Scientific Programming 17 (2009) 77–95
DOI 10.3233/SPR-2009-0272
IOS Press

CellSs: Scheduling techniques to better exploit memory hierarchy

Pieter Bellens [a], Josep M. Perez [a], Felipe Cabarcas [b,c], Alex Ramirez [a,b], Rosa M. Badia [a,d,*] and Jesus Labarta [a,b]

[a] *Barcelona Supercomputing Center – Centro Nacional de Supercomputación, Barcelona, Spain*
[b] *Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, Spain*
[c] *Universidad de Antioquia, Medellín, Colombia*
[d] *Consejo Superior de Investigaciones Científicas, Madrid, Spain*

Abstract. Cell Superscalar's (CellSs) main goal is to provide a simple, flexible and easy programming approach for the Cell Broadband Engine (Cell/B.E.) that automatically exploits the inherent concurrency of the applications at a task level. The CellSs environment is based on a source-to-source compiler that translates annotated C or Fortran code and a runtime library tailored for the Cell/B.E. that takes care of the concurrent execution of the application. The first efforts for task scheduling in CellSs derived from very simple heuristics. This paper presents new scheduling techniques that have been developed for CellSs for the purpose of improving an application's performance. Additionally, the design of a new scheduling algorithm is detailed and the algorithm evaluated. The CellSs scheduler takes an extension of the memory hierarchy for Cell/B.E. into account, with a cache memory shared between the SPEs. All new scheduling practices have been evaluated showing better behavior of our system.

Keywords: Cell superscalar, task scheduling, Cell/B.E., locality exploitation

1. Introduction

While programming models for supercomputers and clusters of SMPs have not significantly changed in recent years, the appearance of multicore chips (and the prediction of manycores in a near future) has been recognized as an inflection point in the computing history that will severely impact the way we write code [2]. The Cell Broadband Engine (Cell/B.E. hereafter) depicted in Fig. 1 serves as an example of such a device [22]. The Cell/B.E. is a multicore chip that consists of a PowerPC Processor Element (or PPE, a 64-bit, 2-way multi-threaded, in-order PowerPC processor) and multiple Synergistic Processor Elements (or SPEs, in-order, 128-bit wide SIMD cores). All of them are connected to the Element Interconnect Bus (EIB), that also couples main memory and I/O devices. The SPEs only access main memory via DMA transfers by programming their individual Memory Flow Controllers (MFCs). For each SPE, data and code reside in its 256 kB Local Store (LS). The Cell/B.E. basically is a single-chip MIMD.

From here on, the question is not only how efficient applications perform on such a parallel platform, but how productive is the source code? In this case, *productivity* means: how many source code lines did we need to add or alter to convert the application into a parallel one? To what degree does the parallel code differ from the sequential version? Can it easily be ported to other platforms (parallel or not)? To this end, Cell Superscalar (CellSs [7,37] hereafter) offers a set of tools that assist in expressing parallel applications on the Cell/B.E. The CellSs programming model hides the complexity of a parallel architecture to the programmer and enables code written with a sequential execution model in mind to behave like parallel code at runtime. As a consequence, porting legacy code or programming new applications for a parallel platform becomes much easier. It is out of the scope of CellSs to SIMD'ize the SPE code, although the authors recognize the importance of this step to achieve high performance. CellSs relies on the backend compiler for this purpose, or uses assembly vector code crafted by the user. The burden of dealing with multiple threads, synchronization and data sharing shifts from the program-

[*]Corresponding author: Rosa M. Badia, Barcelona Supercomputing Center – Centro Nacional de Supercomputación, Building Nexus II, Jordi Girona 29, 08034 Barcelona, Spain. Tel.: +34 934034075; Fax: +34 934037721; E-mail: rosa.m.badia@bsc.es.

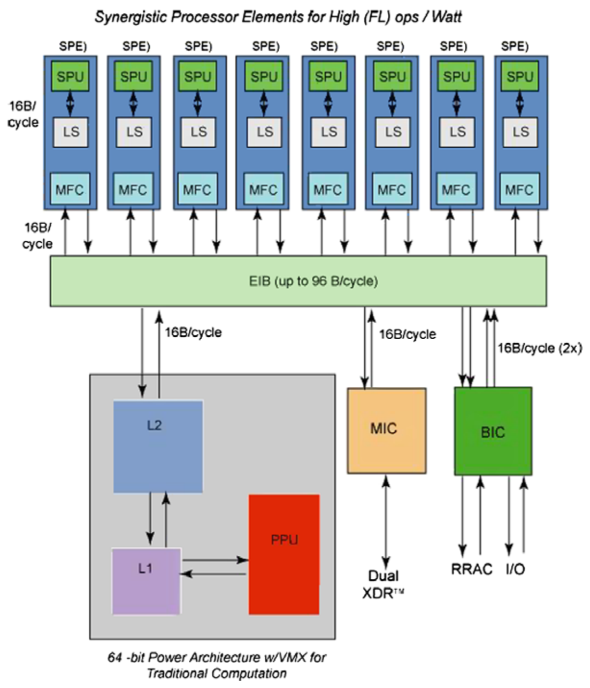

Synergistic Processor Elements for High (FL) ops / Watt
SPE)
SPU
LS
MFC
16B/ cycle
16B/ cycle
EIB (up to 96 B/cycle)
16B/cycle
16B/cycle
16B/cycle (2x)
L2
L1
PPU
MIC
BIC
Dual XDR™
RRAC
I/O
64 -bit Power Architecture w/VMX for Traditional Computation

Fig. 1. Block diagram of the Cell Broadband Engine.

mer to the CellSs runtime. Hence the implementation of each of these aspects in the CellSs runtime determines the quality of the resulting parallel code. CellSs' programming model is based on annotations (or pragmas, as in OpenMP [41]). Similarly to OpenMP version 3.0 [5], the pragmas are associated with functions, or *tasks* in CellSs terminology. The task is the unit of parallel work in CellSs. The CellSs runtime generates a data dependence graph of the tasks while it executes the application. The assignment of tasks to the various SPEs, or scheduling in short, is one of the aspects that determines the quality of the CellSs runtime. The theoretic properties of task scheduling have been well-studied, and heuristics try to bridge the gap between the NP-hard nature of this problem and practical tractability. When the focus is on a particular architecture or programming model, the scheduling problem is affected in two ways. On one hand, this additional constraint makes the scheduling problem easier, since more information about the hardware, the type of the tasks, the execution model, . . . becomes available. In general, the setting for the scheduler gets defined more sharply, and this knowledge can be incorporated into the scheduler to increase its efficiency. On the other hand, the specifics of the runtime can be exploited to guide the search for useful scheduling heuristics or to improve the quality of the produced schedules.

This paper outlines a scheduling practice based on this dual philosophy for CellSs, our programming environment for the Cell/B.E. (Section 3), and contrasts it with previous related work (Section 2). Our very general scheduling model (Section 4) distinguishes CellSs from the rest of the literature. We introduce a simple scheduling algorithm with O(N) time complexity (Section 5) for an N-task dependence graph, and demonstrate how the features of CellSs and the characteristics of the Cell/B.E. can be exploited in order to further reduce the makespan and increase the algorithm's efficiency (Section 6).

This paper contributes to the analysis, design, implementation and validation of different dynamic scheduling techniques for CellSs. Previous scheduling strategies for CellSs assigned data-independent tasks or tasks structured as a chain to the same SPE. These solutions proved to be sensible since they allowed to benefit from data locality and to reduce the number of data transfers. However, the algorithm proposed in this paper demonstrates a possible way to further improve the schedules produced by CellSs. This new algorithm is able to schedule more complicated subgraphs and make better use of the data locality of the application. This paper considers a potential extension of the Cell/B.E. that incorporates a cache memory shared between the SPEs, to analyze the locality of an application. Furthermore, we present SPE-side techniques to improve the quality of a schedule: early callback, minimal stage-out, double buffering and a software cache implemented in the SPE's local store. All techniques have been implemented and this paper documents the results for several example applications.

2. Background and related work

Given a limited set of resources and a set of tasks $\{t_1, t_2, \ldots, t_n\}$, a schedule assigns tasks to resources and starting times to tasks. Each task has a computation time or cost, and possibly requires the result of other tasks. If t_j depends on the output of t_i, there is a communication cost for transferring the result of t_i to t_j.[1] This precedence constraint is expressed through a binary relation, *data dependence*. Data dependence partially orders ('$<$') the task set. $t_i < t_j$ implies that t_i must finish before t_j executes. More accurately, the computation of t_j must not start before all the input data it requires has arrived. In the case where $t_i \nless t_j$

[1]This cost decays to zero for tasks located in the same resource.

and $t_j \not< t_i$, t_i and t_j can run in parallel. A valid schedule is a schedule that does not violate the data dependencies among tasks. The objective of scheduling is to minimize an application parameter, mostly the elapsed time or makespan. This can be achieved by maximally exploiting the available parallelism while minimizing the communication overhead. The concept of granularity [16] quantifies the ratio between task computation time and task communication time and assists in determining the quality of the schedules.

Theoretical results indicate that the NP-complete scheduling problem [11,38] only becomes tractable when factoring in constraints on the type of the tasks and the dependencies, and the amount and type of the available resources or combinations of these. In particular, the literature on this topic contains three polynomial-time algorithms for finding exact solutions (i.e. with minimal makespan) for reductions of the problem. Each of these references assumes zero communication delays and unit-time tasks, and uses a directed acyclic graph (DAG), the task dependence graph (TDG), to model the computation. Hu proposes such an algorithm for free-trees on an arbitrary number of processors [21]. Coffman and Graham confine the scheduling problem to a two-processor system, for arbitrary DAGs [12]. Interval graphs for an arbitrary number of processors have been studied by Papadimitriou and Yannakakis [34]. In the absence of the aforementioned limiting assumptions, Yang and Gerasoulis and others [10,33,43,45] have developed polynomial-time heuristics that approximate optimal solutions to varying degrees of success. An overview of algorithms and techniques for scheduling DAGs can be found in [25].

The common denominator of the scheduling algorithms mentioned previously is that they are all static, in the sense that they require the complete TDG of the application as an input. When task execution and communication times, and dependencies are known a priori, scheduling can be accomplished offline, at compile-time. Even for the heuristic solutions, a global overview of the TDG is required. The majority of these algorithms greedily attempt to reduce the critical path in the DAG, and in order to do so they require global information, e.g., the Dominant Sequence Cluster (DSC) algorithm [43] uses the level of a node to calculate its priority. Cosnard and Jeannot [13] describe a more dynamical approach for scheduling DAGs. Scheduling decisions are made at runtime, using the parametrised task graph. Their scheduling algorithm decodes the TDG as needed, but this approach still requires a description of the complete TDG all the same.

