Scientific Programming 17 (2009) 59–76
DOI 10.3233/SPR-2009-0269
IOS Press

Available task-level parallelism on the Cell BE

Alejandro Rico [a,*], Alex Ramirez [a,b] and Mateo Valero [a,b]
[a] *Universitat Politecnica de Catalunya, Barcelona, Spain*
[b] *Barcelona Supercomputing Center, Barcelona, Spain*

Abstract. There is a clear industrial trend towards chip multiprocessors (CMP) as the most power efficient way of further increasing performance. Heterogeneous CMP architectures take one more step along this power efficiency trend by using multiple types of processors, tailored to the workloads they will execute. Programming these CMP architectures has been identified as one of the main challenges in the near future, and programming heterogeneous systems is even more challenging. High-level programming models which allow the programmer to identify parallel tasks, and the runtime management of the inter-task dependencies, have been identified as a suitable model for programming such heterogeneous CMP architectures.

In this paper we analyze the performance of Cell Superscalar, a task-based programming model for the Cell Broadband Engine Architecture, in terms of its scalability to higher number of on-chip processors. Our results show that the low performance of the PPE component limits the scalability of some applications to less than 16 processors. Since the PPE has been identified as the limiting element, we perform a set of simulation studies evaluating the impact of out-of-order execution, branch prediction and larger caches on the task management overhead.

We conclude that out-of-order execution is a very desirable feature, since it increases task management performance by 50%. We also identify memory latency as a fundamental aspect in performance, while the working set is not that large. We expect a significant performance impact if task management would run using a fast private memory to store the task dependency graph instead of relying on the cache hierarchy.

Keywords: Scalability, multicore, task-based programming models, Cell BE

1. Introduction

Power consumption and design complexity have led the computer architecture community to design chip multiprocessors (CMP). Current commercial CMP integrate 4–8 processors in one chip, but the current interpretation of Moore's Law says that the number of cores will double every 18 months, leading to hundreds or even thousands of cores per chip in the near future [2]. In order to exploit the performance potential of these architectures, huge amounts of parallelism need to be exploited. Supercomputing class applications efficiently exploit thousands of processors to run applications coded using low-level programming models, like MPI. However, explicit data distribution and communication is not the desirable programming model for future multicore architectures.

The increasing hardware complexity, and the matching software complexity, will force programmers to use higher level programming models. The use of *tasks* as high level abstraction, and the runtime detection of task parallelism is becoming widely accepted into mainstream programming models like OpenMP 3.0 [24]. Other example implementations of such task level parallelism are Thread Building Blocks (TBB) [27], a parallel programming model by Intel that encapsulates tasks in a C++ class so as to represent parallel computation and Cilk [5], with specific keywords to spawn independent tasks when calling functions. There are also pure task-based parallel programming models such as Cell Superscalar [4] and Tagged Procedure Calls (TPC) [18]. In all these models, the task concept provides an intuitive abstraction that can be directly mapped to processing units since it encapsulates not only computation but also its working data set.

In this paper, we analyze the behavior of the Cell Superscalar task-based programming model on the Cell Broadband Engine (Cell BE), a state-of-the-art CMP

[*]Corresponding author: Alejandro Rico, Universitat Politecnica de Catalunya, Jordi Girona 1-3, D6-113, 08034 Barcelona, Spain. Tel.: +34 93 40 54097; E-mail: arico@ac.upc.edu.

1058-9244/09/$17.00 © 2009 – IOS Press and the authors. All rights reserved

with distributed on-chip memories. Our results show that the speed at which tasks are generated and managed is an intrinsic limitation to the scalability of task-based programming models. We obtain experimental data to compute how far Cell Superscalar would scale on the current Cell BE chip, and perform a simulation study looking for ways to improve such scalability.

We consider multiprocessor systems to be scalable if there is a linear relationship between the number of cores and the speed-up with respect to the sequential (single-processor) version [13]. As the number of processing elements increases, a scalable system will take less time to solve a problem of a fixed size, or will be able to solve a larger problem in the same amount of time.

In a task-based scenario, the problem size of an application is the number of tasks times the task size (Eq. (1)). Here, the size of a task is the size of its working data set (task problem size). Increasing the problem size means either increasing the size of the tasks, or increasing the number of tasks. The scalability of the system depends on its ability to keep all the processing elements computing such tasks in parallel.

$$\text{Problem size} = \text{number of tasks} \times \text{task size}. \quad (1)$$

Task-based programming models split an application into two types of threads: the *master* thread and a set of *worker* threads. The master thread runs through the application sequentially and spawns tasks, inserting them in the work queue. The worker threads take tasks from the work queue and execute them. Figure 1 shows an example task-based application on an 8-core multiprocessor. In this example, the master thread is executing and is exclusively dedicated to task generation (TG). Each TG takes 1 time unit (t.u.). Thus, the master thread generates one task every time unit which is shown as the initiation interval. Tasks are assigned to the first free worker and are executed in 5 t.u. Worker 1 executes task 1, worker 2 executes task 2, and so on. However, when task 6 is ready to be dispatched, worker 1 has finished executing task 1 and is available to take task 6 instead of worker 6. This situation repeats for tasks 11, 16, and further, leaving workers 6 and 7 idle for the whole execution.

Our example does not consider thread communication delays, and assumes a master thread exclusively generating tasks. Under these assumptions, the number of parallel active tasks is the task execution time divided by the task generation time (Eq. (2)). Task execution time depends on the working set (task size), but also on the type of computation. Some applications (such as sparse linear algebra) do not use all the data on the working set, so enlarging the task size has a smaller impact. Despite of this fact, the task execution time is proportional to the task size.

$$\text{Max. active tasks} = \left\lceil \frac{\text{task execution time}}{\text{task generation time}} \right\rceil,$$
$$\text{task execution time} \propto \text{task size}. \quad (2)$$

The result of Eq. (2) is the maximum number of active tasks. Any additional overhead due to communications, or extra computation for task management, would decrease the maximum parallelism. In a scenario where an application has to use all the processors in the system, the maximum number of parallel tasks should be greater or equal to the number of processors. Otherwise, a number of processors will be idle.

In order to increase the available parallelism, one has to increase the task size. For larger problems, it

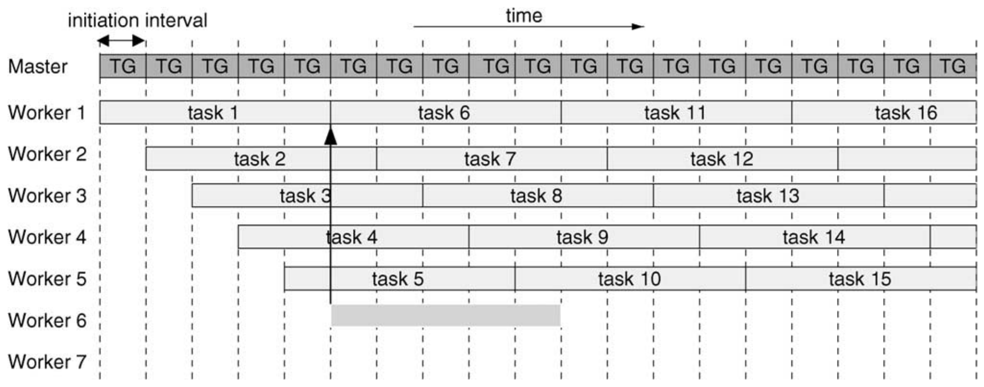


Fig. 1. Task distribution among processing elements on an 8-core multiprocessor. Master thread continuously generates tasks. Workers 6 and 7 remain idle during the whole execution. The initiation interval is the task generation (TG) time.

is possible to split the problem in a fixed number of tasks, and so increase task size. However, for fixed-size problems, increasing the task size also means reducing the number of available tasks, and so the available parallelism. Furthermore, some algorithms already define the natural size of a task. For example, multimedia applications work with fixed size data elements. A PAL-DVD frame is composed of 720 × 576 pixels, and encoding–decoding algorithms work on 16 × 16 pixel macroblocks. Hardware specifications could also be a limiting factor for the task size. Architectures using local memories, such as the Cell BE or several embedded platforms, limit the task size to what can be fit in such local memory. Architectures relying on caches may expand the task size, but still rely on temporal locality and the cache size.

The Cell BE [17] is a 9-core heterogeneous multiprocessor with a general-purpose processor and 8 accelerators and the first implementation of the Cell Broadband Engine Architecture (CBEA) [6]. The accelerators only operate on data located in their local memories. Therefore, enlarging the task size to achieve more parallelism is not a possible solution for CBEA-compliant processors. This situation sets our focus on decreasing the task generation overhead which becomes the critical factor regarding scalability and full resource utilization.

This article presents an analysis of the achievable parallelism of the Cell BE running Cell Superscalar [4] applications. This analysis is performed using high performance scientific applications and validated with a synthetic application (Section 2). Section 3 presents a characterization of the task generation phase of these applications. This study shows the features of the task generation code and the possible strategies to speed it up. Section 4 presents existing works about scalability analysis, task-based execution environments and potential scenarios for processor underutilization. The conclusions are exposed in Section 5.

2. Scalability on the Cell BE

As it is later explained in Section 4, there are several attempts in the literature to provide a universal definition for scalability, but none of them has been successful. In this paper, we assume the intuitive definition for which a parallel system is scalable if performance improves as the problem size and the number of processors increase.

