Task Management Analysis on the CellBE

Alejandro Rico, Alex Ramirez y Mateo Valero[1]

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 programing 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 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, and larger caches on the task management overhead.

Keywords— **Scalability, multicore, task-based programming models, Cell processor**

I. Introduction

POWER consumption and design complexity have led the computer architecture community to design chip multiprocessors (CMP). Current commercial CMP integrate 4 to 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 [1]. 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 [2], Intel's Thread Building Blocks (TBB) [3], and Cilk [4]. There are also pure task-based parallel programming models such as Cell Superscalar [5] and Tagged Procedure Calls (TPC) [6]. 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 CellBE, a state-of-the-art CMP 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 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 [7]. 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. 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

[1]Dpt. d'Arquitectura de Computadors, Univ. Politècnica de Catalunya, e-mail: {arico,aramirez,mateo}@ac.upc.edu.

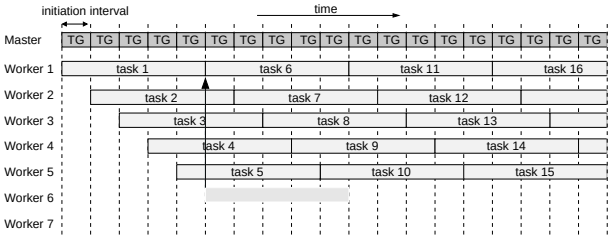


Fig. 1. Task distribution among processing elements on an 8-core multiprocessor. Master thread continuously generates tasks. Workers 6 and 7 remain idle during the whole execution. The initiation interval is the task generation (TG) time.

(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.

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

The result of equation 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 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 (i.e. 16x16-pixel macroblocks). Hardware specifications could also be a limiting factor for the task size. Architectures using local memories, such as Cell 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 Processor [8] is a 9-core heterogeneous multiprocessor with a general-purpose processor and 8 accelerators. 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 [9]. 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 Processor running Cell Superscalar [5] applications. This analysis is performed using high performance scientific applications (section II). Section III 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. The conclusions are exposed in section IV.

II. Scalability on the Cell

This section presents a scalability study of applications written using the Cell Superscalar programming model [5] on the Cell processor. First, we present the execution environment, and the performance analysis methodology. Then, using the presented methodology we measure the achievable parallelism for a set of scientific application kernels.

The Cell Processor is a joint initiative of Sony, Toshiba and IBM (STI). It is composed of a GPP named PowerPC Processor Element (PPE) and 8 SIMD accelerators: the Synergistic Processor Elements (SPE). The PPE is a 64-bit PowerPC-compliant processor with in-order execution and 2-way SMT support. It integrates a VMX unit, 32-KByte L1 instruction and data caches and a 512-KByte L2 cache. Each SPE is composed of a Synergistic Processor Unit (SPU), a 256-KByte 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 are performed through the DMA engine incorporated in the MFC.

The Cell Superscalar (CellSs) programming model is a task-based programming model for the Cell Processor. 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. CellSs applications execute 2 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 which 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.

A. Methodology

The scalability analysis in this section has been performed on IBM QS21 Blades, with 2 Cell processors running at 3.2 GHz and 512MB of XDR memory. The CellSs runtime library is instrumented to provide timing information and detailed performance analysis traces. After program execution, we ob-

tain 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 [10], a visualization tool that allows graphically representing the execution phases and threads communication 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. Moreover, since measurements are performed in a real system, results may significantly vary from one execution to another. Hence, the results presented for an application are the average over 10 executions of the same program.

B. 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 written and compiled with CellSs. 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 [9]. 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.

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. 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.

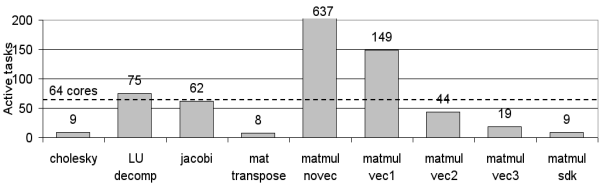


Fig. 2. Maximum active tasks for CellSs high performance scientific applications on the Cell Processor. Only non-fully optimized versions of matrix multiplication and LU decomposition achieve more than 64 active tasks.

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 Processor. However, two Cell 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. 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 processor. 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.

III. 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 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.

A. Simulation Setup

The task generation phase simulations have been carried out using the SMTSIM [11] simulator extended for PowerPC traces. The traces has been generated using the IBM BProber [12] 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 II: Cholesky factorization, LU decomposition, Jacobi, matrix transposition and the five versions of matrix multiplication.

SMTSIM has been extended to support in-order execution and has been configured to simulate a Cell PPE processor. Table I 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 I.

B. 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

TABLE I
SMTSIM parameter configurations. Configuration in-2 corresponds to the Cell PPE. The number of functional units has been increased for wider-issue configurations.

		in-2,4,8	ooo-2,4,8
Execution		in order	out of order
Fetch width		8 (4 each thread)	
Issue width		2, 4, 8	2, 4, 8
FU	Integer	2, 4, 8	2, 4, 8
	Ld/St	1, 2, 4	1, 2, 4
	FP	1, 2, 4	1, 2, 4
L1 I		32KB, 2-way, 1 cycle	
L1 D		32KB, 4-way, 4 cycles	
L2		512KB, 8-way, 16 cycle	
Br Predictor		16K-entry BHT 6-bit GHR (no BTB)	

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 [13]. 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 3 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.