The data-flow graph (DFG) is a different paradigm to define computations. DFGs are pervasive in the area of digital signal processing (DSP), where the demand for computational power requires parallel schedules [9,31,35,36]. The DFG inherently is a static description of the application, in that it needs to be available prior to the execution of the tasks. Even in areas that do not rely on graph-theoretic notions to describe a computation, knowledge of the entire task set is required before execution starts. For example, real-time scheduling [3,4,27,29,30,44] minimally assumes that for each task that will appear in the system, the computation time τ, the period T and the deadline D are known beforehand.

Dynamic scheduling (as defined in this paper) on the other hand has limited knowledge of the task set under scrutiny and its governing dependencies. Scheduling is done at runtime, as the computation unfolds and the task set grows. This approach allows for complex control flow, since the trivial way to deal with branches is not to deal with them until they are resolved. This requires a slightly different scheduling model (see Section 4). As opposed to the aforementioned static methods, only part of the TDG is known at scheduling time. Dynamical task scheduling, defined as such, bears a lot of resemblance with instruction scheduling techniques [17,42] or techniques applied in reconfigurable computing [32].

Scheduling independent tasks on a multiprocessor system can be performed by dynamic scheduling. Similar to the problem faced in CellSs, scheduling decisions must be taken at execution time since only then the tasks and their parameters are known. In [19,20] the authors present the family of SADS algorithms that schedule tasks in a multiprocessor system taking into account the memory locality (affinity) of the tasks and the processors. However, the complexity of the scheduling problem is reduced, since SADS algorithms do not consider data dependences between tasks. In [28] the authors present WBRT, a passive scheduling system. It combines static and dynamic scheduling. Initially, the data and the computations are distributed among all processors, but dynamic load balancing reorganizes the computation as needed. WBRT only migrates tasks when data locality is preserved.

Solutions that tackle dynamic scheduling for multicore architectures are limited in number. In [8] the authors present the scheduler used in SuperMatrix, which is a paradigm that like CellSs considers out of order execution and scheduling techniques from superscalar processors. SuperMatrix, unlike CellSs, can only be applied to linear algebra matrix operations. Even though the authors claim to implement *dynamic* scheduling, tasks are queued and executed once the full task dependence graph has been build. Another important difference is that SuperMatrix does not support data renaming, which is applied in CellSs to increase the graph parallelism.

Cilk [15] and OpenMP 3.0 [5] address task scheduling in a similar fashion, since none of these approaches uses a task dependence graph, but a list of independent tasks instead. For this reason, the scheduling techniques from these environments are not fully applicable to CellSs. The new directives of OpenMP 3.0 allow the user to identify units of independent work, leaving the decisions of how and when to execute them to the runtime system. Traditional OpenMP pragma annotations includes the possibility of describing to the runtime how iterations of parallel loops are divided among the threads in the team. However, the standard does not include any proposal with regard task scheduling in the tasks' pragmas.

Both OpenMP and Cilk implement scheduling strategies that consider the data locality, like Work-first scheduling in the OpenMP case, or work-stealing for Cilk. Within the OpenMP community steps have been made towards the integration of task precedence [18] and task dependence [14].

3. Cell superscalar

The CellSs environment consists of a library and a compiler that implement a programming interface for the Cell/B.E. Basically, it offers an easy way to convert standard (sequential) C or Fortran into a parallel equivalent. The user adds pragmas to the original code to mark the functions (or *tasks*) intended to be executed in an SPE. At run time, CellSs executes the user code and internally organizes the parallel execution: it tracks data dependencies, resolves them and schedules tasks to the multiple cores.

The main program of a CellSs application runs on the PPE, together with the CellSs PPE runtime library that orchestrates the execution and delegates the execution of tasks to the SPEs. Each time an annotated task is called a vertex is added to a data dependence graph and the corresponding detected data dependences between the new task and the existing ones are added by means of edges in the graph. This is performed by the CellSs PPE runtime library, as well as the decision of what tasks to submit for execution to the available SPEs. The CellSs SPE runtime library repeats a three-phase cycle: on task availability, the CellSs PPE runtime library assigns a bundle (see Section 4) to an SPE. Then, for each task of the bundle a *stage-in* phase brings the tasks arguments to the SPE's LS and the task is computed during the *execution* phase. As the task finishes, the output arguments are transferred back to main memory during the *stage-out*. Finally, a *callback* synchronizes the SPE with the PPE by signaling the completion of the entire bundle. A callback per bundle reduces the synchronization overhead. The reception of a callback tells the CellSs PPE runtime library (PPE runtime hereafter) that the corresponding tasks can be removed from the TDG and that a free resource or SPE is awaiting the assignment of new tasks. At that point, scheduling proceeds on the modified TDG.

As stated previously, the CellSs runtime library is composed of a PPE and an SPE component. Both have their importance in the scheduling mechanism. The PPE library runs two separate threads, one of which executes the user application: the master thread. This master thread generates the tasks and takes care of the data dependence analysis based on the task arguments. Also, it renames arguments to avoid false dependencies and defines the task precedence based on the remaining true dependencies. The tasks and the associated dependence information are visible to the other thread run in the PPE, the helper thread. In turn, the latter uses this dependence information to build the TDG for the application. As the helper thread disposes of global dependence information, it can perform task scheduling, and it is in charge of the communication and synchronization with the SPEs through callbacks.

4. Schedule model

Each CellSs task consists of a pair (Fid, Arg). *Fid* identifies the function to be executed and *Arg* is the sequence of arguments for this particular instance of *Fid*. The $TDG(V, E)$ structures the vertex set V, where each $v \in V$ represents a task, according to the edge set E, where each $e \in E$ represents a data dependence between a pair of tasks (u, v). The directionality of the edge indicates for each edge $e = (u, v)$ that u is the source of the data and v the sink. CellSs constructs the TDG at run time, and at the same time offloads tasks to workers. This dynamic behavior influences the schedule model in two ways.

Firstly, scheduling necessarily advances in steps. The CellSs runtime invokes the scheduler more than

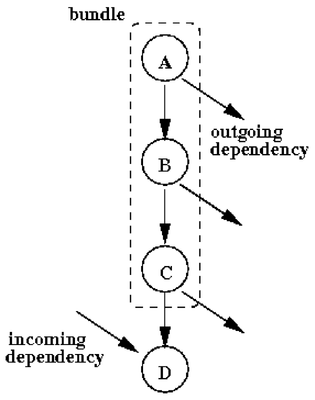


Fig. 2. Task bundle in the TDG.

once, and each time it assigns sets of tasks $B_0, B_1, \ldots, B_n$ to corresponding free resources $R_0, R_1, \ldots, R_n$. The lack of a complete TDG means that scheduling advances in bursts. As the program executes, the TDG grows, and the scheduler assigns tasks to SPEs. The scheduler outputs *bundles*: partial schedules or sequences of tasks to be executed on a specific SPE (Fig. 2). We want to amortize the scheduling overhead over multiple tasks instead of on a per-task basis.

Secondly, the TDG at the time of scheduling will represent only part of the user application. The scheduler sees a sequence $TDG_0, TDG_1, TDG_2, \ldots, TDG_m$, where $V(TDG_{i+1}) = (V(TDG_i) \setminus Finish_i) \cup New_i$, $i = 0, \ldots, m$, where $Finish_i$ is the set of tasks that have finished between step i and $i+1$, and New_i the set of tasks that have been created, and TDG_{m+1} the empty TDG, corresponding to the end of the application when all tasks have been executed.

CellSs renames task arguments (see [24]) in order to resolve output and anti-dependencies. An edge (i, j) in the TDG reflects a true data dependency from task i to task j. j depends on i, if and only if i writes an argument that j reads, with i preceding j in program order. Data dependence naturally leads to the concepts of *input*, *output* and *input–output* (or *inout*) arguments. Given the terminology introduced above, for two tasks (Fid_i, Arg_i) and (Fid_j, Arg_j): $\exists(i, j) \in E \Leftrightarrow \exists arg \in Arg_i$: $arg \in Arg_j$, arg being an output argument in Arg_i and an input argument in Arg_j.

The weight of a node in a TDG indicates the tasks execution time and edge weights serve as an indication for communication cost between the tasks. When scheduling in CellSs, we assume uniform unit task execution times and ignore communication costs. This is acceptable since DMA transfers on the Cell/B.E. can be overlapped with computation. Given enough bus bandwidth and a large task granularity, this assertion effectively holds. We adopt the unit-time task assumption for simplicity, and since the granularity of the tasks in most CellSs applications tends to be similar. The size of the LS limits the size of task arguments, and the vector capacity of the SPEs flattens out the differences in computational complexity of tasks. This is a very nice illustration of how knowledge of the hardware can simplify the scheduling model (see Section 1).

Task execution adheres to the static macro data-flow model, as detailed in [38]. Furthermore, scheduling in CellSs requires only the assignment of tasks to resources (or clustering), which implicitly defines the start times of the tasks.

5. Scheduling in CellSs

5.1. Design of the scheduler

The particular problem setting for scheduling in CellSs (Section 4) requires us to rephrase the goal of makespan minimization. It is not clear how to guide the search for a global measure such as makespan based on a partial TDG. However, every schedule that prevents the occurrence of idle gaps in an SPE trivially approximates an optimal schedule (Fig. 3). We therefore adopt idle time exclusion as a design principle in our scheduler. The two causes that possibly prevent this strategy are:

1. The lack of schedulable tasks, or *ready tasks*: these are tasks without any outstanding dependencies. This type of starvation can be due to the characteristics of the user application. The width of the parallelism can be too small to keep all the resources busy during the whole execution. Especially since CellSs only sees part of the TDG, and therefore only a fraction of the opportunities for concurrent execution.
2. The inability of the scheduler to keep pace with the SPEs. Figure 3(a), shows the scheduler unable to keep up with the execution in the SPEs. Blank periods indicate idle time. At the end of each schedule round, a communication from the helper thread instructs an SPE to start executing the bundle that has just been scheduled. As an SPE finishes the execution of its bundle, it starts idling, because the scheduler has not yet finished assigning tasks to the other SPEs.