In this section, we present a scalability study of applications written using the Cell Superscalar programming model [4] on the Cell BE. First, we present the execution environment, and the performance analysis methodology. Next, we measure the achievable parallelism for a set of scientific application kernels. Finally, we validate our preliminary conclusions using a synthetic application where we can tune the task generation and task execution costs.

The Cell BE is a joint initiative of Sony, Toshiba and IBM (STI) and the first implementation of the Cell Broadband Engine Architecture (CBEA) [6]. It is composed of a general-purpose processor named PowerPC Processor Element (PPE) and eight SIMD accelerators: the Synergistic Processor Elements (SPE). The PPE is a 64-bit PowerPC-compliant processor with in-order execution and two-way simultaneous multithreading support. It integrates a vector multimedia extension (VMX) unit, 32-kB level 1 instruction and data caches and a 512-kB level 2 cache. Each SPE is composed of a Synergistic Processor Unit (SPU), a 256-kB local memory named Local Store (LS) and a Memory Flow Controller (MFC). A SPU is an in-order, dual-issue, SIMD-ISA processor with 128 registers of 128 bits. The LS is shared for both instructions and data, and transfers between the LS and main memory (or other LS) are performed through the DMA engine incorporated in the MFC.

The Cell Superscalar (CellSs) programming model is a task-based programming model for the CBEA. A CellSs program is a sequential C code with OpenMP-like annotations on functions. Annotated functions are specified as parallel tasks to be executed on the SPEs. Task parameters are defined as read-only, write-only or read–write data to allow the CellSs runtime to track the dependencies among SPE tasks before their dispatch to ensure correctness. CellSs applications execute two threads on the PPE (master and helper) and one thread on each SPE (workers). The entry point of a CellSs program is executed by the master thread which starts running the user code. When an annotated function is called, a new SPE task is created and added to the dependence graph. Periodically, the helper thread checks the dependence graph for available tasks. If there are enough tasks, they are bundled together, scheduled to satisfy dependencies, and dispatched to the first available SPE. On bundle execution finalization, the SPE notifies it to the PPE, where the master and helper threads remove the finished tasks from the dependence graph.

2.1. Methodology

The scalability analysis in this section has been performed on IBM QS21 Blades, with 2 Cell BE running at 3.2 GHz and 512 MB of XDR memory. The applications have been written and compiled for the CellSs v1.4 runtime environment [8]. The CellSs runtime library is instrumented to provide timing information and detailed performance analysis traces. After program execution, we obtain a time annotated trace of the different execution phases of all the application threads, including master, helper and all worker threads. The trace is analyzed using Paraver [25], a visualization tool that allows graphically representing the execution phases and thread communications of multithreaded programs.

Time measurements have been made for the task generation cost on the PPE and the task execution time on SPEs. As previously mentioned, the master thread creates tasks and adds them to the dependence graph, while the helper thread schedules tasks, groups them in bundles and dispatches them to the SPEs. Since these two phases are executed in parallel on the PPE, the task generation cost is the maximum of the master thread and helper thread parts. The task creation phase on the master thread is performed for each task, while the task scheduling, grouping and dispatching phases on the helper are executed for each bundle. Our measurements show that the task generation time on the helper thread is always much lower than the time of creating and adding the task to the graph on the master thread. Hence, from now on, *task generation cost* refers only to the master thread part, since it is always the maximum of the two.

In order to calculate the maximum number of active tasks for each program, the durations of the task generation and task execution phases have been measured for each program. The task generation and task execution times for an application are the average over all task generation and task execution instances throughout the whole execution. Misbehaviours due to operating system activity or other external agents (like interrupts or context switches) may condition the results of averaging. Some phase durations may seem larger than they actually were because of execution interference. These data are not representative and have been considered *outliers*. Since the distribution of phase time values resembles a Gauss curve, an outliers removal algorithm can be applied in order to use only representative data for averaging. Equation (3) shows the limits for extreme outliers deletion [29]. The lowest representative value is the lower quartile ($Q1$) minus 3 times the interquartile range (IQR) and the highest valid value is the upper quartile ($Q3$) plus 3 times the interquartile range (IQR):

$$\begin{aligned}
&\mathrm{IQR} = Q3 - Q1,\\
&\quad Q1 = \text{lower quartile},\\
&\quad Q3 = \text{upper quartile},\\
&\text{Lower outlier limit} = Q1 - 3 * \mathrm{IQR},\\
&\text{Upper outlier limit} = Q3 + 3 * \mathrm{IQR}.
\end{aligned}\tag{3}$$

Since the operating system only runs on the PPE, task execution on the SPEs has a negligible variability. However, task generation cost could suffer severe misbehaviours. This leads to the situation where the average of all measured times always shows a task generation cost higher than the average without outliers. On the other hand, task execution remains almost invariable. Therefore, considering all measured times for task generation would lead to much lower achievable parallelism results when applying Eq. (2), which is not representative of the mean behaviour of the program. For this reason, all averages of execution phases presented in this paper have been calculated after deleting outliers.

Nevertheless, since measurements are performed in a real system, results may significantly vary from one execution to another in spite of outliers deletion. Hence, the results presented for an application are the average over 10 executions of the same program.

2.2. Scientific applications

The first part of the scalability analysis consists on the measurement of the task generation and task execution times for a set of high performance scientific applications. These applications are written and compiled with CellSs v1.4 [8]. This set is composed by the following programs: Cholesky factorization, LU decomposition, Jacobi, matrix transposition and matrix multiplication. For the matrix multiplication there are 5 versions with different levels of optimization: non-vectorized, vectorized 1, vectorized 2, vectorized 3 and one using the SPU code of the matrix multiplication in IBM's SDK [6]. The non-vectorized version is the usual three loop implementation using scalar code. Vectorized 1 performs the computation with vector instructions for *multiply and add*. Vectorized 2 uses vector instructions and unrolls the innermost loop. Vectorized 3 performs as vectorized 2 and also unrolls the second level loop. The SDK matrix multiplication code for the SPE is a fully unrolled, vectorized and scheduled version achieving over 90% efficiency on the SPU.

As discussed in Section 1, the size of SPE tasks is limited by the LS size. For this analysis, the working data set for each application has been set to the maximum that fits the LS in order to study the maximum task size. Table 1 shows the task data size for each application. Each bundle of tasks in matrix multiplication works on a matrix block which is independent from the rest. Thus, all tasks are independent and parallelism is not limited by data dependencies. Both Cholesky factorization and LU decomposition algorithms have inter-task dependencies that further restrict the available parallelism. The Jacobi algorithm works as a sequential scan of the matrix rows, making all tasks depend on the previous one, removing all possible parallelism.

For the purpose of this study, we will use the task generation and task execution cost of these algorithms in order to compute the maximum available parallelism that could be achieved by a fully-parallel application having the same costs. We consider those measurements to be a better indication than wild-guessing on the costs of random fully parallel applications.

Table 1

Task working set of the analyzed high performance scientific applications

Application	Task data size
Cholesky	3 blocks, 64 × 64 float
LU decomposition	2 blocks, 64 × 64 float
Jacobi	3 rows, 8192 float
matrix transpose	2 blocks, 64 × 64 float
matmul novec	3 blocks, 64 × 64 float
matmul vec1	3 blocks, 64 × 64 float
matmul vec2	3 blocks, 64 × 64 float
matmul vec3	3 blocks, 64 × 64 float
matmul spu sdk	3 blocks, 64 × 64 float

All applications have been executed 10 times. Task generation and task execution durations have been measured on each program so as to compute the maximum number of parallel active tasks (Eq. (2)). This calculation results in 10 values, one for each execution of the program. The final number of maximum parallel active tasks for an application is the average over the 10 executions after removing outliers for each as mentioned before. Figure 2 shows the results of these measurements. The dotted line marks the required number of active tasks to fully utilize a 64-core multiprocessor (this is the expected by Moore's Law by 2013). As shown in the graph, only the non-vectorized and vectorized 1 versions of the matrix multiplication and LU decomposition could achieve more than 64 active tasks.

Matrix multiplication benchmarks show the impact of task optimization when task size is limited. Optimized task code requires less execution time while task generation overhead is the same. This leads to the situation where the non-optimized version of matrix multiplication provides a high amount of parallelism (up to 637 parallel tasks) but, as task code is optimized, the achievable parallelism decreases. The vectorized 2 version (vectorization and innermost loop unrolling) achieves a maximum of 44 active tasks and the fully unrolled version in the SDK only 9. This is also the case for Cholesky, which is an optimized version for the SPE and only achieves 9 concurrent tasks. Likewise SDK matrix multiplication and Cholesky, matrix transposition exploits enough parallelism to use all the resources on a single Cell BE. However, two Cell BE chips can be connected through a coherent I/O controller which allows sharing their SPEs. In this case, none of these applications would be able to feed all 16 SPEs. On the other hand, LU decomposition and Jacobi are able to exploit enough parallelism for the Cell

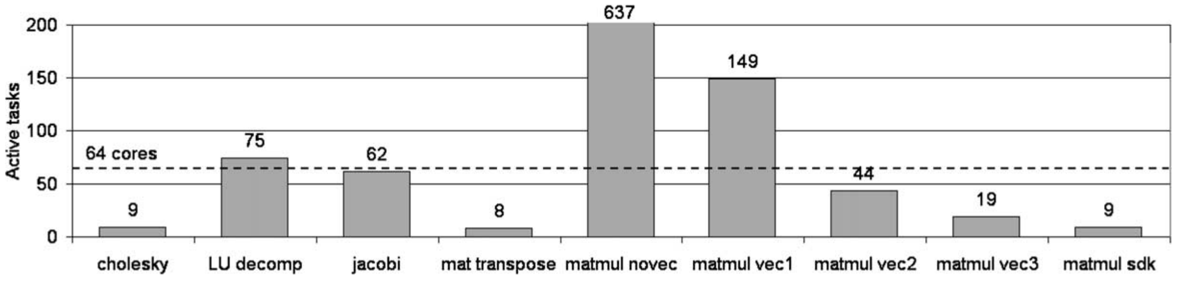


Fig. 2. Maximum active tasks for CellSs high performance scientific applications on the Cell BE. Only non-fully optimized versions of matrix multiplication and LU decomposition achieve more than 64 active tasks.