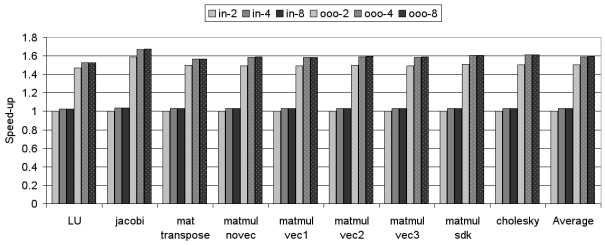


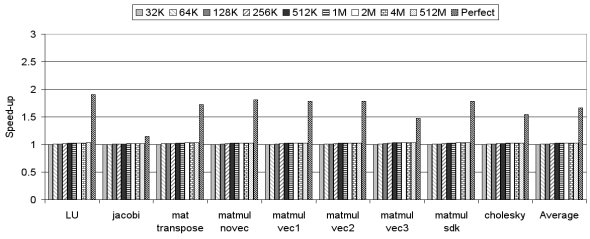
Fig. 3. Normalized IPC with respect to the Cell PPE (in-2). Out-of-order execution provides a 50% speed-up over in-order. Increasing issue width and functional units has small impact on in-order configurations and up to a 10% on out-of-order.

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%.

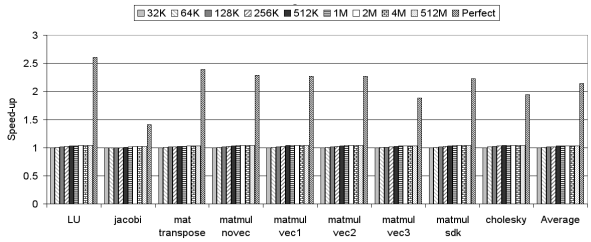
The results show that out-of-order execution is a very desirable hardware design for CellSs task generation performance. A 2/4-wide issue out-of-order design would decrease by 50%-60% the task generation overhead which means exploiting a 50%-60% more task-level parallelism.

C. 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 32KB 2-way associative instruction cache and a 32KB 4-way associative data cache for level 1. The level 2 cache is 8-way associative and provides 512KB 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.



(a) L1 data cache size on in-2



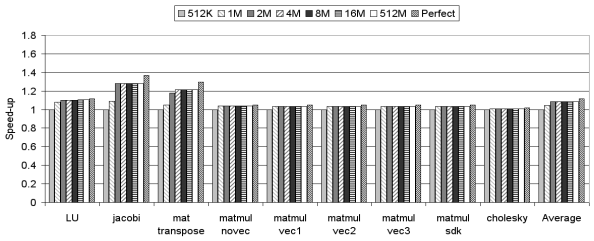
(b) L1 data cache size on ooo-2

Fig. 4. Normalized IPC with respect to 32KB L1 data cache on the in-2 and ooo-2 configurations. L2 cache is 512MB. Speed-up increases with larger L1 data cache sizes up to 2.6% for 4MB in-2 and 3.3% for 4MB ooo-2. Perfect L1 data caches have a much larger impact 67% (in-2) and 120% (ooo-2).

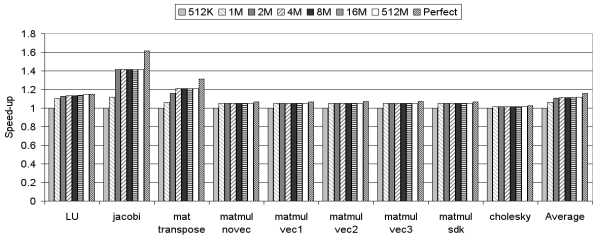
We have simulated L1 cache sizes from 32KB to 4MB, plus a pseudo-infinite 512MB configuration, with a fixed L2 size of 512MB, so that the entire contents of the L1 fits in L2, and a perfect cache (always hit). Figure 4 shows results for 2-wide in-order execution, and for 2-wide out-of-order.

In both cases, IPC improves with cache size up to 4MB. Cache sizes larger than 4MB do not further improve performance. Even though the impact is greater for out-of-order execution, results for both execution orders are very small, 3.3% for 4MB out-of-order and 2.6% for 4MB in-order. However, the perfect L1 configurations provide much higher speed-ups: up to 67% for *in-2* and 2.2x (120%) for *ooo-2*.

L2 cache size simulations have been carried out maintaining the original 32KB data and instruction caches. The L2 cache size has been varied from the original 512KB to 16MB. As we did for the L1 cache, we have also studied a pseudo-infinite configuration of 512MB and a perfect L2 cache. Figures 5(a) and 5(b) show the normalized IPC for several L2 cache sizes with respect to the one with 512KB on in-order



(a) L2 cache size on in-2



(b) L2 cache size on ooo-2

Fig. 5. Normalized IPC with respect to 512KB L2 on the in-2 and ooo-2 configurations. Speed-up increases with larger L2 cache sizes up to 13% for 4MB in-2 and 15% for 4MB ooo-2. Perfect L2 caches provide a larger improvement, 20% for in-2 and 25% for ooo-2, but much less than perfect L1 data caches.

and out-of-order executions respectively.