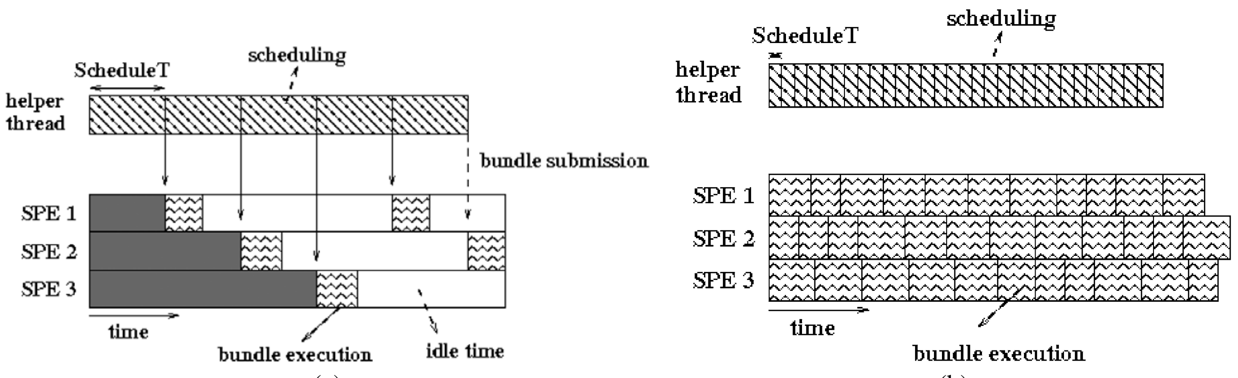


Fig. 3. A different optimality criterion for scheduling. (a) Idling, (b) no idling.

The latter occurs when the time required to schedule a bundle (*scheduleT*), multiplied by the number of SPEs, exceeds the time required to execute the bundles on the SPEs. An important characteristic related to scheduling is the task execution time or task size. Larger tasks imply more freedom for the scheduler, because its complexity clearly does not vary with the task size, while the allotted scheduling time increases. Smaller tasks require a higher throughput, hereby forcing the scheduler to finish in a shorter time frame. The implementation of the scheduler then should be such that *scheduleT* is small enough not to cause SPE idling. Anticipating a small task size and *scheduleT*, we avoid backtracking or search techniques. Essentially, each node that the scheduler considers as a candidate, should be included in the schedule, or rejected but not unnecessarily revisited. We amortize the scheduling latency by *pre-scheduling* tasks. While an SPE executes a bundle, the scheduler anticipates the pending callback and preemptively constructs a new bundle.

The makespan can further be shortened by compacting the bundle execution time. The main idea here is to reduce the number of DMA transfers to the local store, as well as to hide the latency of the remaining ones. Double buffering successfully achieves the latter. DMA transfers can be eliminated only if the objects already reside in the LS. For the scheduler, this implies that it has to take the temporal locality of the task arguments into consideration, while the CellSs SPE runtime library (SPE runtime hereafter) incorporates a caching mechanism. The SPE-side measures that reduce the bundle execution time complement the scheduling algorithm, and are discussed in Section 5.3.

To summarize, the scheduler in CellSs requires a fast algorithm that operates on a partial TDG and schedules multiple tasks per invocation. Moreover, scheduled tasks should have temporal locality. These considerations lead to the linear-time algorithm described in Section 5.2.

5.2. *Scheduling in the PPE runtime*

Figure 4 lists our scheduling algorithm. It outputs a bundle B, which we define as an ordered sequence of tasks (Fig. 2). When building a bundle B, tasks are added to the end of the sequence. A bundle can have maximally *Bmax* entries. In order to build B, the scheduler traverses lists of ready tasks, R_i, $i = 1, \ldots, N$. These tasks are the entry nodes of the partial TDG. After adding a ready task $t \in R_i$ to B, the scheduler visits its child nodes and *co-parent nodes*. If none can be found, we loop back and restart scheduling a task from a ready list. The number of outstanding dependencies of a task t, *ODep*(t) equals the dependency count of t in the TDG minus the number of t's parents in B. A bundle built as such permits to assign tasks with incoming dependencies at the time of scheduling and increases the number of valid candidates for scheduling beyond the tasks in the ready lists. Additionally, tasks found by descending into the graph or co-parent edges by definition reuse arguments. Lines 8–18 form the loop that descends into the TDG and adds a sequence of tasks without outstanding dependencies (in agreement with the definition of *ODep*) to B. This depth search halts whenever B has reached size *Bmax*, or *BTemp* becomes empty. The predicate *DepthSearch* expresses this double condition in Fig. 4. The main loop, from lines 6–22 repeats until there are no more ready tasks or B has size *Bmax* (predicate *ScheduleStop*).

A task becomes ready and enters the ready queues when it has no more incoming edges in the TDG. All

```
1:  ready lists $R_i, i = 1, \ldots, N$
2:  partial task dependency graph TDG
3:  temporary task list Btemp
4:  bundle B
5:  task set CParents
6:
7:  bool ScheduleStop = $(|B| = Bmax || \forall i$: $R_i$ empty)
8:  bool DepthSearch = $(|B| = Bmax || BTemp$ empty)
9:  while not ScheduleStop do
10:     dequeue task $t$ from the head of $R_M, M = \max\{i \mid 0 < i \leqslant N$ and $R_i$ not empty$\}$
11:     add $t$ to Btemp
12:     while DepthSearch do
13:         get task $u$ from the head of Btemp
14:         if $ODep(u)$==0 then
15:             add $u$ to $B$
16:             if HasCoParents($u$) then
17:                 CParents = processCoParents($u$)
18:                 add CParents to the front of Btemp
19:             else
20:                 retrieve a successor $s$ from the set of successors of $u$ in the TDG
21:                 add $s$ to Btemp
22:             end if
23:         end if
24:     end while
25: end while
```

Fig. 4. Basic schedule algorithm.

ready tasks are equal, but some ready tasks are preferred to others: if the descent into the TDG does not allow to extend the bundle any further, the scheduler should ideally select the "best" ready task (line 7 of algorithm in Fig. 4). Our scheduler operates on a partial TDG, and hence we are restricted to a local criterion for the quality of a ready task. In this context, a good candidate reuses as much of the objects in the LS or cache of the SPEs as possible, as argued in Section 5.1. Hence, we define the quality of a ready task in terms of the temporal locality of its arguments. To distinguish among the ready tasks according to their temporal locality, we use a hierarchy of ready queues (R_0 to R_N in Fig. 5). Tasks move up along the hierarchy according to information that the SPEs send back to the scheduler: each SPE runtime constructs an array of *locality hints* in main memory at runtime (*LocHints*). The hints could be related to recently used objects in the LS, objects that have been transferred back/from main memory, or combinations of these. Equivalently, each task in a ready queue has an associated set of objects that it uses. These are recorded in a global queue in main memory: *ReadyLocs*. A fresh ready task t starts off at ready list R_0. In Fig. 5, suppose a task t is in

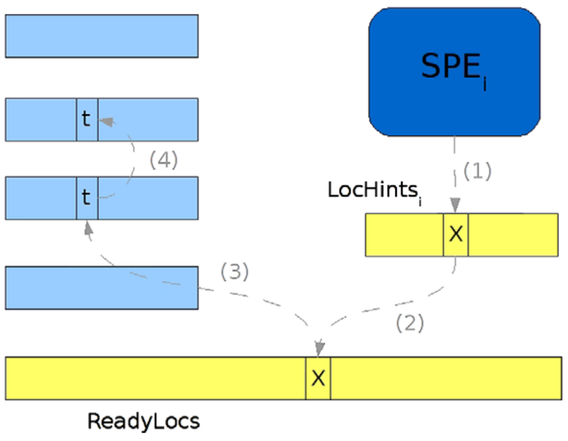


Fig. 5. Locality feedback mechanism.

ready queue R_i. t uses an object X for its computation, so an identifier for X is recorded in *ReadyLocs*. If an SPE executes a task that causes object X to enter the memory hierarchy, the SPE records the temporal locality by entering the identifier for X in its associated *LocHints*. For *SPE*$_i$, X enters *LocHints*$_i$ (step (1) in Fig. 5). Each time the scheduling infrastructure detects a match between an entry in *ReadyLocs* and an entry in a *LocHints* queue, the corresponding task is upgraded in the ready queue hierarchy. In Fig. 5, the match is detected at step (2), after which step (3) bumps the associated task t from ready queue R_i to ready queue R_{i+1}. The traversing and matching of these data structures can be performed out of the critical path of the scheduling algorithm, at the cost of losing accuracy (e.g. while awaiting callbacks).

The concept of "*co-parent tasks*" improves the search for tasks with no outstanding dependences and stimulates argument reuse. In particular, this simple extension allows the scheduler to recognize join-subgraphs as in Fig. 6(b). Straightforward descent into the TDG has problems finding eligible tasks for all but the simplest TDGs. In Fig. 6(a), the scheduler would not be able to schedule task 9 after scheduling task 1, since its parent task 2 would likely not have been found first. Here, tasks 1 and 2 are co-parents of 9. To solve this shortcoming, our runtime adds "co-parent edges" between tasks that share a child (Fig. 6(b)), so that a child will not be scheduled before all his co-parents. At scheduling time, the call to *processCoParents* returns a set of tasks. Suppose tasks u and y are co-parents of a task z, i.e. there is a co-parent triple (u, y, z) and u is a candidate for scheduling. If y has already been scheduled, then $z \in$ *CParents* and $u, y \notin$ *CParents*, else $y \in$ *CParents* and $u, z \notin$ *CParents* (Fig. 7).

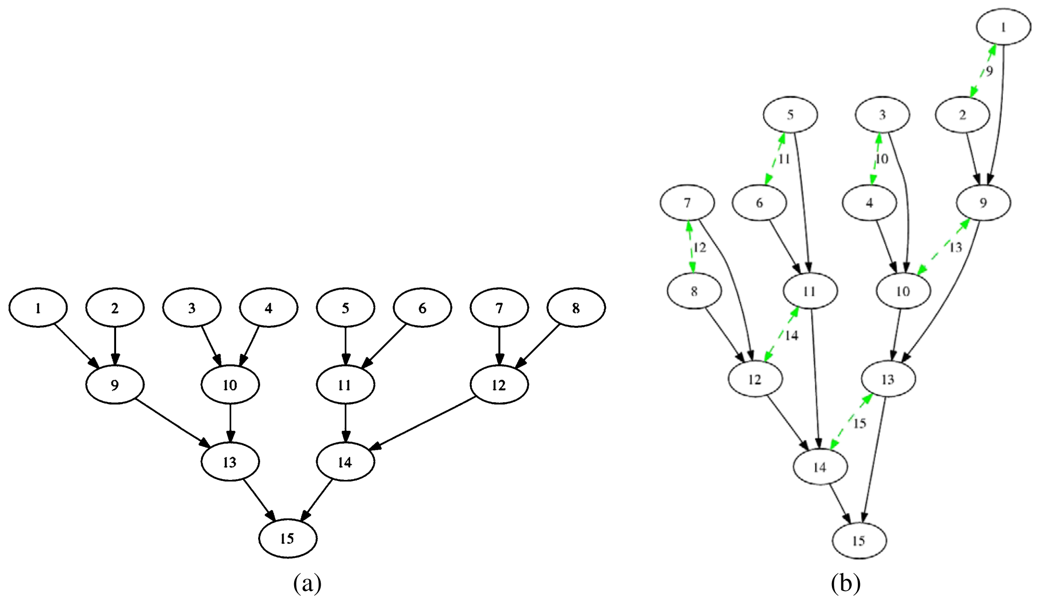


Fig. 6. TDGs for a hierarchical reduction. Tasks are labeled according to the order in which they are generated, or equivalently, the sequential program order. Co-parent edges are labeled with the label of the task that generated the co-parent relation. (a) Basic TDG, (b) TDG after identifying co-parents.

```
1: task $u$
2: task set CParents
3: **for** all $coparent - triple(u, y, z)$ **do**
4:     **if** scheduled($y$) **then**
5:         add $z$ to CParents
6:     **else**
7:         add $y$ to CParents
8:     **end if**
9: **end for**
```

Fig. 7. processCoParents(u) of algorithm in Fig. 4.