BE. However, these two programs are not optimized so the maximum number of active tasks could drop when applying specific optimization.

Our results show that current applications would need to enlarge the task size in order to exploit all the available parallelism in the Cell BE. However, the size of the Local Store prevents such task size enlargement. Other CMP implementations relying on caches do not have such a hard limitation, however, task performance is still strongly dependent on the working set fitting in the cache hierarchy. Therefore, task size can not keep growing indefinitely since it will be limited by the memory system. If task execution can not be increased further, then we must reduce the task creation overhead for next generation CMP architectures to be able to exploit all the available parallelism.

2.3. *Synthetic application*

In this section a synthetic application is used to validate the methodology proposed in Section 2.1 for the computation of the maximum number of parallel tasks. The synthetic application is tuned so that it does not use all the SPEs in order to check the expected underutilization. It is also used to study the CellSs task generation variability in different scenarios. CellSs master thread code creates tasks and adds them to the dependence graph in the css_addTask function. In this phase, it creates the suitable data structures for task management and performs the suitable modifications on the tasks dependence graph. These modifications are performed once for each task parameter since dependencies among tasks are due to data hazards on their parameters. Thus, the number of task parameters determines the task generation cost so the variability study is performed with several numbers of task parameters.

Figure 3 shows a simplified version of the code of the synthetic application. The master thread executes the main *while* loop. Each iteration spawns a task with the specified number of parameters, so the number of iterations of the master thread loop is the total number of tasks in the application. Spawned tasks are executed by the worker threads in the SPEs and all of them are independent. Task execution is essentially composed of a *for* loop that performs a dummy operation in each of its iterations.

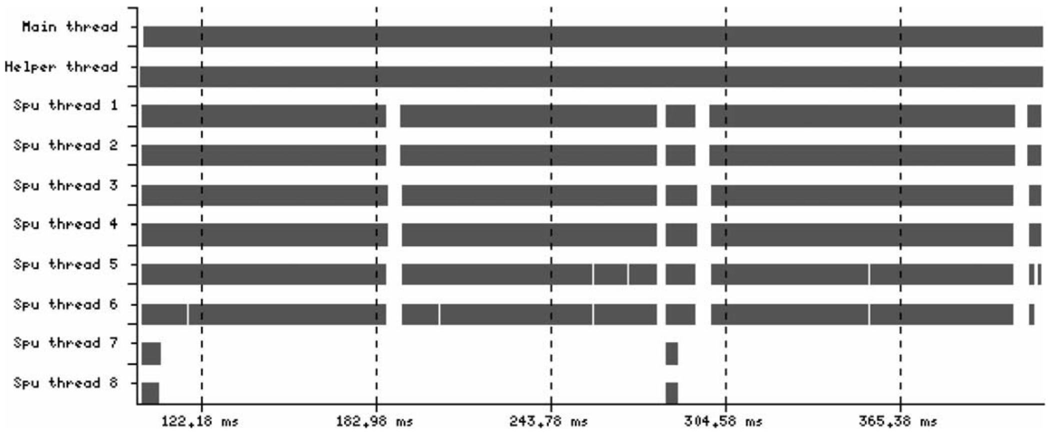
The synthetic application can be tuned in three different ways. First, task duration can be adjusted by specifying the number of times the task repeats its computation (task loop iterations). This allows varying task size in order to achieve different levels of parallelism. The synthetic application also allows modifying the number of task parameters so as to see their impact on task generation cost (task parameters). Finally, the number of total tasks can be also established so as to manipulate the program execution time (total tasks). This allows avoiding too short executions where initialization effects can affect master thread execution. The total tasks parameter has been set to 32,768 for all the experiments in this section which is enough to achieve a steady execution state.

2.3.1. *Methodology validation*

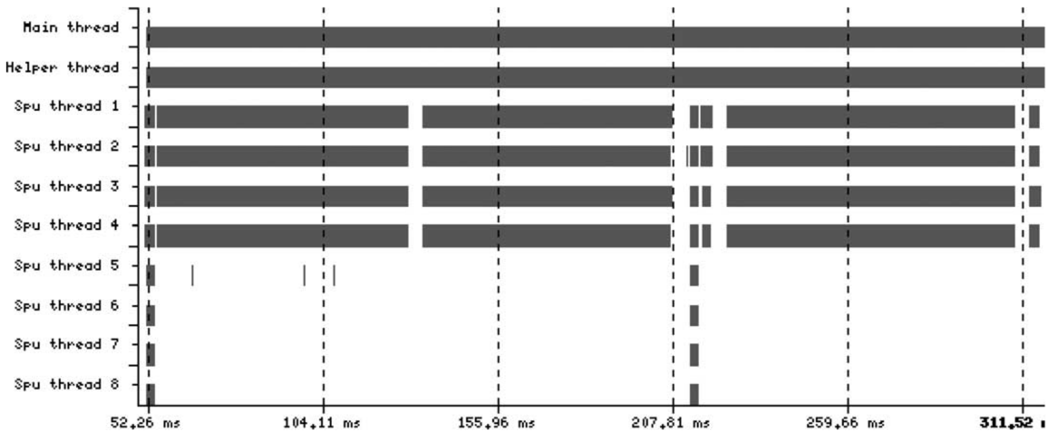
The first experiment is to perform executions with small task sizes so as to see the processor underutilization effect on the Cell BE. This allows validating the presented methodology for measuring achievable parallelism. As an example, the synthetic application has been run with 128 task loop iterations, each task receiving 4 parameters. The measured average task generation time is 5.96 µs while average task execution time is 34.46 µs. Applying Eq. (2) for computing the maximum number of active tasks we obtain $\lceil 5.78 \rceil = 6$ active tasks. Figure 4(a) shows the Paraver trace of the whole execution for 128 task loop iterations and 4 task parameters. X-axis is time and horizontal rows are execution on different threads. *Main thread* stands for the master thread while *Spu thread* is each one of the worker threads on the SPEs. The graph shows that 6 SPEs are continuously working (grey color), while Spu threads 7 and 8 are idle (white color) most of the time. Actually, workers 7 and 8 are waiting for tasks the 96.8% of the total time. Figure 4(b) shows another example: the execution for 32 task loop iterations and 8 task parameters. In this case, task generation lasts for 4.31 µs on average and task execution requires 15.4 µs. This results in $\lceil 3.58 \rceil = 4$ active tasks when applying Eq. (2). Workers 5, 6, 7 and 8 are idle the 99.2% of the time. These processing elements are not idle the whole execution for several reasons. CellSs is configured to wait until a specific number of tasks is added to the dependence graph before sending tasks to workers. Then, when the PPE starts dispatching tasks, there is a large number of ready tasks so it is able to feed all SPEs with the first burst. This also happens when operating system activity stalls the helper thread on the PPE, which does not allow it to dispatch tasks to SPEs. In this time, the SPEs have finished their work and the master has created more tasks so that all SPEs will be free and there would be a lot of ready tasks as well as in the situation of the first burst. Interconnection network contention may also delay a DMA transfer more than

```
//Includes and definitions

void work(vector float *block) {
        //dummy operation: adds and substracts 1 to all positions in
        //                 block using intrinsics
}
#pragma gss task input(it) inout(A)
void shortAddTask(int it, float A[SZ]) {
        int i; vector float *vecA = (vector float *)A;
        for(i=0;i<it;i++) {
                work(vecA);
        }
}
#pragma gss task input(it,B,C) inout(A)
void midAddTask(int it, float A[SZ], float B[MIN_SZ], float C[MIN_SZ]) {
        int i; vector float *vecA = (vector float *)A;
        for(i=0;i<it;i++) {
                work(vecA);
        }
}
#pragma gss task input(it,B,C,D,E,F,G) inout(A)
void longAddTask(int it, float A[SZ], float B[MIN_SZ], float C[MIN_SZ],
                float D[MIN_SZ], float E[MIN_SZ],float F[MIN_SZ], float G[MIN_SZ]) {
        int i; vector float *vecA = (vector float *)A;
        for(i=0;i<it;i++) {
                work(vecA);
        }
}
int main(int argc, char* argv[])
{
        //Declarations and initialization
        //ppe_it       -> main loop iterations (total tasks)
        //task_it      -> task loop iterations
        //add_task_len -> number of parameters modifier

        #pragma css start
        while(ppe_it>0) {
                ppe_it--;
                switch(add_task_len) {
                case 0: shortAddTask(task_it,mat_one[ppe_it]);
                        break;
                case 1: midAddTask(task_it,mat_one[ppe_it],victim1,victim2);
                        break;
                case 2: longAddTask(task_it,mat_one[ppe_it],victim1,victim2,
                                    victim3,victim4,victim5,victim6);
                        break;
                }
        }
        #pragma css finish

        //checks and error printing
        return 0;
}
```

Fig. 3. Simplified synthetic application source code.



