Tagged Procedure Calls (*TPC*): Efficient runtime support for task-based parallelism on the Cell Processor

George Tzenakis[†], Konstantinos Kapelonis, Michail Alvanos[†], Konstantinos Koukos[†], Dimitrios S. Nikolopoulos[†], and Angelos Bilas[†]

Institute of Computer Science (ICS)
Foundation for Research and Technology - Hellas (FORTH)
100 N. Plastira Av. Vassilika Vouton, Heraklion, GR-70013, Greece
{tzenakis,kkapelon,alvanos,koukos,dsn,bilas}@ics.forth.gr

Abstract. Increasing the number of cores in modern CPUs is the main trend for improving system performance. A central challenge is the runtime support that multi-core systems ought to use for sustaining high performance and scalability without increasing disproportionally the effort required by the programmer. In this work we present *Tagged Procedure Calls* (*TPC*), a runtime system for supporting task-based programming models on architectures that require explicit data access specification by the programmer. We present the design and implementation of *TPC* for the Cell processor and examine how the runtime system can support task management functions with on-chip communication only. Through minimizing off-chip transactions in the runtime, we achieve sub-microsecond task initiation latency and minimum null task initiation/completion latency of 385 ns. We evaluate *TPC* with several kernels and applications, demonstrating that *TPC* achieves scalable on-chip execution of codes previously parallelized and optimized for shared-memory multiprocessors, can exploit additional fine-grain parallelism in codes previously parallelized at coarse levels of granularity, and performs competitively to existing task-based parallel programming frameworks that statically optimize data layout and task placement.

1 Introduction

Technology trends dictate that future high-performance, general-purpose and embedded systems will be built using heterogeneous chip multi-processors (CMPs) with many cores and tightly-coupled interconnects. Heterogeneous many-core CMPs require a large degree of parallelism in applications as well as dealing with heterogeneity, without significantly increasing programming effort.

[†]Also, with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.

For this reason, the role of the programming model is significant for future CMPs. The two main, explicitly parallel programming models used today are shared memory and message passing. Shared memory requires programs to specify synchronization information for memory accesses. Message passing on the other hand requires programs to deal with data placement and communication buffer management. In both cases, application and system designers have been tantalized by the effort required to program and debug such systems for over two decades. The main issue appears to be drawing a different balance between the mechanisms that are available in the underlying system and the abstraction that is exposed to the applications.

We believe that task-based programming models have the potential to achieve this balance. At a high level, explicitly parallel, task-based programming models have two advantages: On one hand they force the programmer to consider code complexity and data transfers at design time without worrying about the underlying mechanisms for communication and synchronization. On the other hand they provide the underlying system (runtime and architecture) with extensive information for efficient execution and runtime optimization. Thus, tasks as an abstraction, present the potential for achieving efficient execution and reducing programmer effort.

Although task-based programming models have been proposed in the past, modern CMPs present new opportunities. Previous efforts with task-based programming models had to deal with coarse-grained tasks due to task management overhead. Task management operations, such as initiation, completion, queuing, and scheduling, in traditional parallel systems cost in the order of tens of thousands of cycles, relative to the clock cycle time of modern processors, due to communication and memory management overheads [14]. In turn, coarse-grained tasks make it hard for the programmer to identify and delineate tasks and, even more so, task and data dependencies. In contrast, fine-grained tasks are easier to identify in sequential codes by inspection as they require analyzing and resolving fewer data and control dependencies. Modern CMPs have the potential of significantly reducing the required task size and achieve efficient execution while reducing the associated effort to identify parallelism.

In this paper we introduce a runtime system for the Cell processor [8], *Tagged Procedure Calls* (*TPC*), that aims at supporting task-based programming models. The notion of a task is general and can be interpreted in various ways. In our work we consider a task to be a piece of code that can execute in parallel *as well as* the data that will be accessed by the code. Despite their advantages, fine-grained tasks impose significant challenges for the runtime system. They require efficient basic mechanisms for task management, in particular, task initiation and completion that now become common-path operations. In this work we focus on better understanding and minimizing these basic overheads associated with task management.

We first examine the overhead associated with task management operations on a real system. We focus on task initiation, task completion, task queuing, and task data transfer. Our implementation of *TPC* achieves null task initiation

latency from 180 to 380 cycles on the 3.2 GHz Cell processor, depending on the size of the argument list. This represents a significant improvement over task initiation latencies reported in earlier work on task-level parallel execution systems on the Cell [14]. The null task round-trip overhead in *TPC* is about 385 ns, when the ideal DMA round-trip latency of the Cell is just under 312 ns [2].