5.3. *Scheduling assistance in the CellSs SPE runtime library*

The SPE runtime complements the PPE runtime scheduling algorithm with a few performance-enhancing techniques, including double buffering. Part of their interest lies in the relative independence from the PPE runtime, the scheduler in particular. The majority of these techniques try to decrease the bundle execution time by overlapping or bypassing phases in the SPE runtime cycle (see Section 3). The scheduling strategy proposed in this paper defines the makespan of the application as the maximum over all SPEs of the sum of the execution times for all the bundles (plus the time spent in the PPE), barring idle time. A reduction of the bundle execution time then equally shortens the makespan.

5.3.1. *Early callback*

The placement of the callback at the very end of a bundle sometimes delays the discovery of ready tasks. For example, in Fig. 8(a), the sooner the PPE runtime receives the confirmation that task A has finished, the better. The end of this task opens up parallelism for this application. The earlier the scheduler can access this region, the better the quality of the following schedules, and the better the performance. Therefore, for this type of bottleneck tasks, CellSs performs an early callback. In general, this situation arises whenever a task has more than one outgoing dependency. When scheduling task A, the algorithm in Fig. 4 already queries the children of A in the TDG, so detecting whether this improvement applies at scheduling time essentially is free of cost.

5.3.2. *Minimal stage-out*

In the final stage of the three-part task cycle in the SPE runtime (Section 3), the output arguments of a task are transferred to main memory. Under certain circumstances, this stage-out can be avoided without changing the program semantics. Given an argument A, if another task in the bundle overwrites A, and it can be proven that no other SPE requires the current value of A, A does not have to be copied to main memory. Again, this strategy does not need the participation of the PPE runtime, and can be implemented without complicating the scheduler.

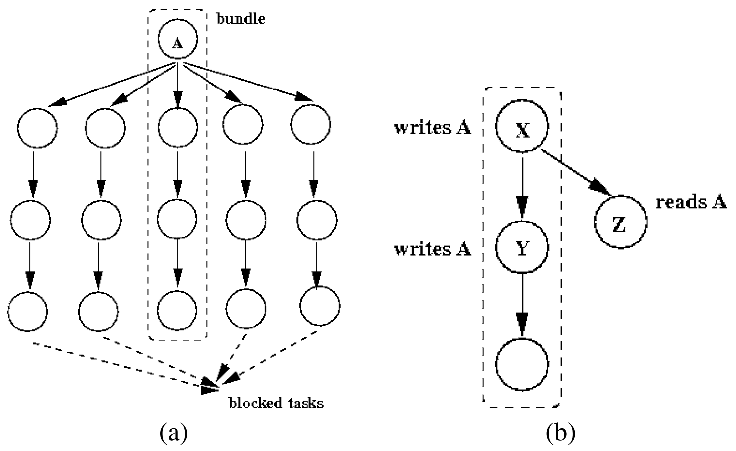


Fig. 8. SPE-side techniques for reducing the makespan. (a) Sample graph for early callback, (b) stage-out reduction.

5.3.3. Caching

Each SPE runtime maintains a software cache populated with task arguments inside the local store. It uses a LRU replacement strategy, and the decision of which type of argument to cache (in, out or inout) is configurable. The scheduler in the PPE runtime library is unaware of this behavior of the SPEs. As such, the scheduler does not take argument reuse into account when assigning tasks to an SPE. Rather, it is the SPE runtime that tries to exploit the locality that inherently presents itself in the task bundle. This relieves the PPE runtime from keeping track of the location of arguments in the SPE, or from doing expensive lookahead to determine at scheduling time which arguments should be cached. The software cache identifies each object with its main memory address and a version number. At stage-in, a hit in the software cache avoid a DMA transfer from main memory to the local store. A miss makes the object eligible to be cached itself. Our distributed software cache resembles the one in [6], but with variable-sized objects as units instead of cache lines.

5.3.4. Double buffering

This well-known technique has been thoroughly described and illustrated [23]. In an SPE, CellSs overlaps DMA transfers with computation and uses a software pipeline for iterating over the sequence of tasks in a bundle (Fig. 9). The stage-in phase has been separated into a part that starts the asynchronous DMA transfer, and a part that waits for the transfer to end (the tail and the head of the arrows in Fig. 9, respectively). The dark areas mark the parts where the SPE waits on an asynchronous DMA operation. Blank areas are stage-in phases, barred areas indicate the stage-out phases, grey areas task execution. The depicted stage-in and stage-out phases represent the start of the asynchronous DMA transfers.

6. Experiments

All measurements were conducted with a prototype of CellSs on a Cell blade at the Barcelona Supercomputing Center and the presented numbers average fifty executions. For each instance we ran the application with a different set of parameters for our scheduler, and the TDGs were unrolled 1–10% before starting execution. The results in Section 6.2 have been obtained with CellSs 1.4 and a prototype of the locality scheduler. We present results for the following applications:

matmul: A blocked matrix multiplication, implemented with the kernel from the Cell SDK.

sparselu: A blocked LU decomposition, that computes L and U and checks if $A = L \times U$ up to a certain accuracy.

choleskyC: A blocked Cholesky factorization. The TDG for a small execution is depicted in Fig. 10(b). The matrix is traversed by columns to perform the factorization.

choleskyR: A blocked Cholesky factorization, but here the matrix is traversed by rows. This algorithm is slightly different from *choleskyC*.

reduct: This is a symmetric, hierarchical reduction of an array. The array is divided into vectors, and the vectors are combined according to the pattern in Fig. 6(a) to compute the reduction.

fft3d: A 3D-FFT of a cube of complex numbers, computed as a series of FFTs and transposes of the various planes [39,40].

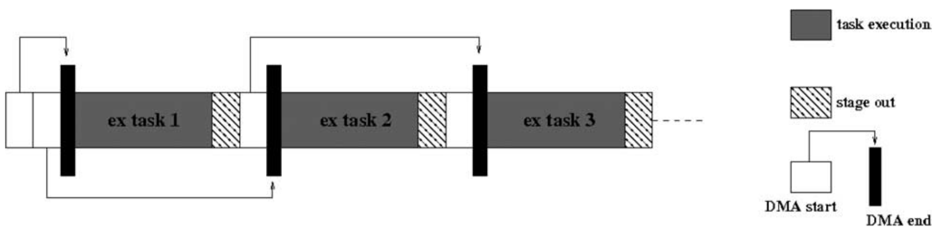


Fig. 9. A software-pipelined execution in the SPE.

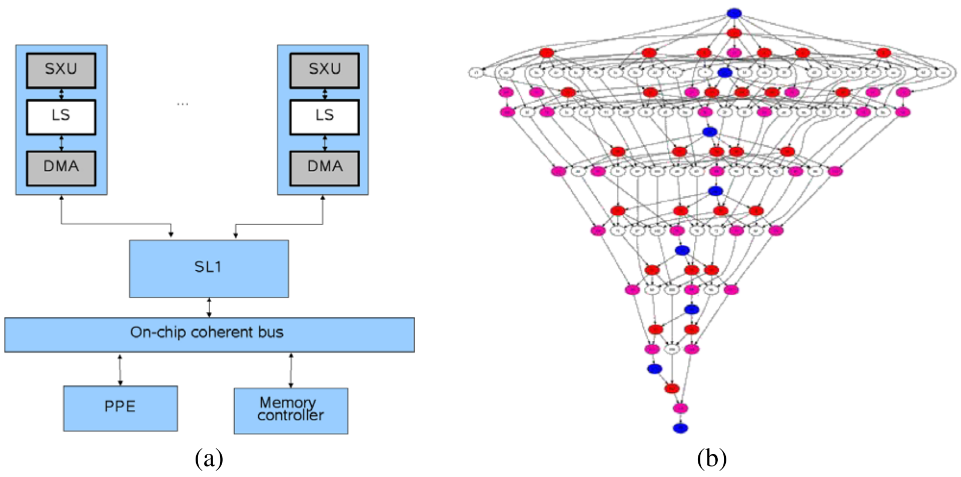


Fig. 10. (a) Block diagram of the architecture, (b) TDG for *choleskyC* for an 8 × 8 float matrix of 64 × 64 blocks. Different task types have different colors.

The source code for these applications can be downloaded from our website. Unless mentioned otherwise, the default block size is 64 × 64 ($BS = 64$). For *matmul* and *sparselu* the input matrices consist of 32 × 32 blocks ($NB = 32$). For *choleskyC* and *choleskyR*, the input matrices were scaled to 48 × 48 blocks ($NB = 48$). *reduct* reduces an array of 16384 × 4096 elements and each vector consists of 4096 elements ($NV = 16384$, $VS = 4096$). These dimensions were chosen in order to roughly generate the same number of memory accesses for each application. The cube in *fft3d* contains 256 × 256 × 256 elements ($CS = 256$). The FFTs are performed on slices of 256 × 32 elements ($BS = 32$), and the transpositions on blocks of 64 × 64 elements ($BS\_TRS = 64$).

To reflect the productivity factor of the CellSs programming model mentioned in the Introduction, Table 1 indicates the number of lines added to each of the codes mentioned above to convert them from pure sequential to CellSs applications. Most of these lines are pragma annotations that have been added and that will be ignored when compiling for a sequential architecture. Therefore, is not only that there are a few changes added, but that the code keeps very similar to the original one.