(a)



(b)

Fig. 4. Synthetic application Paraver [25] traces. (a) Task loop iterations: 128. Task parameters: 4. The maximum active tasks is 6 and workers 7 and 8 are idle the 97% of the time. (b) Task loop iterations: 32. Task parameters: 8. The maximum active tasks is 4 and workers 5, 6, 7 and 8 are idle the 99% of the time.

expected. This prolongs the corresponding task duration which causes an SPE to be busy for a longer period, so a task that would be dispatched to it may go to an idle worker. These situations make idle processors to work, however, they seldom occur and only suppose work for the 1–5% of the total time.

The vertical white spaces in the trace are due to the trace flushing work on the master thread. The CellSs tracing engine buffers the occurred events in memory so as to minimize its intrusion on program execution with periodic disk accesses. When the events buffer is full, the master thread flushes it to disk. During this operation it does not generate tasks and SPEs do not have work which is shown as an idle period on the trace.

As a result, previous examples demonstrate that the presented methodology is useful for calculating achievable parallelism and predict processor underutilization. Despite the fact that processing elements that are supposed to be idle may infrequently perform tasks, they are unused more than the 90% of the time because the hardware is not capable of exploiting enough parallelism.

2.3.2. *Task generation cost variability*

The second aim of this study using the synthetic application is to measure the variability of the task generation phase on the master thread. As it is previously explained, task generation cost depends on the number of task parameters. Dependencies among tasks are due to hazards on the data ranges specified in their parameters, so the master thread has to update the dependence graph for each task parameter. Figure 5 shows the task generation time on the PPE for execution of tasks with different numbers of loop iterations (X-axis) and 2, 4 or 8 task parameters. Configurations with less than 128 task loop iterations do not

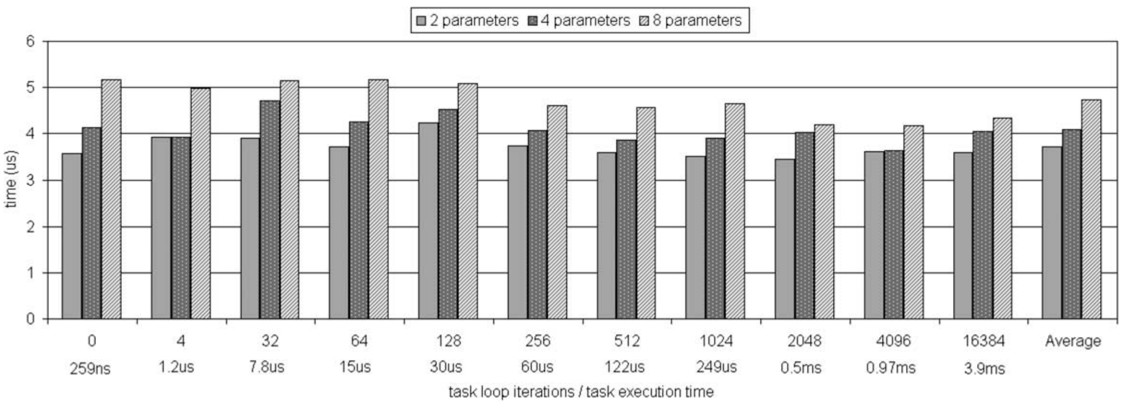


Fig. 5. Task generation time of the synthetic application with different task loop iterations and task parameters. On average, specifying 4 parameters is 11% more expensive than 2, and 8 parameters costs a 27% more than generating a task with 2 parameters.

achieve enough parallelism to feed all 8 SPEs. Hence, the master thread never waits for tasks finalization and is continuously generating tasks. Moreover, as task duration is short, DMA transfers (which are performed before and after task execution) occur more frequently and may cause interconnection network contention and cache conflicts. This affects the execution on the PPE, which is continuously performing the task generation phase. However, DMA transfers occur less often for larger tasks and may be overlapped with the *waiting for tasks finalization* phase on the master thread so the task generation phase is not interfered. This situation results in more expensive task generations for shorter tasks while, for larger ones, task generation is much less interfered by workers activity.

Average task generation times are 3.7 µs for tasks with 2 parameters, 4.1 µs for tasks with 4 parameters and 4.7 µs for tasks with 8 parameters. This generation costs require tasks with 2 parameters executing in at least 30 µs, tasks with 4 parameters executing in 33 µs or tasks with 8 parameters executing in 38 µs in order to fully utilize the Cell BE. Two Cell BE chips sharing their SPEs would require tasks during 60 (2 parameters), 66 (4 parameters) or 76 (8) µs.

3. Task generation analysis

As exposed in previous section, task-based applications could face a problem when trying to use all the processing units on future multiprocessor architectures because the hardware could not be able to exploit enough parallelism. Achievable parallelism depends on the task execution time and the cost of task generation. Since task execution time is proportional to the task size, and it is restricted by the characteristics of the memory system, task generation overhead becomes a critical factor. The task generation phase of CellSs is executed on the PPE. The PPE is quite limited in terms of superscalar performance, being only 2-wide and executing instructions in-order, making it *slow* compared to other current commercial general-purpose architectures.

This section analyzes the CellSs v1.4 task generation phase. This analysis focus on the features of the PPE that determine task generation performance and tries to find opportunities to speed it up.

3.1. Simulation setup

The task generation phase simulations have been carried out using the SMTSIM [30] simulator extended for PowerPC traces. The traces has been generated using the IBM BProber [15] instrumentation tool. This tool allows instrumenting PowerPC executable binaries by inserting function calls at specific locations in the code. CellSs task generation phase is completely performed in the master thread *css_addTask* function so only this function had to be instrumented for this study. Hence, traces are composed of the instructions of all css_addTask instances along program execution. The applications traced for this study are the same ones used in Section 2: Cholesky factorization, LU decomposition, Jacobi, matrix transposition and the five versions of matrix multiplication.

Table 2 shows the average percentage of instruction types in the generated traces. Almost half of the instructions perform integer arithmetic operations (49%) including integer multiplication and division (0.2%). Memory operations are 31%: 20% loads and 11%

Table 2
Percentage of instructions composing css_addTask traces

Arithmetic	Load	Store	Branch	Other
49%	20%	11%	17%	3%

stores. Branches are 17% of the total instructions: 13% conditional and 4% unconditional. The remaining 3% is mainly composed of special purpose register operations (2.6%), synchronization (0.33%) and cache management (0.07%) instructions. No floating point is carried out by css_addTask. These are average values for all traces but the mean is very representative because the variation among all applications is less than 1% for each instruction type.

Compared to SPEC-type applications, the ratio of load/store and branch instructions is very high. This is typical of control intensive code, as is the case of *css_addTask*. Given the high ratio of load and branch instructions, it is to be expected that cache size, latency, and branch prediction accuracy play a significant role in performance. There is still a high number of integer instructions, and our simulation study will also explore the amount of ILP available there, while evaluating the impact of wider superscalar issue.

SMTSIM has been extended to support in-order execution and has been configured to simulate a Cell BE PPE processor. Table 3 shows the parameters for several SMTSIM configurations we used. The column entitled in-*2* is the default PPE configuration with dual-issue in-order execution. The number of functional units has been enlarged for wider-issue configurations.

The following sections evaluate several architecture features and their impact on task generation execution using the SMTSIM configurations in Table 3.

3.2. Execution order

Most high-performance processors execute instructions out of order, in order to exploit more Instruction Level Parallelism (ILP), and to hide part of the memory latency (either in a cache hit or a cache miss). Out-of-order execution requires a number of complex, and power-hungry structures such as the issue queue and reorder buffer, plus larger register file for register renaming. In order to minimize its size, the PPE was designed for in-order execution [14]. This section evaluates the performance impact of wide superscalar and out-of-order execution on the CellSs task generation phase against the baseline dual-issue, in-order PPE.

Figure 6 shows the normalized IPC of wider superscalar configurations (with increased number of functional units to match), in-order and out-of-order with respect to the 2-wide in-order (PPE-like) configuration.

For in-order execution, increasing the superscalar issue width only has minimal impact. The 4-wide and 8-wide configurations are only 3% faster than the baseline. However, out-of-order execution does have an important effect on performance. The ooo-2 configuration, which represents an out-of-order PPE, achieves a 50% higher IPC. Increasing issue width and functional units has a higher impact for out-of-order configurations. The ooo-4 configuration achieves an additional 9% speedup, and the ooo-8 only a mere extra 1%. We must take into account that by deciding to run out-of-order, the processor may not be able to run at the same frequency as an in-order one, and so the 50% higher IPC may not fully translate into a 50% higher performance.

In addition, we must also consider that the task generation code has not been recompiled for each simulator configuration. Code compiled and scheduled for the wider issue configurations may have been able to exploit them better. This is specially meaningful for the 8-wide configurations, which may suffer from restricted fetch efficiency caused by the high number of taken branches, and has not been scheduled to take advantage of register renaming and higher number of functional units.

