc.

[2]The kernel version available at BSC when writing this version of the paper does not support direct DMA transfers between SPEs. For this reason, a data transfer from SPE1 to main memory and a data transfer from main memory to SPE0 will be required implementation for this case.

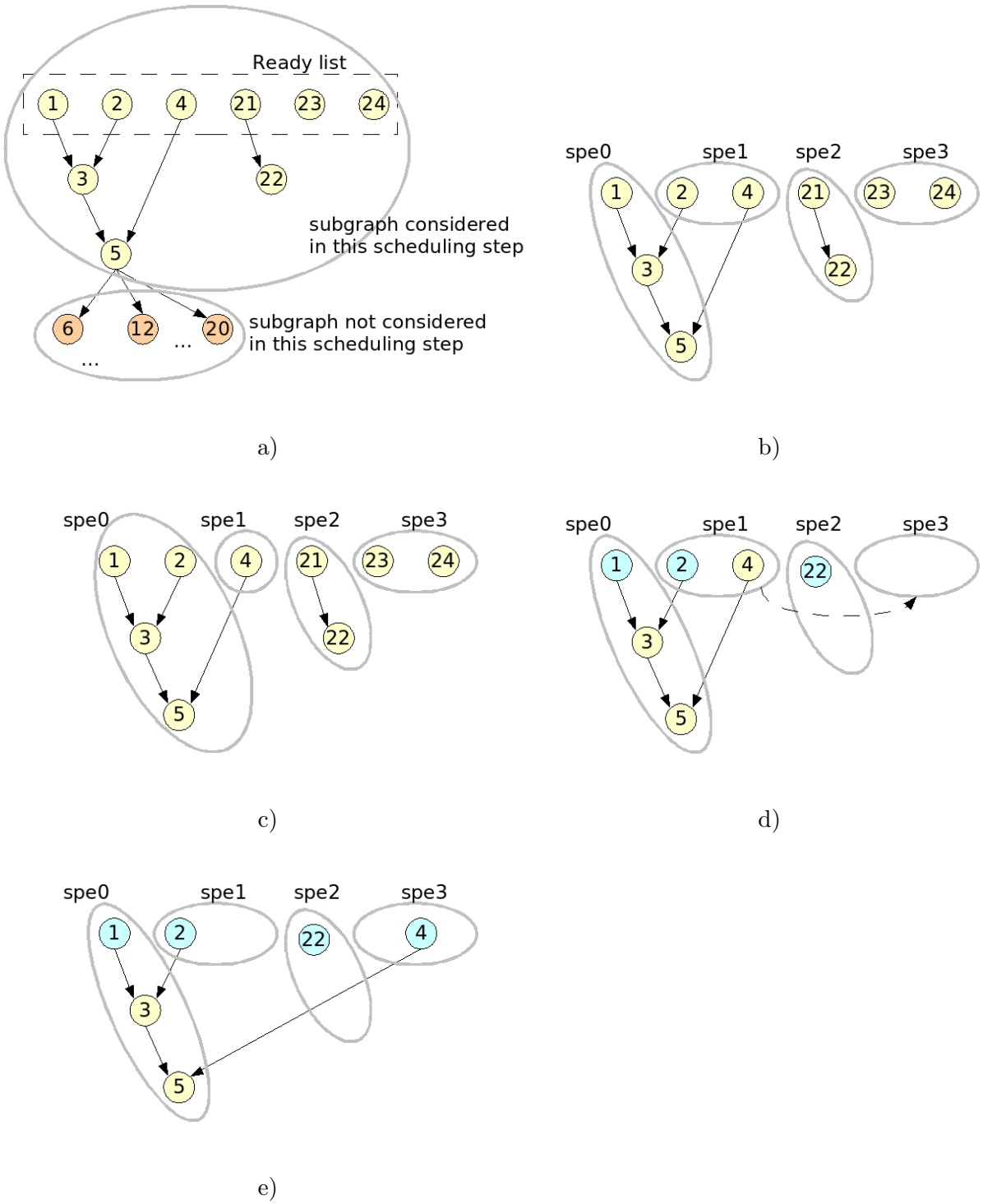

Figure 6: Locality aware task scheduling. a) Subgraph selection; b) Initial task assignment to SPEs; c) Re-assignment of task to SPE; d) Final assignment.