Table 1
Productivity of CellSs in the application cases

Application	# Original lines	# Added lines
matmul	262	4
sparselu	560	14
choleskyC	190	6
choleskyR	489	8
reduct	142	5
fft3D	349	18

To analyze the temporal locality of these benchmarks under CellSs, we used DMAsim, a memory simulator, that simulates an extension of the Cell/B.E. memory model (Fig. 10(a)): we assume the presence of an SL1-cache as described in [22]. Additionally, an SPE manages its own software cache (Section 5.3.3). Both caches use a LRU replacement strategy and are fully associative. With this setup, the presence of temporal locality can be quantified by the amount of reuse in the software cache and the SL1. The reuse in the software cache is measured by the number of cache hits. Reuse in the SL1 is measured by a dual figure, namely the amount of accesses to main memory. As the application succeeds in reusing objects from the SL1-cache, the number of main memory accesses decreases.

An application under CellSs potentially benefits from an improved temporal locality of task arguments (Section 5.2). This section omits performance results that prove that our locality scheduler is able to convert the gain in temporal locality into a reduction of the makespan. A gain in locality translates into an improvement in performance through a severe reduction in DMA transfers from and to main memory. But the improved locality will only become manifest in the presence of a hardware cache, which the Cell/B.E. lacks, although it has been proposed as a future extension to the architecture. A larger LS or the use of *shortcircuiting* (see Section 8) are alternatives that improve the execution time via good temporal locality. Therefore we would like to stress that the following experiments aim to demonstrate CellSs' ability to extract temporal locality by reordering tasks. The impact of such a locality scheduler on the execution time and performance is left as future work. Consequently, the measurements in Section 6.3 are expressed in units of memory accesses per execution instead of GFlops. The experiments in Section 6.2 on the other hand evaluate the worker-side techniques that help to reduce the bundle execution time (Section 5.2). In this case the choice for GFlops as a unit of measure is perfectly appropriate.

To summarize, Section 6.2 analyses the effect of the SPE-side techniques on the execution time of a CellSs application. Section 6.3 quantifies our major interest: whether the scheduler proposed in Section 5 succeeds in bringing out the temporal locality of an application. Sections 6.5 and 6.6 briefly introduce the extension to multiple SPEs and the importance of the block size respectively.

6.1. DMAsim

Figure 11 shows the structure of DMAsim for 2 clusters. DMAsim can simulate the DMA traffic generated by the SPEs of a Cell/B.E. compliant processor [23]. DMAsim is driven by a trace of an application's DMA operations. Each DMA operation is identified by the SPE it originates from, the starting time, the address, the size (in bytes) and type (get or put).

The traces used as input to DMAsim in this paper have been obtained from the Paraver [26] trace that is generated by an instrumented CellSs application.

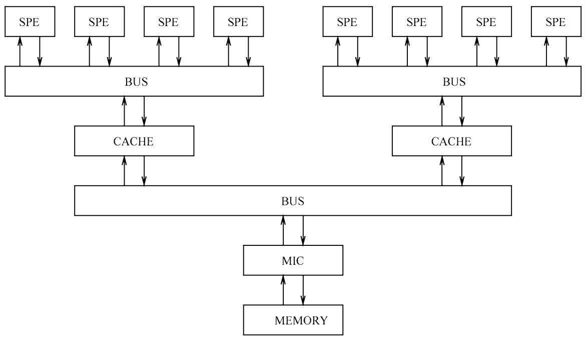


Fig. 11. Simulator structure.

The MFC of an SPE divides the DMA operations in blocks of 128 bytes [1] to be transferred through the Cell EIB to their destiny. Since DMAsim is concerned only with the DMA operations that originate on the SPEs and go to the main memory, the EIB was replaced by a single-ring bus that can transfer one 128-byte package each 5 ns (the clock cycle of DMAsim). This gives a maximum bandwidth of 25.6 GB/s bandwidth for the bus and the SPEs in each direction.

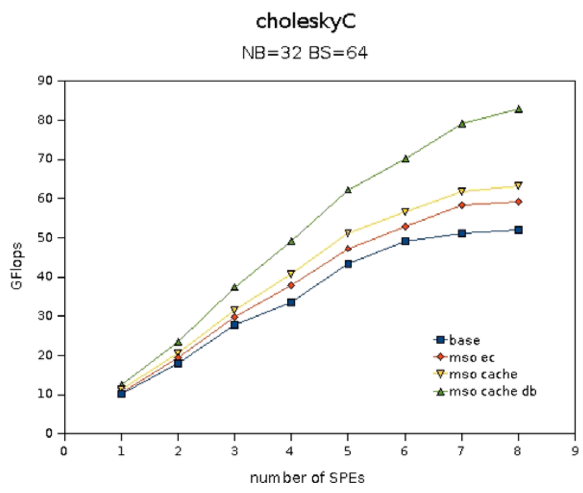
The main objective of DMAsim is to analyze the data impact on the cache, shared between the SPEs, and on the traffic that goes to main memory.

The simulator reads the DMA trace and assigns each entry to its corresponding SPE. The SPE divides the DMA in 128-byte packages. Each cycle the bus passes a token and allows one SPE to send a package from an active DMA to the cache in a round-robbin fashion. For more than one cluster, the bus that connects the caches, uses a priority token to select the cluster that has access to memory. A DMA is considered active if the internal simulator time is greater or equal to time recorded in a DMA trace entry. It is possible that one SPE has more than one active DMA, in this case the SPE would always choose packages from the oldest active DMA.

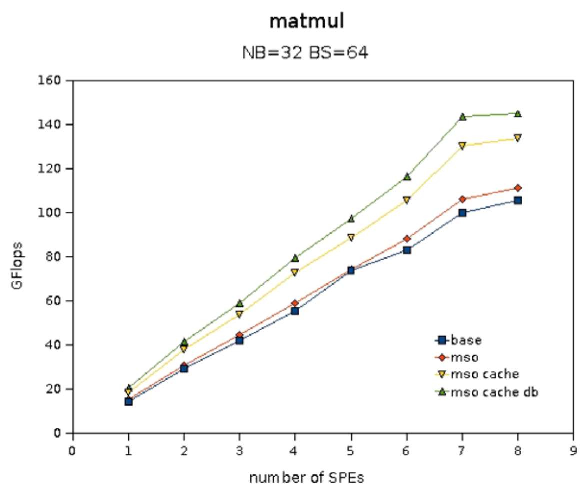
Each selected package from the bus is sent to the cache. The cache simulates the hits and misses, and models the traffic to main memory (no miss information/status handling registers (MSHR) or latency are considered). The traffic generated by the cache is passed to main memory, which simulates the page hits and misses in the cache. The cache can be configured with any number of banks, page size, and interleaving strategy.

6.2. Scalability and worker-side techniques

We quantify the impact of the SPE-side techniques that assist the scheduler described in Section 5.3:



(a)



(b)

Fig. 12. Reducing the makespan using SPE-side techniques. (a) Cholesky, (b) matmul.

minimal stage-out (mso), early callback (ec), caching (cache), and double buffering (db). We selected two benchmarks from our experimental test set and incrementally enabled each of the mechanisms. As can be observed from Fig. 12, for both *choleskyC* and *matmul*, these techniques reduce the execution time of the application, so the performance improves.[2] The TDG of *matmul* consists of tasks with no or one single incoming edge, because of the inout argument that accumulates the result for a block. It follows that no benefit can be achieved from early callbacks for this particular application. The rest of the worker-side techniques improve the makespan of *matmul* (Fig. 12(b)). Without these optimizations, the need for bus bandwidth is higher, and each additional SPE adds to the bus contention and further slows down the execution and hurts the speedup. With all optimizations enabled, *matmul* scales perfectly up till 4 workers. The speedup is 4 for 4 workers, and steadily declines for an increasing number of workers, with a speedup of 7.2 for 8 workers. For *choleskyC*, the TDG no longer is trivial (Fig. 10(b)). The speedup for *choleskyC* is 7.8 for 8 workers.

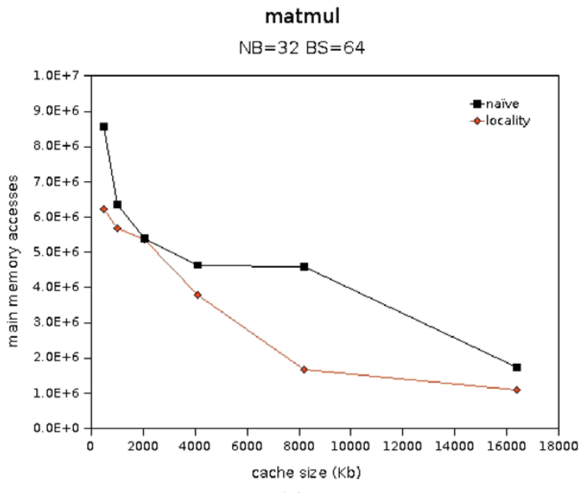
6.3. *Locality exploitation*

We are interested in the temporal locality our scheduler is able to detect or generate. For that purpose, we compare the number of memory accesses generated by executions with the locality scheduler with the behavior of the application under a naive scheduler. A naive scheduler assigns tasks according to the sequential program order. We want to qualify how the memory access pattern generated by the locality scheduler differs from the access pattern according to program order. In this section we execute the application on a single SPE. This eliminates noise caused by the interaction between the DMA transfers of the various SPEs. First we want to evaluate whether the locality scheduler is able to improve the temporal locality. The interaction with more than one SPE is only secondary.

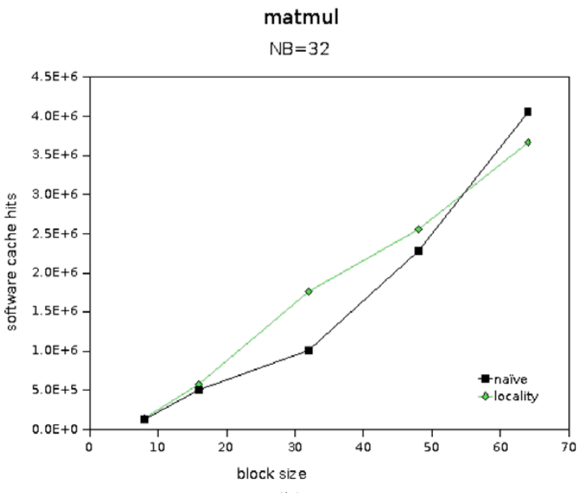
For *matmul*, except for a cache size of 2048 kB (see Fig. 13(a)), the access pattern to main memory has been significantly improved. Especially for a cache size of 8192 kB the locality scheduler succeeds in improving the temporal locality. Here, the accesses to main memory have been reduced by more than 50%. As can be seen from Fig. 13(b), this increase in temporal locality in the SL1-cache comes at the expense of a decreased hit rate in the software cache. For a block size of 64×64, the amount of hits in the software cache is lower for the locality scheduler than for the naive scheduler, but the number of accesses to main memory decreases nevertheless because of good locality in the SL1-cache. For smaller block sizes, the locality scheduler does a better job than the naive scheduler at utilizing the software cache.