In spite of the specific-compilation issue, the results show that out-of-order execution is a very desirable hardware design for CellSs task generation performance. A 2/4-wide issue out-of-order design would decrease by 50–60% the task generation overhead which means exploiting a 50–60% more task-level parallelism.

3.3. Branch prediction

The PPE branch predictor is a 4 kB by 2-bit branch history table (BHT) with 6 bits of global history per thread, and it does not incorporate a branch target buffer (BTB) for target address prediction. In this section, we evaluate a wider set of branch predictors: the BHT [26], Gshare [22] and Perceptron [16] predictors, and an ideal (perfect) branch predictor. All of them are compared to the PPE branch predictor using the in-*2* and ooo-*2* configurations so as to see the impact of branch prediction not only on the PPE but also on an alternative out-of-order design. A BTB has also been added to simulations except for the perfect predictor which already performs both perfect branch and target prediction. The evaluated BTB has 512 entries

Table 3

SMTSIM parameter configurations

	in-2	in-4	in-8	ooo-2	ooo-4	ooo-8
Execution	In order			Out of order		
Fetch width	8 (4 each thread)					
Issue width	2	4	8	2	4	8
Func units						
Integer	2	4	8	2	4	8
Ld/St	1	2	4	1	2	4
FP	1	2	4	1	2	4
L1 ICache						
Size	32 kB					
Assoc.	2					
Latency	1					
L1 DCache						
Size	32 kB					
Assoc.	4					
Latency	4					
L2 cache						
Size	512 kB					
Assoc.	8					
Latency	16					
Br predictor	4 kB by 2-bit BHT, 6-bit GHR (no BTB) 9-cycle misprediction penalty					

Notes: Configuration in-2 corresponds to the Cell BE PPE. The number of functional units has been increased for wider-issue configurations.

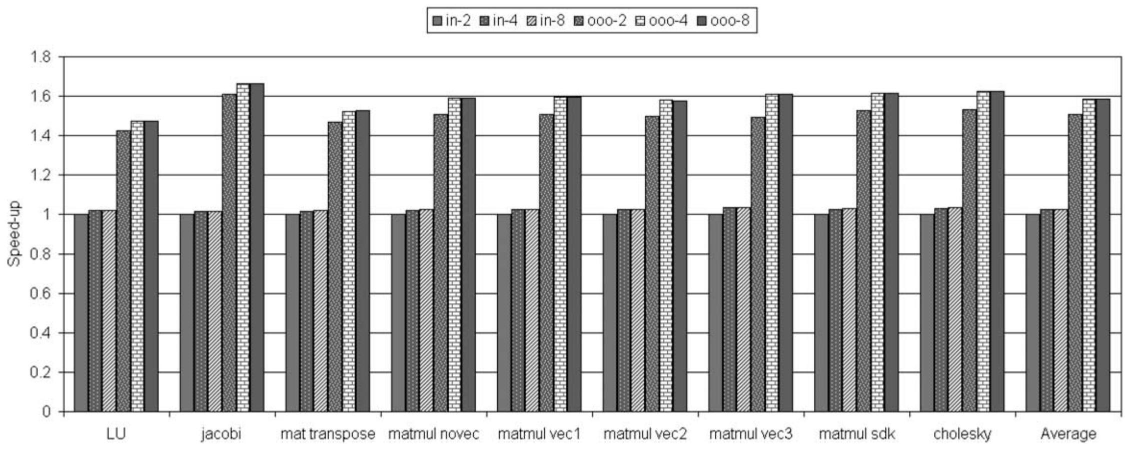
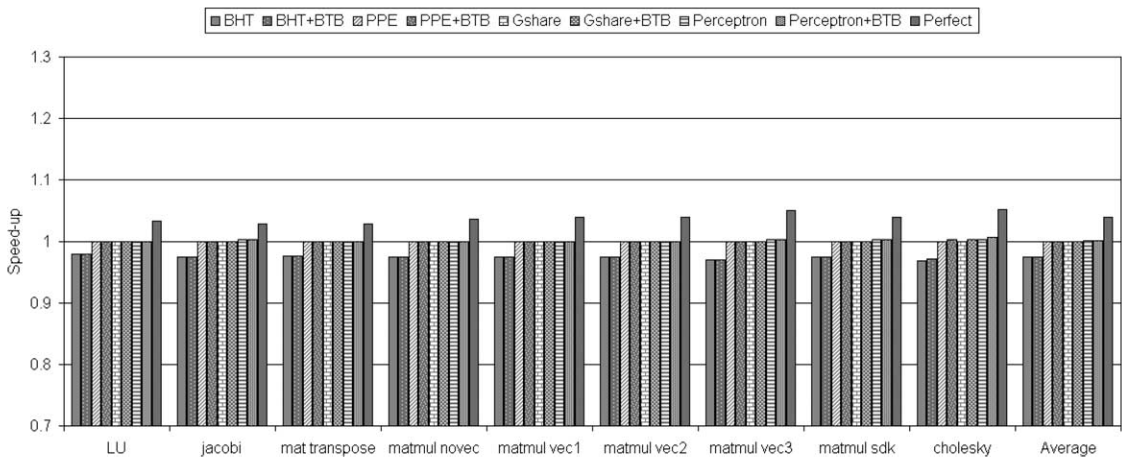


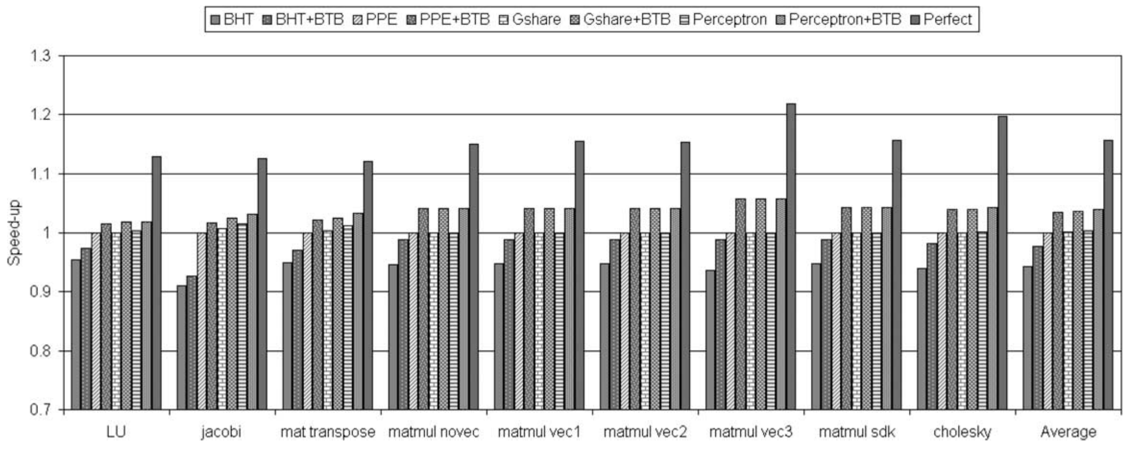
Fig. 6. Normalized IPC with respect to the Cell BE PPE (in-2). Out-of-order execution provides a 50% speed-up over in-order. Increasing issue width and functional units has small impact on in-order configurations and up to a 10% on out-of-order.

and is 4-way associative. The BHT and pattern history tables (PHT) of all the evaluated branch predictors have been configured with 16K 2-bit wide entries. The branch misprediction penalty is 9 cycles. Despite being quite lower than the Cell BE PPE 23 cycles, we have checked that both configurations show similar speed-up results for branch prediction.

Figure 7(a) shows the normalized IPC for the different branch predictors on in-order execution with respect to the in-2 configuration. The results show that the performance impact is very small. BHT is only 2.5% worse while Gshare and Perceptron performs the same as the PPE branch predictor. Perfect branch prediction outperforms the PPE's by 3.8%. Adding a BTB to any of these branch predictors on the in-*2* configuration does not provide any further improve-



(a)



(b)

Fig. 7. Normalized IPC with respect to the PPE branch predictor. BHT loses a 2.5% (in-2) and a 5.8% (ooo-2) performance, Gshare does not have any impact as well as Perceptron for in-2, which gains a mere 0.3% for ooo-2 on average. BTB does not impact in-2 but improves ooo-2 in a 3.5% for all branch predictors. Perfect branch prediction improves IPC in a 3.8% for in-2 and a 16% for ooo-2. (a) Branch prediction on in-2; (b) Branch prediction on ooo-2.

ment. Clearly, the in-order configurations are limited by something else than the branch predictor.

Figure 7(b) shows the results for the same experiments but on the ooo-2 configuration. As shown in the graph, all branch predictors have a larger influence on out-of-order execution than on the in-order one. BHT loses 5.8% performance on average, Gshare has the same performance, and Perceptron provides a negligible 0.3% speed-up. Also, adding a BTB does have a performance impact in this case. It provides a further 3.5% performance improvement on all branch predictors. Finally, perfect branch prediction provides an impressive 16% IPC improvement.