We examine the performance of *TPC* using both kernels and real applications. We port two applications from the SPLASH-2 [18] suite (FFT and LU) and demonstrate that porting applications written and optimized for shared-memory multiprocessors to *TPC* requires mostly simple and mechanical code changes. *TPC* achieves nearly perfect scaling of these codes on the Cell cores. We further port two applications written previously to exploit coarse-grain parallelism on multi-processors and clusters, PBPI [7] and an H.264 video encoder [17]. We demonstrate that *TPC* enables the exploitation of further fine-grain on-chip parallelism in these applications, with manageable programming effort. Lastly, we port and evaluate several benchmarks distributed with the Sequoia programming language [6]. This effort demonstrates that *TPC* performs competitively to existing task-based parallel programming models for the Cell.

The rest of this paper is organized as follows. Section 2 presents the design and implementation of *TPC* and its runtime system on the Cell processor. Section 3 presents the hardware and software environment we used for our performance evaluation. Section 4 presents our experimental results. In Section 5 we discuss the advantages of *TPC* over previous efforts and related work. Finally, we draw our conclusions in Section 6.

2 *TPC* Design and Implementation

The Cell processor contains a general purpose PowerPC Processing Element (PPE) and eight special purpose Synergistic Processing Elements (SPEs) with their own instruction set. Each SPE has 256 KBytes of local memory without any other cache between this memory and the SPE core. The PPE has a coherent memory hierarchy with two levels of cache prior to the single global external memory. DMAs in the Cell are capable of scatter/gather functions and can have multiple (16 per SPE) outstanding transfers. Moreover, the PPE can access the local memories of SPEs with remote load/store operations as they are mapped to the main memory address space (MMIO). The PPE and SPEs can also communicate with messages via small mailbox registers. These options create trade-offs that need to be understood before the runtime system is able to take advantage of them. Finally, all communication in the Cell processor happens over an on-chip element interconnect bus (EIB) that consists of four bi-directional rings.

TPC uses program annotations to identify certain procedure calls as concurrent tasks. Currently, annotations occur at the procedure level. Programmer can encapsulate blocks of code or groups of loop iterations in *TPC* procedures. *TPC* procedure calls execute in the same or another core, as asynchronous tasks, with the current core continuing execution. In this work, procedure arguments can be *in*, *out*, or *inout*. The issuing task can wait for tasks using point-to-point or bar-

rier synchronization. When issuing an asynchronous task, the runtime returns a handle that can be used later for managing the specific *instance* of the issued task, while the issuing task continues with program execution. When a task completes, it notifies the issuer for its completion. *TPC* procedures have no return values and all arguments are passed by reference. *TPC* arguments and their sizes are determined at runtime before task initiation. *TPC* supports continuous and fixed-stride arguments. We expect that programming interface extensions for specifying memory layout for task arguments will play an important role on programmer effort.

The *TPC* runtime library consists of two main parts, the *initiator* and the *target.* Although any core can play the role of the initiator or target, currently, and due to the Cell architecture, we only support task initiation from the PPE. Similarly, only SPEs can execute tasks as targets. Each task consists of a descriptor. Task descriptors are prepared by the initiator and they are placed in task queues for execution. There is one task queue per target, located in its local storage. The task descriptor contains the function id and the list of arguments. For every argument, the descriptor specifies the argument's address in main memory, the argument size, a flag indicating if it is *in*, *out*, or *inout*, and for stride arguments the stride between the elements.

TPC uses a private task queue for each SPE. The task queue itself is an array of task descriptors. Since our goal is to eliminate off-chip operations, we place each task queue in the local storage of the corresponding SPE. In addition to the task queue, the runtime maintains a completion queue for each SPE (Figure 1(a)). The PPE polls each completion queue for task status notifications from the SPEs. When a completion is received the task entry in the corresponding task queue is released. Since tasks run to completion in each SPE, tasks complete in order. The task completion status consists of a flag and a task id.

An important architectural aspect for implementing a task-based runtime is the available mechanisms for communication among different memories and cores. Although DMA performance on the Cell has been thoroughly analyzed in previous work [2], low-latency control transfer mechanisms have not been fully explored. In this work we examine PPE to SPE round-trip latency with various mechanisms. We use the PPE as initiator, so the available options are: mailbox messages, remote stores to SPE's local store (MMIO), and PPE-initiated DMAs.