In the case of *sparselu*, the analysis is slightly different (Fig. 14). For all block sizes, the locality scheduler makes better use of the software cache compared to the naive scheduler. For all sizes of the SL1-cache, this results in an improvement of the temporal locality. The locality scheduler gains a $5-15\%$ improvement in

[2]The matrix size for choleskyC in this case is 2048×2048. For a size of 4096×4096 CellSs achieves up to 125 GFlops.

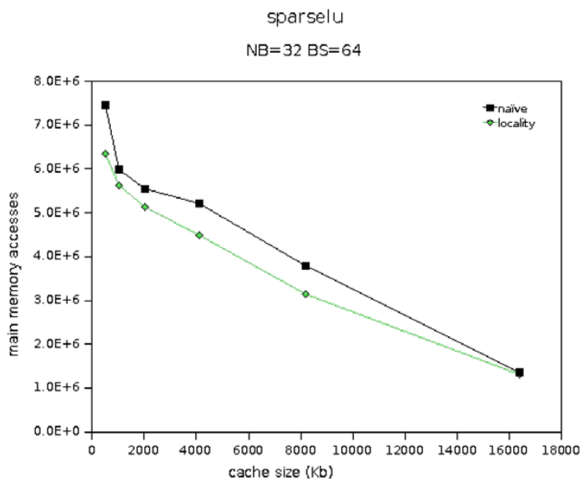


(a)

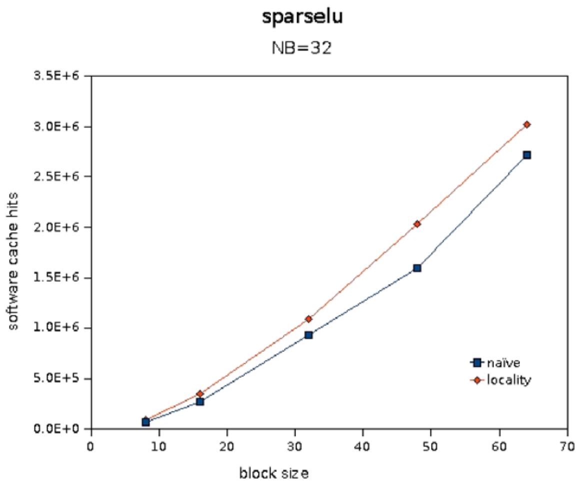


(b)

Fig. 13. Temporal locality for *matmul*. (a) Locality in the SL1-cache for *matmul*, (b) locality in the software cache for *matmul*.



(a)



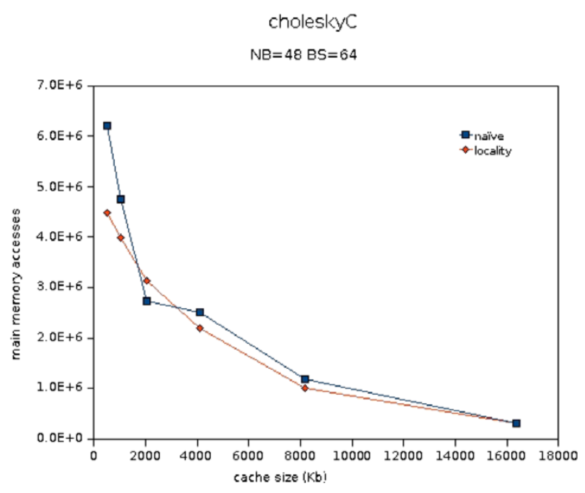
(b)

Fig. 14. Temporal locality for *sparselu*. (a) Locality in the SL1-cache for *sparselu*, (b) locality in the software cache for *sparselu*.

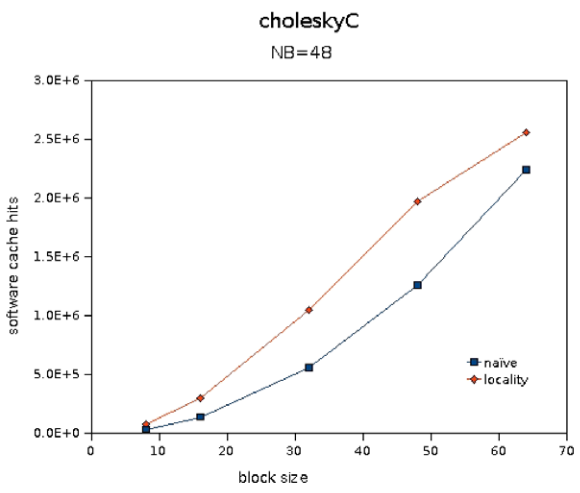
memory accesses for this application for an SL1-cache size between 512 and 8192 kB.

Both *choleskyC* and *choleskyR* exhibit a better software cache usage for the locality scheduler. For smaller block sizes, *choleskyC* maximally achieves a 55% increase in the software cache hit ratio. The hit ratio improvement tops off at 16% for a block size of 64 $\times$ 64 (Fig. 15(b)). For this largest of block sizes, the locality scheduler improves the accesses to the SL1-cache with 25% for a cache size of 512 kB, and from there on steadily converges with the naive scheduler, with an outlier at a cache size of 2048 kB (Fig. 15(a)). *choleskyR* does not improve its software cache accesses to the same extent as *choleskyC*: for a block size of 64 $\times$ 64 the number of hits increases by 15% (Fig. 16(b)). Nevertheless, the improvement of the temporal locality due to the locality scheduler results in a 10–30% decrease in accesses to main memory for an SL1-cache size of 2048 and 8192 kB, respectively (Fig. 16(a)).

The original source code for *reduct* is a generic, straightforward implementation of the algorithm. It is a sequential implementation, written without considering the memory hierarchy, parallelism or scheduling issues. We simply added a pragma to the function that does the vector reduction, to indicate the CellSs task.

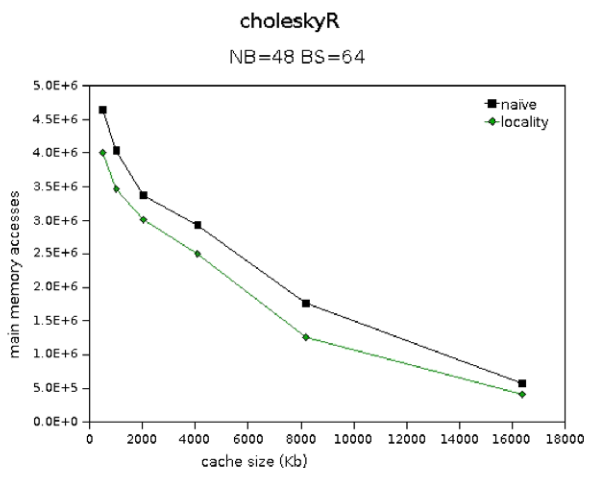


(a)

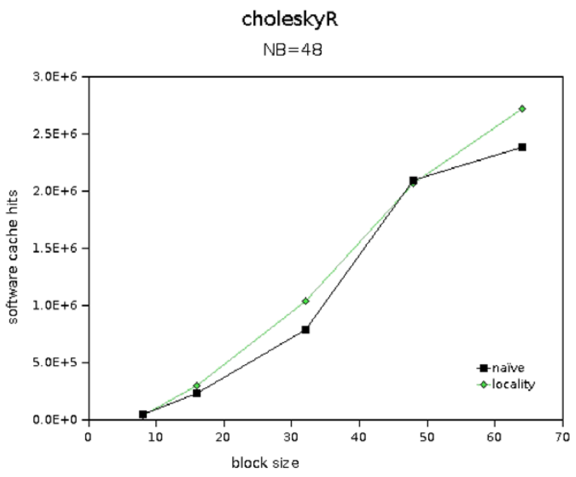


(b)

Fig. 15. Temporal locality for *choleskyC*. (a) Locality in the SL1-cache for *choleskyC*, (b) locality in the software cache for *choleskyC*.



(a)



(b)

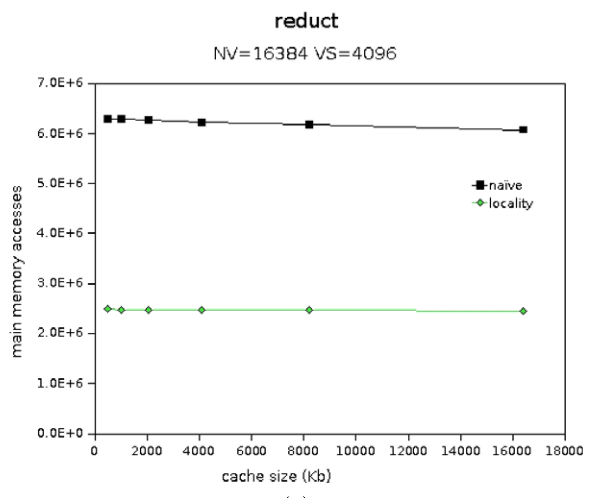
Fig. 16. Temporal locality for *choleskyR*. (a) Locality in the SL1-cache for *choleskyR*, (b) locality in the software cache for *choleskyR*.

This reduces the number of main memory accesses by 60% on average, and the application makes much better use of the software cache (Fig. 17).

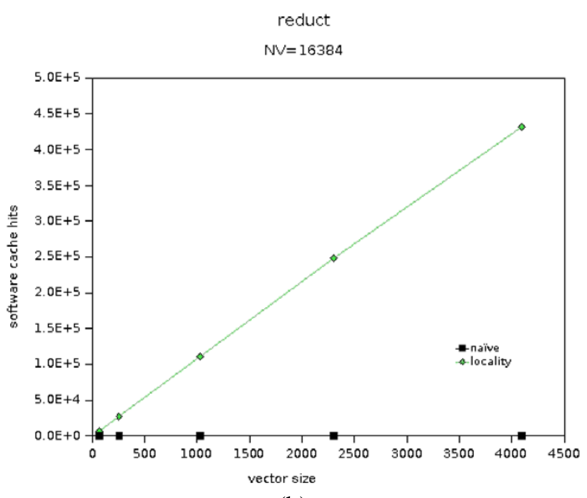
For *fft3d*, the results are comparable to *reduct*, although less outspoken (Fig. 18). The number of main memory accesses decreases by 6–7%. The locality scheduler makes better use of the software cache as well, but since the task arguments have non-uniform sizes, it is hard to calculate the exact number of hits.