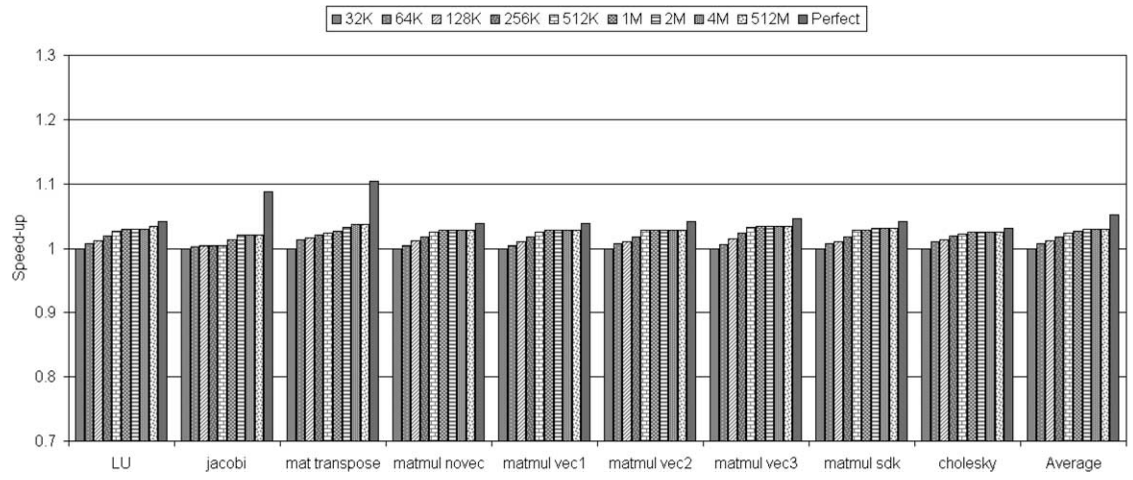
As a result, the evaluated branch predictors do not alter much the execution of the task generation phase. Real branch predictors speed-up ranges from −2.5% to 0% on in-order execution while the impact on the out-of-order PPE goes from −5.8% to 3.8% (Perceptron + BTB). Larger improvements were expected due to the fact that the amount of branch instructions in the task generation code is quite high, 17%, being conditional the 13%. However the results for perfect branch prediction are more interesting. While the task generation execution on the in-order configuration is a 3.8% faster with perfect branch prediction, the impact on out-of-order execution is a 16%. Nevertheless, this improvements are due to the Return Address Stack (RAS) accuracy, which is quite low for real branch predictors (20–30%). Perfect branch prediction implies perfect RAS behaviour which results in the observed higher speedups.

Given the negligible impact of perfect branch prediction on in-order execution, and its higher impact on out-of-order, we conclude that the major impact of improved branch prediction is an increase in the number of in-flight (overlapping) cache misses. In the next sections, we evaluate the impact of cache size and memory latency.

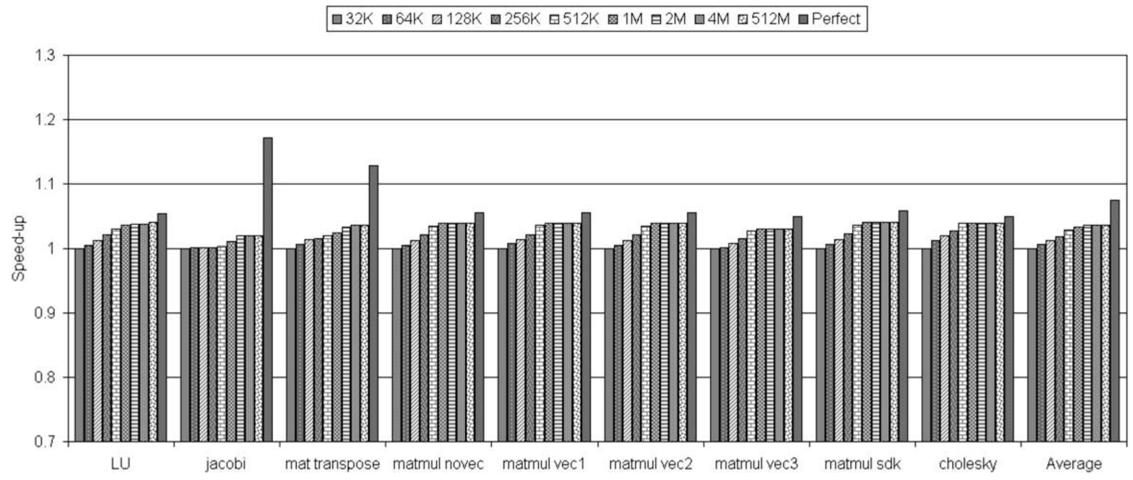
3.4. Cache size

The results in the previous sections show that Memory Level Parallelism (MLP) is a critical factor for the task generation phase of CellSs. In this section we analyze the importance of the cache size on the task generation execution. The Cell PPE includes a 32 kB 2-way associative instruction cache and a 32 kB 4-way associative data cache for level 1. The level 2 cache is 8-way associative and provides 512 kB of storage. In this section both L1 data cache and L2 cache are evaluated. All experiments assume a constant cache latency of 4 cycles to L1 and 16 extra cycles to L2 (total 20), since changing both parameters at the same time would make results harder to interpret.

We have simulated L1 cache sizes from 32 kB to 4 MB, plus a pseudo-infinite 512 MB configuration, with a fixed L2 size of 512 MB, so that the entire contents of the L1 fits in L2, and a perfect cache (always hit). Figure 8(a) shows results for 2-wide in-order execution and Fig. 8(b) for 2-wide out-of-order.



(a)



(b)

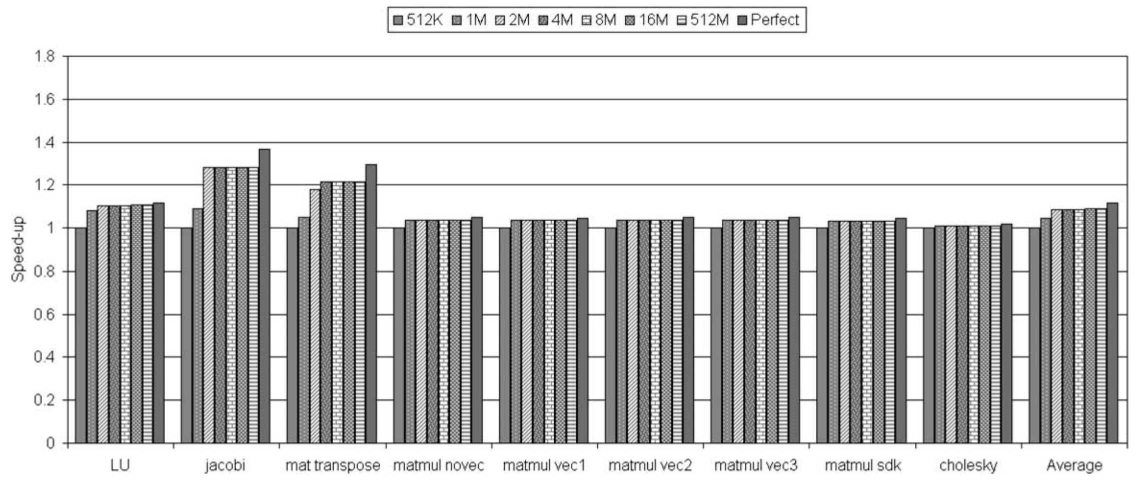
Fig. 8. Normalized IPC with respect to 32 kB L1 data cache on the in-2 and ooo-2 configurations. L2 cache is 512 MB. Speed-up increases with larger L1 data cache sizes up to 2.9% for 4 MB in-2 and 3.5% for 4 MB ooo-2. Perfect L1 data caches have a higher impact 5.2% (in-2) and 7.5% (ooo-2). (a) L1 data cache size on in-2; (b) L1 data cache size on ooo-2.

In both cases, IPC improves with cache size up to 4 MB. Cache sizes larger than 4 MB do not further improve performance. Even though the impact is greater for out-of-order execution, results for both execution orders are very small, 3.5% for 4 MB out-of-order and 2.9% for 4 MB in-order. However, the perfect L1 configurations provide higher speed-ups: up to 5.2% for in-*2* and 7.5% for ooo-*2*.

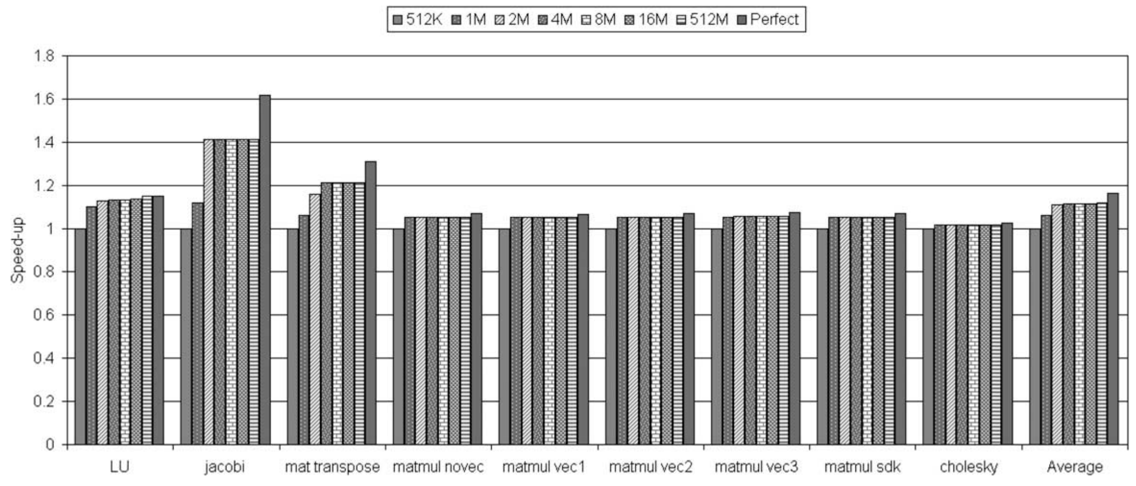
L2 cache size simulations have been carried out maintaining the original 32 kB data and instruction caches. The L2 cache size has been varied from the original 512 kB to 16 MB. As we did for the L1 cache, we have also studied a pseudo-infinite configuration of 512 MB and a perfect L2 cache. Figure 9(a) and (b) shows the normalized IPC for several L2 cache sizes with respect to the one with 512 kB on in-order and out-of-order executions, respectively.