As we already observed with the L1 simulations, IPC improves as the L2 grows up to 4MB, but caches larger than 4MB do not make a difference. The maximum speed-up for *in-2* is 13% and for *ooo-2* is 15%. 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 20% improvement for in-order and a 25% for out-of-order which is not as far from realistic configurations as the perfect and L1 cache size experiments.

From these results, we conclude that the task generation code is very sensitive to the memory latency, since a perfect L1 (4 cycles) has a 2x performance impact, while a perfect L2 (20 cycles) only provides a 20% speedup. Regarding cache size, L1 cache size can be kept reasonably small, since its latency seems so important and there is no big improvement when going from 32KB to 1MB. The L2 cache size does have a more significant impact, and increasing cache size to 4MB 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.

D. Memory Latency

In the previous section we have seen that the performance of the task generation code does not improve with cache sizes beyond 4MB, and that cache latency plays a very important role, since a perfect L1 cache improves performance by over 100%, while a perfect L2 only improves by 20%.

From those simulations, we conclude that the whole working set of the task generation code could fit comfortably in a local on-chip memory. However, a 4MB memory would not have the 4 cycle latency of the 32KB L1 cache. In this section we evaluate

the performance impact of a memory hierarchy composed of a 32KB L1 cache and an ideal memory with different latencies, ranging from 8 to 256 cycles. Figure 6 shows our simulation results for the different latencies of the ideal memory relative to the 16-cycle 512KB L2 cache of the baseline configuration on the 2-way out-of-order processor.

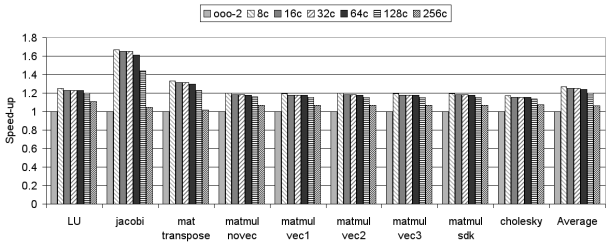


Fig. 6. Normalized IPC of an ideal memory with latencies ranging from 8 to 256 cycles with respect to the out-of-order PPE configuration (ooo-2). Ideal memories provide more than 20% performance improvement with latencies up to 128 cycles. Only the 256-cycle very high latency harm this performance, achieving only a 6% IPC improvement.

Our results show that the latency of the 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 20%. This plot shows that increasing the latency to 64 cycles still provides a similar improvement, and that even a 128-cycle memory would improve a 20%.

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.

IV. Conclusions

In this paper we have evaluated the performance of Cell Superscalar applications in terms of their scalability to next generation Cell architectures 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 microseconds to run, while the task generation overhead is close to 3 microseconds, 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, 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 Cell architecture 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.

Acknowledgements

We want to thank Rosa M. Badia, Josep M. Perez and Pieter Bellens from the BSC and Felipe Cabarcas from the UPC 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 SARC project FP6/FET-27648; and the HiPEAC European Network of Excellence FP6/IST-0044608.

References

[1] K. Asanovic et al., "The Landscape of Parallel Computing Research: A View from Berkeley," Tech. Rep., Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 2006.
[2] "OpenMP home page," http://www.openmp.org.
[3] "Thread Building Blocks home page," http://threadbuildingblocks.org.
[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995.
[5] P. Bellens, J. M. Perez, R. M. Badia, and Jesus Labarta, "CellSs: a Programming Model for the Cell BE Architecture," in *SC '06*, Nov. 2006, p. 86.
[6] K. Kapelonis, S. Karlsson, and A. Bilas, "Tagged Procedure Calls: A Programming Model for Scalable Systems using Multi-core Processors and Tightly-coupled Interconnects," 2nd HiPEAC Industrial Workshop, Oct. 2006.
[7] M. D. Hill, "What is Scalability?," *SIGARCH Comput. Archit. News*, vol. 18, no. 4, pp. 18–21, 1990.
[8] 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*, vol. 49, no. 4-5, pp. 589–604, July 2005.
[9] "Cell Broadband Engine Architecture home page," http://www.ibm.com/developerworks/power/cell/.
[10] V. Pillet, J. Labarta, T. Cortés, and S. Girona, "PARAVER: A Tool to Visualize and Analyse Parallel Code," in *WoTUG-18*, Apr. 1995, pp. 17–31, Also as Technical Report UPC-CEPBA-95-03.
[11] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism," in *ISCA'95*, June 1995, pp. 392–403.
[12] "Ibm Binary Prober," http://www.alphaworks.ibm.com/tech/bprober.
[13] H. Peter Hofstee, "Power Efficient Processor Architecture and the Cell Processor," in *HPCA '05*, Washington, DC, USA, 2005, pp. 258–262, IEEE Computer Society.