6.4. *Locality with software cache disabled*

We found that the same simulations and measurements with the software cache disabled delivered no different results. Figure 19 compares an execution of *choleskyC* without software cache with an execution with a software cache for the case of the locality scheduler. Neither is there a difference for the number of main memory accesses for the same experiment but with the naive scheduler. The software cache is replicated in the SL1-cache, so the number of main memory accesses is the same regardless of the presence of a software cache. The advantage of a software cache hit is the immediate availability of the object (in that sense the software cache functions as a higher level in the memory hierarchy for an SPE) and the reduction of MFC traffic.



(a)



(b)

Fig. 17. Temporal locality for *reduct*. (a) Locality in the SL1-cache for *reduct*, (b) locality in the software cache for *reduct*.

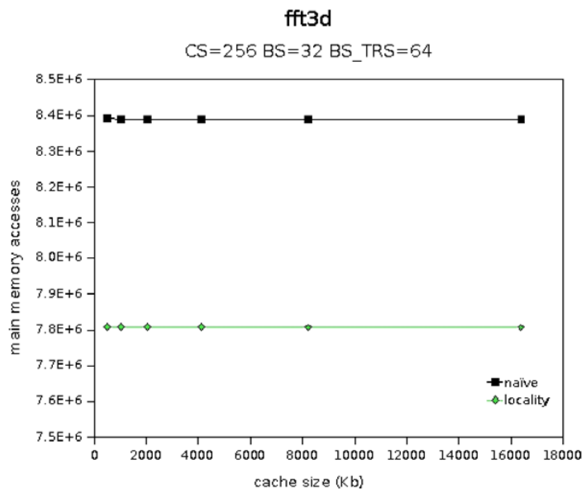


Fig. 18. Locality in the SL1-cache for *fft3d*.

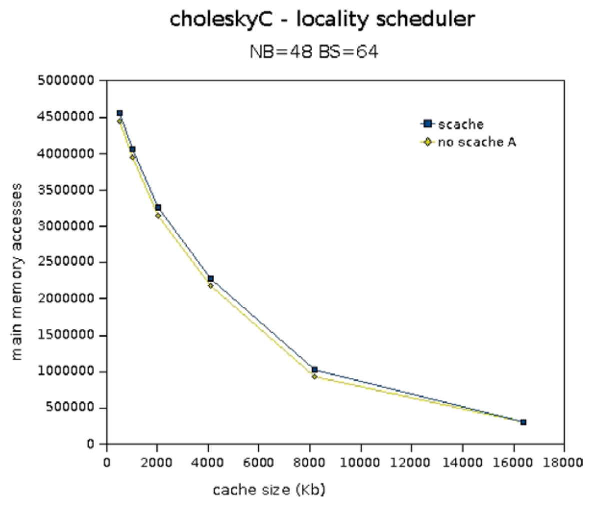


Fig. 19. Locality in the SL1-cache for *choleskyC*, with and without software cache.

6.5. Extension to multiple SPEs

In this section, an "access" is a transfer operation from/to the SPE. As such, an access can be satisfied by the SL1-cache or by main memory. A "memory access" is an access that goes to main memory (because it misses in the SL1-cache). Depending on the use of the SL1-cache, the number of main memory accesses can increase or decrease with the number of SPEs. The "working set" is the union of all the objects that have been used recently and still carry reuse, and the objects that reside in the SL1-cache. Ideally, both sets involved in the union coincide.

A naive scheduler for multiple SPEs can be constructed by sending a ready task to an SPE in a round-robin fashion, and waiting for the callback before advancing the next task. Incidentally, such a scheduler would generate, independently of the number of SPEs, exactly the same memory accesses as the naive scheduler for a single SPE, so we can compare with the results from Section 6.3 here.

For more than one SPE, the feedback of locality information to the scheduler suffers an additional delay. An execution with a single SPE as in Sections 6.4 and 6.3 executes a tight cycle in which a bundle is scheduled and the locality information is fed back. In contrast, with e.g. 2 SPEs, when the scheduler constructs a bundle for SPE 1, it is very likely that SPE 2 in the meanwhile is processing a bundle of its own.

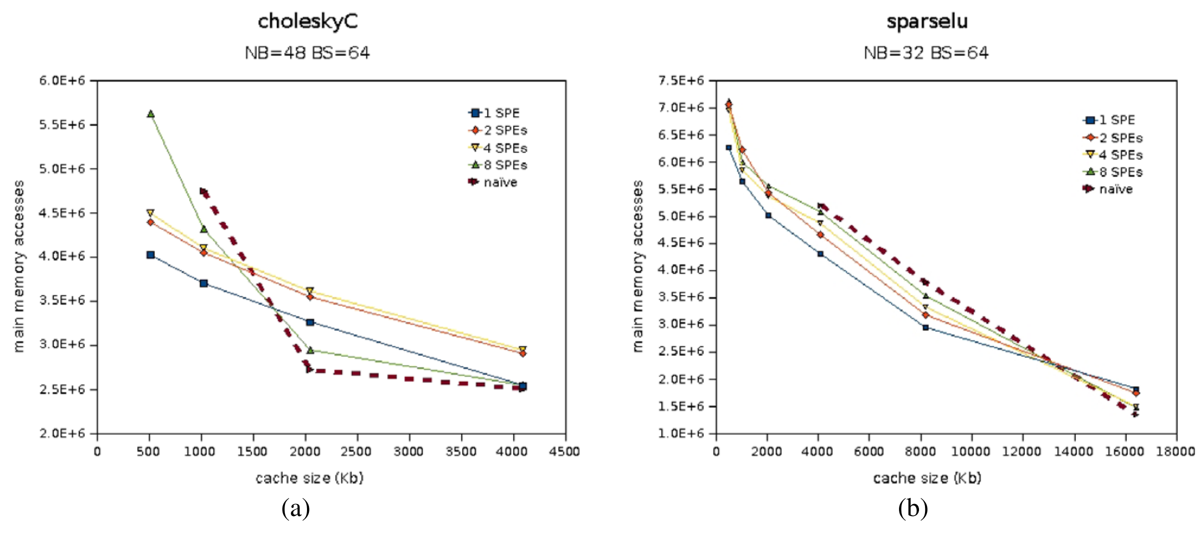


Fig. 20. Temporal locality for *choleskyC* and *sparselu* with more than one SPE. (a) Locality in the SL1-cache for *choleskyC* with more than one SPE, (b) locality in the SL1-cache for *sparselu* with more than one SPE.

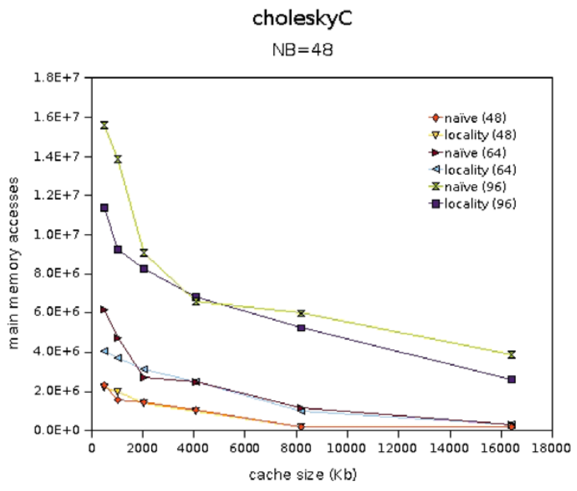
Therefore only partial locality information for this bundle will have been flushed to the PPE runtime, and the schedule for SPE 1 will consider only part of the actual working set. Furthermore, unless the computation can be restructured to have good temporal locality, more SPEs simply imply a larger working set. These observations suggest that the hit rate for a small cache will inevitably decrease as the number of SPEs increases. The best one can hope for is that the scheduler manages to detect enough reuse to benefit from larger cache sizes.

Figure 20(a) confirms this tendency for *choleskyC*. Even for cache sizes larger than 1024 kB an increase in the memory accesses accompanies the increasing number of SPEs, except for the case of 8 SPEs. For 8 SPEs, an interesting phenomenon occurs. The reuse of objects in the SL1-cache is far better, more accesses hit in the cache and reduce the number of main memory accesses for larger cache sizes. For less than 8 SPEs, the reuse is worse than compared to a single SPE, and the number of accesses to main memory increases. For *sparselu*, the conclusion for smaller cache sizes is confirmed (Fig. 20(b)), although the locality improvement as observed for *choleskyC* for larger cache sizes is not present here. Despite the slight increase in memory accesses, the locality scheduler still manages to improve on a naive scheduler by almost 8% for cache sizes between 2048 and 8192 kB. For *choleskyC* and a cache size of 2048 kB, the locality scheduler closes the gap with the naive scheduler, compared to Fig. 15(a).

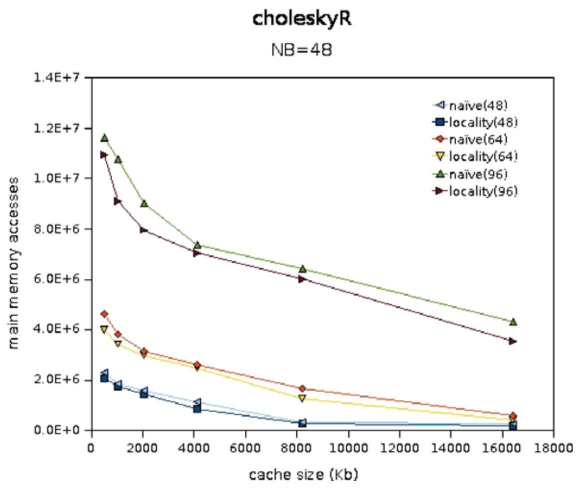
Finally, a naive scheduler by definition only includes ready tasks in its schedules, and thus is bound by the width of parallelism of the application, while the locality scheduler is better at finding schedulable tasks. Locality is only one side of the picture, and although it is very interesting to reduce the number of main memory accesses, the requirement to keep all the SPEs from idling should not be ignored.

6.6. *Impact of the block size*

The SPEs are vector processors. Conceptually, a computation inside an SPE repeatedly maps data sets to vector registers (via vector loads and stores) and performs vector operations. The size of those data sets, or the block size, together with the time complexity of the code, determines the execution time. The larger the time spent in a computation, the more opportunity to overlap computation and communication. Due to the small size of the LS in the Cell/B.E., it is the communication (and not the computation) that bottlenecks the execution, and a larger block size and thus a larger LS are preferable. There is another argument in favor of expanding the LS. Because of the characteristics of an SPE, main memory tends to be accessed in blocks, rather than in non-adjacent cachelines. This results in relatively more cache hits as the block size grows (for example, for *choleskyR* in Fig. 21(b)) as long as the SL1-cache is large enough so that the capacity misses for larger block sizes are compensated by the increase in hits. In Fig. 21(a), we see the same relative increase in cache hits for *choleskyC*. Note that the point for which the locality scheduler performs worse than the naive scheduler moves from $x = 1024$ for $BS = 48$ to $x = 2048$ for $BS = 64$ to $x = 4096$ for $BS = 96$.