As we already observed with the L1 simulations, IPC improves as the L2 grows up to 4 MB, but caches larger than 4 MB do not make a difference. The maximum speed-up for in-*2* is 9% and for ooo-*2* is 11.5%. On average, the improvement on out-of-order is only a 2% larger than for in-order so the L2 cache influence does not depend as much on the type of execution. In this case, a perfect L2 provides a 11.5% improvement for in-order and a 16% for out-of-order which is not as far from realistic configurations as the perfect and L1 cache size experiments.

Regarding cache size, L1 cache size can be kept reasonably small, since there is no big improvement when



(a)



(b)

Fig. 9. Normalized IPC with respect to 512 kB L2 on the in-2 and ooo-2 configurations. Speed-up increases with larger L2 cache sizes up to 9% for 4 MB in-2 and 11.5% for 4 MB ooo-2. Perfect L2 caches provide a larger improvement, 11.5% for in-2 and 16% for ooo-2, but much less than perfect L1 data caches. (a) L2 cache size on in-2; (b) L2 cache size on ooo-2.

going from 32 kB to 1 MB. The L2 cache size does have a more significant impact, and increasing cache size to 4 MB would provide some benefits. In any case, in the next section we perform some evaluations to verify the importance of the cache latency that we have observed here. From those simulations, we conclude that the whole working set of the task generation code could fit comfortably in a local on-chip memory since caches larger than 4 MB do not provide further speedups.

3.5. *Memory latency*

In the previous section we have seen that the performance of the task generation code does not improve with cache sizes beyond 4 MB. It would be interesting to off-load all css_addTask data to a local on-chip memory to avoid first reference and collision cache misses. Technologies such as IBM's eDRAM ("embedded DRAM") allow large on-chip memory sizes. However, a 4 MB memory would not have the 4 cycle latency of the 32 kB L1 cache. In this section we evaluate the performance impact of a memory hierarchy composed of a 32 kB L1 cache and an ideal memory with different latencies, ranging from 8 to 256 cycles. Figure 10 shows our simulation results for the different latencies of the ideal memory relative to the 16-cycle 512 kB L2 cache of the baseline configuration on the 2-way out-of-order processor.

Our results show that the latency of a local memory used exclusively by the task generation code does not seem to have a significant impact. As shown in the previous section, and ideal memory with a 16-cycle latency increases performance by over 15%. This plot shows that increasing the latency to 64 cycles still provides a similar improvement, and that even a 128-cycle memory would improve by close to 15%.

These simulations only serve the purpose of showing the potential benefits of using a local on-chip memory for the task generation code. Fully verifying these results, and measuring the actual size of such memory, would require rewriting part of the Cell Superscalar runtime library, which is beyond the scope of this work. However, given the potential improvement, we will consider it as future work to be explored.

4. Related work

The term scalability is widely used in the hardware and software communities. However, there is not a common understanding of the *scalability* concept. This issue was discussed by Hill in 1990 [13]. Hill presented several failed attempts to rigorously define scalability and finally questioned the usefulness of this concept and challenged the technical community to either provide a rigorous definition or stop using the term to describe systems. Several researchers took the challenge and attempted to provide a better definition [12,21]. Duboc et al. in [9] assert that a universal definition should be avoided and present a framework to analyze and predict the scalability of software systems. In this paper we use the intuitive definition of scalability applied to parallel systems presented by Zhang et al. in [31]. They claim that: "scalability measures the

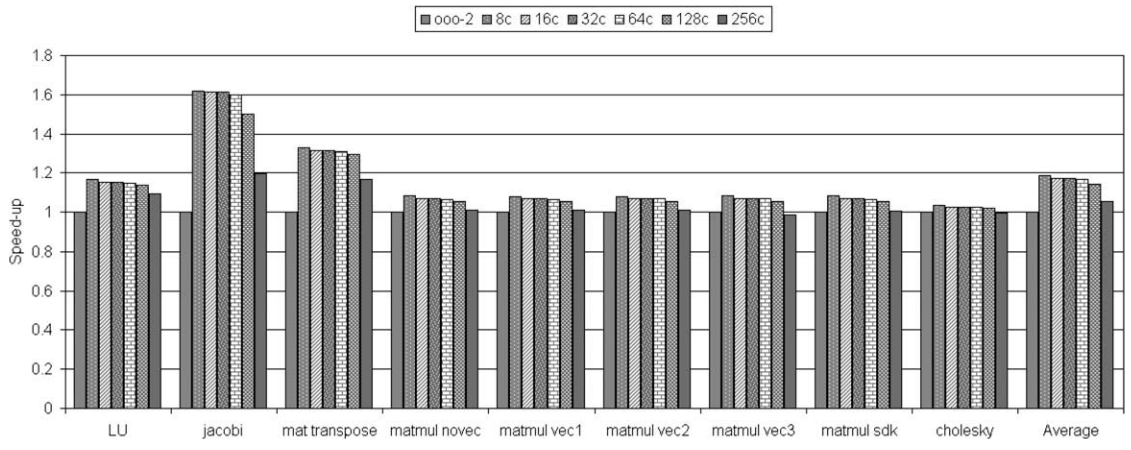


Fig. 10. Normalized IPC of an ideal memory with latencies ranging from 8 to 256 cycles with respect to the out-of-order PPE configuration (ooo-2). Ideal memories provide more than 15% performance improvement with latencies up to 128 cycles. Only the 256-cycle very high latency harm this performance, achieving only a 5% IPC improvement.

ability of a parallel system to improve performance as the size of an application problem and the number of processors involved increase".

Regarding resource utilization bounds, Liu and Layland presented in 1973 a study of the achievable processor utilization on single-processor systems [19]. This study provided the least upper bound (Liu and Layland bound (LLB)) to processor utilization factor depending on the scheduling algorithm. The processor utilization factor in Liu and Layland's work was defined as the fraction of processor time spent in the execution of a task set, which is equivalent to one minus the idle processor time. Lopez et al. extended this analysis for multiprocessor systems in [20]. They analyzed some allocation algorithms and proved their optimality in processor utilization for a specific scheduling scheme.

There are several studies and tools for measuring and analyzing the overheads of parallel programming models. Ovaltine [3] is a tool that performs OpenMP code instrumentation for analyzing OpenMP overheads, however it is restricted only to automatically compute load imbalance and *unparallelized* overheads. Scalea [28] is another tool for performance analysis of Fortran OpenMP, MPI and HPF codes. It allows the categorization of either a specific overhead for all parallel regions or all the overheads of a specific parallel region. Fürlinger and Gerndt presented a methodology for the analysis of the overheads and the scalability of OpenMP codes on CMP architectures in [11]. This analysis is performed using an OpenMP profiler that extracts four classes of overheads: synchronization, imbalance, limited parallelism (i.e, mutual exclusions) and thread management.

As previously mentioned in Section 1, there is a trend towards the use of tasks on parallel programming models. OpenMP [24] version 3.0 includes a pragma annotation for identifying functions as parallel independent tasks. Intel's Thread Building Blocks [27] parallel programming model provides an object-oriented interface to encapsulate the computation and the working set of independent tasks. Cilk is another parallel programming model with specific keywords that allow off-loading tasks on function calls. Tagged Procedure Calls (TPC) [18] is a pure task-based parallel programming model. It provides semantics for declaring and spawning functions as parallel tasks. Their communication and synchronization is transparent to the programmer, who only has to specify the data used by the parallel computation.

The Cell BE SDK [6] provides a low-level library for programming all the processors in the chip. Several programming models were proposed in order to distribute computation among the processing elements [7]. However, all of them require the programmer to deal with code and data distribution. Bellens et al. presented the Cell Superscalar task-based programming model for the CBEA in [4]. This programming model provides a higher-level abstraction and allows programming the Cell BE using sequential code and OpenMP-like annotations for functions. Annotated functions are considered parallel tasks and, as well as TPC, all synchronization and communication among tasks is performed by the runtime.