The traces obtained will allow to analyze the behavior of the CellSs runtime and of the application in general. This tracing capability can be enabled or disabled by the user.

5 Examples and results

Since now we have measured and analyzed the behavior of three examples: a block matrix multiplication (matmul), an exact implementation of the travel salesman problem (TSP),and a block matrix Cholesky factorization (cholesky),

The examples generate different levels of difficulty for the CellSs runtime scheduler: while the TSP generates a embarrassingly parallel graph (all tasks instances are data independent between them) and the matrix multiply generates a relatively easy to schedule graph, the Cholesky factorization generates a much more connected data dependence graph which is more challenging for the CellSs scheduler.

5.1 Performance analysis

The first tests were performed with the matrix multiply example. As it has been mentioned before, in this example the elements of the matrixes are blocks, i.e, each element of the matrixes is itself a smaller matrix. In this case, the blocks of data we used are blocks of 64×64 double precision floats, while the matrixes are of 16×16 blocks. This schema generates 4096 tasks, each of them performing a 64×64 matrix multiplication. This 4096 tasks are organized in groups of 16 dependent tasks, where each tasks reads the result of the previous one. Thus we expected that this organization was an easy to schedule graph for the locality aware task scheduler.

By using the tracing mechanism we were able to tune the initial results. For example, by means of the analysis of the trace files, we were able to observe that while the scheduler was working as expected the synchronization mechanism was reducing the performance of the application. Each time an SPE signals an event to the PPE resulted in a high time penalty. For the moment we have been able to reduce this overhead by reducing the number of signals an SPE sends to the PPE to the minimum. However, we foresee further tuning in the synchronization mechanism to increase the performance.

Figure 7 shows a sample plot of the visualization with Paraver of a tracefile. The x-axis represents the timeline and each line in the y-axis represent a thread. The PPE executes two threads: the Master thread and the Helper thread. While the Master thread executes the main program application and the runtime features, the Helper thread is in charge of hearing to the SPE threads. Each SPE executes a single thread (SPU thread `i`). In the plot, the dark blue indicates that the thread is busy and the light blue that the thread is idle. This plot is from an execution where the SPEs where signaling two events per task to the PPE. It can be easily observed that the threads run in the PPE are much more busy than the threads run in the SPEs. This situation improved a lot when we reduced the number of signals generated by the SPEs. This is just a simple example of how the trace file generation and analysis is worthwhile, at least for development purposes.

The final results for the matrix multiply example are shown in figure 8, where we can see that this examples scales almost perfectly. The figure presents the performance results obtained when running with 1, 2, 4 and 8 SPEs scaled to the case when 1 SPE is used.

5.2 Execution results

The TSP example performs an exhaustive search to find the optimum solution to this traditional problem. The scheme of the implemented algorithm is recursive, but the recursion is composed of two parts: the part that it is execution in the PPE and the part that is executed in the SPE. The level of recursion at which this decomposition is performed determines the number and size of tasks executed in the SPEs: at earlier levels a small number of coarse grain tasks are generated, and at later levels a larger number of fine grain tasks are obtained. The algorithm passes as arguments to the SPE

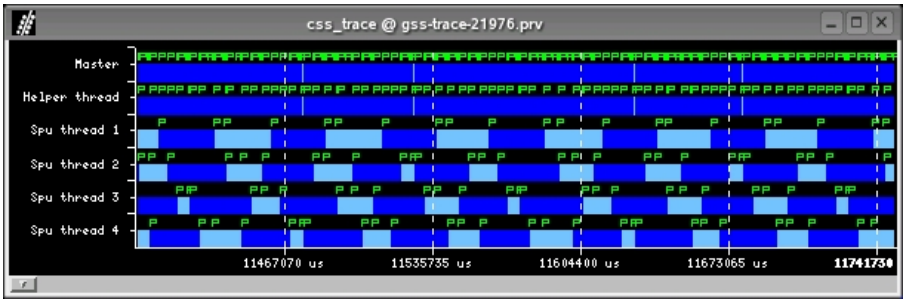

Figure 7: Sample generated tracefile

tasks, a vector with the partial path solution built until the moment. As expected, since all tasks are independent between them, the execution scale perfectly with the number of processors (see figure 8).