(a)



(b)

Fig. 21. Temporal locality for *choleskyC* and *choleskyR* with varying block size. (a) Locality in the SL1-cache for *choleskyC*, (b) locality in the SL1-cache for *choleskyR*.

7. Conclusion

We have presented a linear-time scheduling algorithm for the Cell/B.E. that schedules bundles of tasks and is dynamical in the strong sense: it schedules tasks at runtime and is able to operate on a partial task dependence graph. The latter characteristic increases the complexity of the scheduling problem to the point where we cannot make claims about the makespan of an application anymore. Instead, we have opted for a design that avoids idling of the SPEs, while simultaneously trying to reduce the task execution time of these SPEs. A very lightweight scheduler tries to keep pace with the SPEs and reorganizes the TDG to detect temporal locality. Our measurements and simulations point out that a simple feedback mechanism, together with an equally straightforward graph transformation, have the potential to significantly increase the temporal locality of an application on the Cell/B.E. platform. The SPEs benefit from the temporal locality of the scheduled tasks, and succeed in reducing the makespan of the application. This software is available for download at: http://www.bsc.es/cellsuperscalar.

8. Future work

Our current efforts focus on reducing the overhead of the scheduler. As demonstrated in Section 6, we are able to extract temporal locality from an application. But the lack of optimized code for the scheduler currently causes this improvement not to carry through to the application execution time. We hope to solve this issue by hand-crafting optimized PowerPC code for the scheduler, reducing the contention between the PPE threads, and using light-weight data structures wherever possible.

Furthermore, we are implementing a *lazy renaming* policy, that detects temporary renamings, and prevents stale buffers to be copied from and back to main memory. On the SPE-side, *short-circuiting* can be used to reduce main memory accesses. Short-circuiting attempts to keep the task arguments in the LS as long as possible or necessary. An SPE then tries to fetch its arguments from the other SPE's stores, instead of from main memory. Ideally, arguments will no longer be copied back to main memory, unless the program semantics require so.

We are also looking at a method to analyze and further understand the impact of the feedback mechanism and co-parent edges on the temporal locality of a CellSs execution. These simple ideas look promising, but from the locality experiments in Section 6 we can see there is still room for improvement. It would also be interesting to evaluate the effect of both separately.

Acknowledgment

The authors acknowledge the financial support of the Commission Interministerial de Ciencia y Tecnologa (CICYT, Contract TIN2007-60625), the BSC-IBM MareIncognito research agreement and the Programme AlBan, the European Union Programme of

High Level Scholarships for Latin America, scholarship No. E05D058240CO.

References

[1] T.W. Ainsworth and T.M. Pinkston, On characterizing performance of the Cell Broadband Engine element interconnect bus, in: *Proceedings of the First International Symposium on Networks-on-Chip*, Princeton, NJ, 2007.

[2] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson et al., The landscape of parallel computing research: A view from Berkeley, Technical Report EECS-2006-183, University of California at Berkeley, 2006.

[3] N.C. Audsley, Deadline monotonic scheduling, Technical Report YCS 146, Department of Computer Science, University of York, October 1990.

[4] N.C. Audsley, Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, Technical report, Department of Computer Science, University of York, 1991.

[5] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin and G. Zhang, A proposal for task parallelism in OpenMP, in: *Proceedings of the 3rd International Workshop on OpenMP*, Reims, France, June 2006.

[6] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang and K. O'brien, A novel asynchronous software cache implementation for the Cell-BE processor, in: *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, Urbana, IL, 2007.

[7] P. Bellens, J.M. Perez, R.M. Badia and J. Labarta, CellSs: A programming model for the Cell BE architecture, in: *Proceedings of the ACM/IEEE SC 2006 Conference*, Tampa, FL, November 2006.

[8] E. Chan, E.S. Quintana-Orti, G. Quintana-Orti and R. van de Geijn, Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures, in: *Proceedings of the 19th Annual ACM symposium on Parallel Algorithms and Architectures*, San Diego, CA, 2007, pp. 116–125.

[9] L.F. Chao and E. Sha, Scheduling data-flow graphs via retiming and unfoloding, *IEEE Transactions on Parallel and Distributed Systems* **8**(12) (1997), 1259–1267.

[10] H. Chen, B. Shirazi and J. Marquis, Performance evaluation of a novel scheduling method: Linear clustering with task duplication, in: *Proceedings of the 2nd International Conference on Parallel and Distributed Systems*, Taiwan, December 1993.

[11] P. Chretienne, Task scheduling over distributed memory machines, in: *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Chateau De Bonas, Gers, France, 1989.

[12] E. Coffman and R. Graham, Optimal scheduling for two-processor systems, *Acta Informatica* **1** (1972), 200–213.

[13] M. Cosnard and E. Jeannot, Compact DAG representation and its dynamic scheduling, *Journal of Parallel and Distributed Computing* **58**(3) (1999), 487–514.

[14] A. Duran, J.M. Perez, E. Ayguade, R.M. Badia and J. Labarta, Extending the OpenMP tasking model to allow dependent tasks, in: *Proceedings of the 4th International Workshop on OpenMP*, Purdue University, West Lafayette, IN, 2008.

[15] M. Frigo, C.E. Leiserson and K.H. Randall, The implementation of the Cilk-5 multithreaded language, *SIGPLAN Notices* **33**(5) (1998), 212–223.

[16] A. Gerasoulis and T. Yang, On the granularity and clustering of directed acyclic task graphs, *IEEE Transactions on Parallel and Distributed Systems* **4**(6) (1993), 686–701.

[17] P.B. Gibbons and S. Muchnick, Efficient instruction scheduling for a pipelined architecture, in: *Proceedings of the SIGPLAN Symposium on Compiler construction*, Palo Alto, CA, 1986.

[18] M. Gonzalez, E. Ayguadé, X. Martorell and J. Labarta, Exploiting pipelined executions in OpenMP, in: *Proceedings of the 32nd Annual International Conference on Parallel Processing*, Kaohsiung, Taiwan, October 2003, pp. 153–160.

[19] B. Hamidzadeh, L.Y. Kit and D.J. Lilja, Dynamic task scheduling using online optimization, *IEEE Transactions on Parallel and Distributed Systems* **11**(11) (2000), 1151–1163.

[20] B. Hamidzadeh and D.J. Lilja, Dynamic scheduling strategies for shared-memory multiprocessors, in: *Proceedings of the International Conference on Distributed Computing Systems*, Hong-Kong, 1996.

[21] T. Hu, Parallel sequencing and assemby line problems, *Operation Research* **9** (1961), 841–848.

[22] IBM, Cell Broadband Engine Architecture, version 1.02, IBM Technical Document.

[23] IBM, Cell Broadband Engine Programming Handbook, version 1.1, International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, 2007.

[24] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure and M. Wolfe, Dependence graphs and compiler optimizations, in: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Williamsburg, VA, 1981, pp. 207–218.

[25] Y. Kwok and I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, *ACM Computing Surveys* **9**(4) (1999), 406–471.

[26] J. Labarta, S. Girona, V. Pillet, T. Cortes and L. Gregoris, DiP: A parallel program development environment, in: *Proceedings of the 2nd International EuroPar Conference (EuroPar'96)*, Lyon, France, 1996.

[27] C. Liu and J. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, *Journal of the ACM* **20**(1) (1973), 46–61.

[28] P. Liu, J.-J. Wu and C.-H. Yang, Locality-preserving dynamic load balancing for data-parallel applications on distributed-memory multiprocessors, *Journal of Information Science and Engineering* **18**(6) (2002), 1037–1048.

[29] G. Manimaran and C.S.R. Murthy, An efficient dynamic scheduling algorithm for multiprocessor real-time systems, *IEEE Transactions on Parallel and Distributed Systems* **9**(3) (1998), 312–319.

[30] R.R. Muntz and E.G. Coffman, Preemptive scheduling of real-time tasks on multiprocessor systems, *Journal of the ACM* **17**(2) (1970), 324–338.

[31] P.K. Murthy and E. Lee, On the optimal blocking factor for blocked, non-overlapped schedules, Memo No. UCB/ERL M94/46, Electronics Research Lab., University of California, Berkeley, CA, 1994.

[32] J. Noguera and R.M. Badia, Dynamic run-time HW/SW scheduling techniques for reconfigurable architectures, in: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, Estes Park, CO, 2002.

[33] M.A. Palis and J. Liou, Task clustering and scheduling for distributed memory parallel architectures, *IEEE Transactions on Parallel and Distributed Systems* **7**(1) (1996), 46–55.

[34] C. Papadimitriou and M. Yannakakis, Scheduling interval-ordered tasks, *SIAM Journal of Computing* **8**(3) (1979), 405–409.

[35] K. Parhi and L. Lucke, Data-flow transformations for critical path time reduction in high-level DSP synthesis, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and System* **12**(7) (1993), 1063–1068.

[36] K. Parhi and D. Messerschmitt, Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding, *IEEE Transactions on Computers* **40**(2) (1991), 178–195.

[37] J.M. Perez, P. Bellens, R.M. Badia and J. Labarta, CellSs: Programming the Cell/B.E. made easier, *IBM Journal of R&D* **51**(5) (2007), 593–604.

[38] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, MIT Press, Cambridge, MA, USA, 1989.

[39] H. Servat, C. Gonzalez-Alvarez, X. Aguilar, D. Cabrera-Benitez and D. Jimenez-Gonzalez, Drug design on the Cell Broadband Engine, in: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, 2007.

[40] H. Servat, C. Gonzalez-Alvarez, X. Aguilar, D. Cabrera-Benitez and D. Jimenez-Gonzalez, Drug design issues on the Cell BE, in: *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers*, Paphos, Cyprus, 2008.

[41] The community of OpenMP users, researchers, tool developers and provider website, http://www.compunity.org/.

[42] R.M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM Journal of Research and Development* **11**(1) (1967), 25–33.

[43] T. Yang and A. Gerasoulis, A fast static scheduling algorithm for DAGs on an unbounded number of processors, in: *Proceedings of the ACM/IEEE Conference on Supercomputing*, Albuquerque, NM, 1991, pp. 633–642.

[44] W. Zhao, K. Ramamritham and J.A. Stanovic, Preemptive scheduling under time and resource constraints, *IEEE Transactions on Computers* **36**(8) (1987), 949–960.

[45] H.B. Zhou, Scheduling DAGs on a bounded number of processors, in: *Proceedings of PDPTA*, Sunnyvale, CA, Vol. 2, August 1996, pp. 823–834.