There are other type of architectures that are proper to task-based programming. The Viper [10] by NXP is a multimedia-specific processor for set-top box devices. It is composed by a general-purpose MIPS processor, a VLIW TriMedia coprocessor and a set of multimedia specific accelerators and controllers. Viper applications start their execution on the MIPS processor which off-loads work to the TriMedia core. Both of them eventually dispatch tasks to the specific accelerators for encoding/decoding/filtering audio and/or video signals. This is also the case of the Nomadik [1] architecture by STMicroelectronics, which integrates an ARM processor, and video and audio accelerators. Apart from embedded multimedia architectures, the task-based scheme can be also applied to the GPGPU programming concept. It consists on programming GPUs in order to execute general-purpose code. In this case, program execution is launched on a general-purpose core which may off-load computation- and data-intensive tasks to GPUs in order to accelerate their execution. An example of GPGPU is the NVIDIA CUDA technology [23] which allows general-purpose execution on NVIDIA GPUs.

The contributions of our paper are the scalability study for Cell Superscalar on the Cell BE, based on the comparison of the task execution costs and the task generation overhead, and the simulation study leading to the desirable design of a processor targeting a faster execution of the task generation phase.

5. Conclusions

In this paper we have evaluated the performance of Cell Superscalar applications in terms of their scalability to next generation CBEA implementations including more SPE processors. We observe that the fact that the SPU must fit all of its working set on the Local Store effectively limits the size of the tasks to be executed there, making the task generation overhead the limiting factor for scalability with the number of processors. Our results show that highly optimized tasks, such as matrix multiply, take approximately 20 μs to run, while the task generation overhead is close to 3 μs, limiting scalability to 6–7 processors.

Since task size can not be increased due to memory size limitations, we must reduce the task generation overhead, which currently runs on the PPE. The PPE had to be very small, and run at a high frequency, leading to a simple 2-way in-order superscalar design. We have evaluated the impact of out-of-order execution, wider superscalar issue, better branch prediction, and larger L1 and L2 caches.

Our results show that out-of-order execution has the highest impact on the task management, increasing performance by over 50%. Further analysis shows that cache latency is also critical for the task generation overhead, but it only has a significant impact when coupled with out-of-order execution. We conclude that the combination of a larger cache and the overlapping of multiple cache misses due to out-of-order execution are the key to lower overheads, and higher scalability. Finally, a set of simulations using a local memory, private to the task generation code, shows that there is a high potential compared to relying on the conventional cache hierarchy.

Given these results, a next generation CBEA implementation should include either a more aggressive PPE with out-of-order execution, or some other form of task management accelerator coupled with a larger local memory in order to be able to use all the available on-chip processors.

Acknowledgments

We want to thank Rosa M. Badia, Josep M. Perez and Pieter Bellens from the Barcelona Supercomputing Center and Felipe Cabarcas from the Universitat Politecnica de Catalunya for their collaboration on the study of CellSs.

This work has been supported, in part, by the Spanish Ministry of Science and Education, scholarship AP2005-4245 and contract CICYT TIN2007-60625; by the Scalable Architecture (SARC) project FP6/FET-27648; and the HiPEAC European Network of Excellence FP6/IST-0044608.

References

[1] A. Artieri, V. D'Alto, R. Chesson, M. Hopkins and M.C. Rossi, Nomadik open multimedia platform for next-generation mobile devices, STMicroelectronics Technical Article TA305, 2003.

[2] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams and K.A. Yelick, The landscape of parallel computing research: A view from Berkeley, Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[3] M. Bane and G. Riley, Automatic overheads profiler for OpenMP codes, in: *EWOMP2000: Proceedings of the 2nd Workshop on OpenMP*, Edinburgh, UK, 2000.

[4] P. Bellens, J.M. Perez, R.M. Badia and J. Labarta, CellSs: a programming model for the Cell BE Architecture, in: *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Tampa, FL, USA, 2006, p. 86.

[5] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall and Y. Zhou, Cilk: An efficient multithreaded runtime system, *ACM SIGPLAN Notices* **30**(8) (1995), 207–216.

[6] Cell Broadband Engine Architecture home page, http://www.ibm.com/developerworks/power/cell/.

[7] Cell Software Solutions Programming Model, http://www.power.org/resources/devcorner/cellcorner/DucVianney_GDC-CellSoftwareSolutionsProgrammingModel.pdf.

[8] Cell Superscalar version 1.4, http://www.bsc.es/plantillaG.php?cat_id=179.

[9] L. Duboc, D.S. Rosenblum and T. Wicks, A framework for modelling and analysis of software systems scalability, in: *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, 2006, pp. 949–952.

[10] S. Dutta, R. Jensen and A. Rieckmann, Viper: A multiprocessor SOC for advanced set-top box and digital TV systems, *IEEE Design and Test of Computers* **18**(5) (2001), 21–31.

[11] K. Fürlinger and M. Gerndt, Analyzing overheads and scalability characteristics of OpenMP applications, in: *VECPAR'06: Proceedings of the 7th International Meeting High Performance Computing for Computational Science*, Rio de Janeiro, Brazil, 2006.

[12] D. Gustavson, The many dimensions of scalability, in: *Digest of Papers Compcon Spring*, San Francisco, CA, USA, 1994, pp. 60–63.

[13] M.D. Hill, What is Scalability?, *SIGARCH Computer Architecture News* **18**(4) (1990), 18–21.

[14] H.P. Hofstee, Power efficient processor architecture and the Cell Processor, in: *HPCA'05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, IEEE Computer Society, San Francisco, CA, USA, 2005, pp. 258–262.

[15] IBM Binary Prober, http://www.alphaworks.ibm.com/tech/bprober.

[16] D. Jimenez and C. Lin, Dynamic branch prediction with perceptrons, in: *HPCA'01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, Monterrey, Mexico, 2001, pp. 197–206.

[17] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer and D. Shippy, Introduction to the Cell multiprocessor, *IBM Journal of Research and Development* **49**(4/5) (2005), 589–604.

[18] K. Kapelonis, S. Karlsson and A. Bilas, Tagged procedure calls: A programming model for scalable systems using multicore processors and tightly-coupled interconnects, in: *2nd HiPEAC Industrial Workshop*, Eindhoven, The Netherlands, 2006; http://www.hipeac.net/industry_workshop2.

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

[20] J. Lopez, J. Diaz and D. Garcia, Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling, *IEEE Transactions on Parallel and Distributed Systems* **15**(7) (2004), 642–653.

[21] E. Luke, Defining and measuring scalability, in: *SPLC'93: Proceedings of the Scalable Parallel Libraries Conference*, Mississippi, MS, USA, 1993, pp. 183–186.

[22] S. McFarling, Combining branch predictors, Technical Report WRL-TN-36, 1993.

[23] NVIDIA CUDA toolkit, http://www.nvidia.com/cuda.

[24] OpenMP home page, http://www.openmp.org.

[25] V. Pillet, J. Labarta, T. Cortés and S. Girona, PARAVER: A tool to visualize and analyse parallel code, in: *WoTUG-18: Proceedings of Transputer and Occam Developments*, Manchester, UK, 1995, pp. 17–31 (also as: Technical Report UPC-CEPBA-95-03).

[26] J.E. Smith, A study of branch prediction strategies, in: *ISCA'81: Proceedings of the 8th Annual Symposium on Computer Architecture*, Minneapolis, MN, USA, 1981, pp. 135–148.

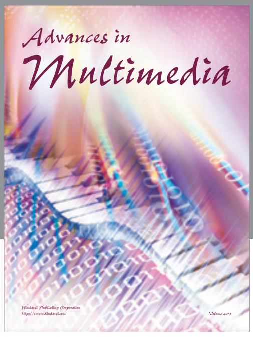
[27] Thread Building Blocks home page, http://threadbuildingblocks.org.

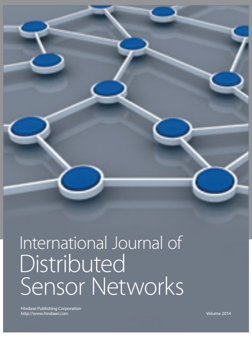
[28] H.L. Truong and T. Fahringer, SCALEA: A performance analysis tool for distributed and parallel programs, in: *Euro-Par'02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Paderborn, Germany, 2002, pp. 75–85.

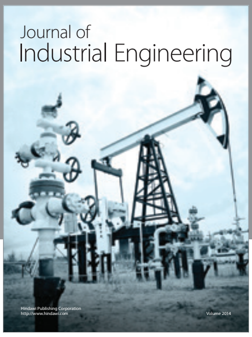
[29] J.W. Tukey, *Exploratory Data Analysis*, Addison-Wesley, 1977.

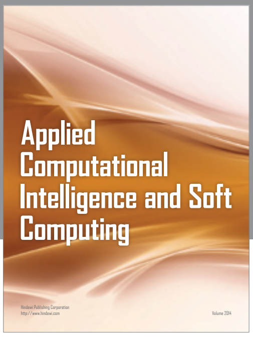
[30] D. Tullsen, S. Eggers and H. Levy, Simultaneous multithreading: Maximizing on-chip parallelism, in: *ISCA'95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995, pp. 392–403.

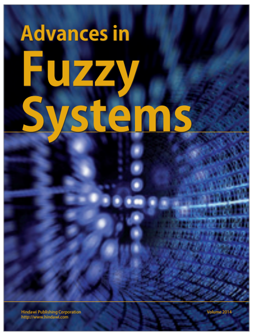
[31] X. Zhang and Z. Xu, Multiprocessor scalability predictions through detailed program execution analysis, in: *ICS'95: Proceedings of the 9th International Conference on Supercomputing*, Barcelona, Spain, 1995, pp. 97–106.











Hindawi

Submit your manuscripts at
http://www.hindawi.com