The Cholesky factorization is implemented as the matrix multiply with matrixes of blocks. The block-size is again 64 × 64 double precision floats, and the matrixes are of 11 × 11 blocks, resulting in 705 tasks. The challenge of the Cholesky example is duo-fold: first, these 705 tasks compose a highly connected dependency graph, and second, there are up to six different tasks of different grain-level. We consider that the results obtained for this example (figure 8) are reasonable taking into account the above mentioned reasons.

6 Conclusions

This paper presents CellSs, an alternative to traditional parallel programming models. The objective is to be able to offer a simple and flexible programming model for parallel and heterogeneous architectures. Following CellSs paradigm, input applications can be written as sequential programs. This paradigm is currently customized for the Cell BE architecture. The runtime builds a task dependency graph of the calls to functions that are annotated in the user code and schedules these calls in the SPEs, handling all data transfers from and to the SPEs. Besides, a locality-aware scheduling algorithm has been implemented to reduces the amount of data that is transferred to and from the SPEs.

The initial results are promising but there is a lot of work left, as for example: new annotations to be taken into account by the source to source compiler, improvement in the scheduling and data handling, and improvement of the synchronization mechanism between the PPE and the SPEs.

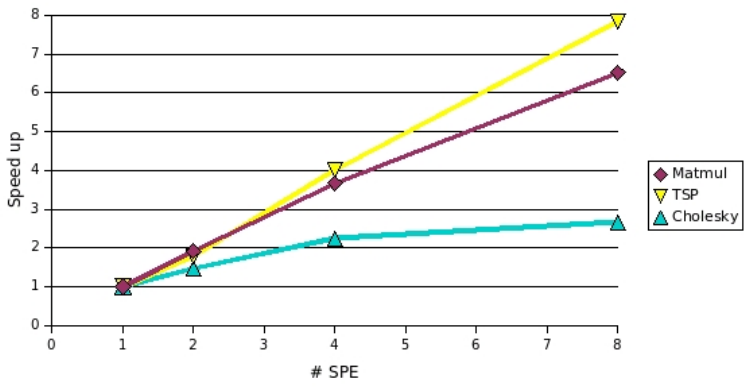

Figure 8: Results obtained for the evaluated examples. Speed up is measured against the execution with 1 SPE

References

[1] D. Geer, Chip Makers Turn to Multicore Processors, Computer, May 2005.

[2] D. Pham et al., The Design and Implementation of a First-Generation Cell Processor, in Proceedings of the 2005 IEEE International Solid-State Circuits Conference (ISSCC), 2005.

[3] The Community of OpenMP Users, Researchers, Tool Developers and Provider website, http://www.compunity.org/

[4] A.E. Eichenberger et al., Using Advanced Compiler Technology to exploit the performance of the Cell Broadband Engine^{TM}Architecture, IBM System Journal, Vol 45, Num 1, 2006.

[5] Rosa M. Badia, Jesús Labarta, Raül Sirvent, Josep M. Pérez, José M. Cela, and Rogeli Grima, *Programming Grid Applications with GRID superscalar*, Journal of Grid Computing, Vol. 1, No. 2, 151–170, June 2003.

[6] GRID superscalar homepage, http://www.bsc.es/grid/grid_superscalar/

[7] M. Gonzalez, J. Balart, A. Duran, X. Martorell, and E. Ayguad, Nanos Mercurium: a Research Compiler for OpenMP. In Proceedings of the European Workshop on OpenMP 2004. October 2004.

[8] Josep M. Perez, Rosa M. Badia and Jesus Labarta, Scalar-aware GRID superscalar, DAC technical report UPC-DAC-RR-CAP-2006-12, www.ac.upc.edu, 2006.

[9] T. Yang and A. Gerasoulis, A Fast Static Scheduling Algorithm for DAGs on an unbounded number of processors, in Proceedings of the 1991 ACM/IEEE conference on Supercomputing, 1991.

[10] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, Luis Gregoris, DiP : A Parallel Program Development Environment, 2nd International EuroPar Conference (EuroPar 96), Lyon (France), August 1996.

[11] PowerPC Hosted Environment for the Cell Broadband Engine Version 1.0.1, http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/ppc-cellsimulator-sdk1.0.1.html

[12] J. L. Hennessy, D. A. Patterson, D. Goldberg, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 2002.