SPEs can communicate with the PPE via mailbox messages, DMA, or a variant of DMA using the Atomic Cache Unit (ACU). A simple, non-atomic DMA transfer writes results to main memory and invalidates the PPE's cache, thus requiring off-chip accesses. The ACU is intended for implementing high-performance atomic synchronization primitives between SPEs and the PPE in the global address space, using direct cache-to-cache transfers that remain on-chip. This mechanism supports reserve-line (load-locked), conditional-store, and unconditional-store operations.

Task initiation: Mailboxes are not appropriate mechanisms for initiating tasks. First, the mailbox register resides in the SPE's MFC. Sending mailbox messages incurs in the PPE the same cost as a remote store operation because the SPE

mailbox register is memory mapped to the PPE in the same way as the SPE local memory. In addition, to safely use the mailbox register a remote load is required first to check the status of the mailbox register and to ensure that previous mailbox messages have been consumed by the SPE. This introduces a network round-trip latency before posting the mailbox message. Using PPE-initiated DMA requires five remote store operations to special SPE registers. Then, the DMA controller of the SPE performs the actual DMA from main memory to the local SPE memory. Thus, after preparing a task descriptor in (cached) memory, the only two realistic options for the PPE to initiate a task are: (a) issuing remote stores to post the descriptor to the SPE task queue or (b) issuing fewer stores to indicate the existence of a new task descriptor, which then the SPE can pull using DMA. Note that the first approach requires a number of MMIO stores from the PPE that depend on the size of the task descriptor for each task. The second approach requires a fixed number of remote stores at the PPE but introduces an additional DMA transfer in the SPE. Assuming the task descriptor is not evicted from the PPE cache, both approaches result in on-chip traffic only. In all cases, PPE stores to SPEs are cache inhibited and complete in program order. The PPE can use vector store instructions to reduce the number of stores required to post a single task descriptor. The final store instruction sets the active flag of the task descriptor in the task queue to notify the SPE of a new task arrival, while the SPE polls its local memory. In our evaluation we examine both options for task initiation.

Task pre-fetching and execution: Once a new task has been posted to the SPE's task queue, the SPE extracts the task descriptor, fetches *in* arguments, executes the designated function, and writes back *out* arguments. The main challenge in executing these steps is to maximize overlapping of data transfers with task execution. To achieve this, *TPC* pipelines the different stages of task execution and uses pre-fetching to overlap argument transfers and task execution.

Each task can be in one of the states ACTIVE, FETCH, READY, WRITE-BACK, COMPLETE. Before executing a task that is ready, the SPE prepares and issues the DMA commands for as many active tasks as possible from its task queue, depending on the available space in the local storage, and places these tasks in the fetch state. Then, it turns to executing the first task in the queue whose arguments are available. When a task is done executing, the SPE will initiate the write-back of *out* arguments and task completion status. Write-back is asynchronous; The SPE places the task in write-back state and during write-back it tries to pre-fetch data for the next active tasks in the queue. When write-back finishes, the task turns to the complete state. If there are no more active tasks in the task queue or the data of the next task has arrived, the next task starts execution. Multiple write-backs and pre-fetches might be outstanding and being overlapped with task execution.

Task completion: When a task completes, the SPE sends its completion status to the SPE's completion queue that is placed in main memory. The transfer of the completion status is ordered with respect to the write-back of the task's

results. The PPE polls these queues for completed tasks from each SPE. A task completion informs the PPE that an entry in the corresponding task queue is now free and that it can issue a new task. Thus, the PPE polls the completion queue when: (a) there is no more space in the task queues (b) the application waits on task completion for synchronization purposes. We indicate the first type of wait as *queue stall* time and the second as *synchronization wait* time.

The SPE can signal completion via a mailbox register or DMA transfer. Although the writing of the mailbox register incurs very low overhead in the SPE, it requires the PPE to poll the status of the register via remote loads that generate unnecessary EIB traffic. Thus, it is preferable for the SPE to use a DMA transfer to a memory location. Then the PPE can poll using cached loads. In this case, to avoid the cache invalidation and the resulting off-chip transfer, we use the "putqlluc" atomic DMA command to unconditionally update the PPE's cache. Finally, each completion queue entry is padded and aligned to cache line boundaries (128-bytes) for optimal DMA performance.

Based on these observations the main task management operations in *TPC* are shown in Figure 1(a). Overall, task management operations in *TPC* require only on-chip transfers. Next, we discuss our evaluation methodology and the applications we use.

3 Experimental Platform and Methodology

In our experiments we use a Playstation3 (PS3) game console system, equipped with a 3.2 GHz Cell processor and 256 MBytes of main memory. On the PS3, applications are allowed to access only six of the eight SPEs in the Cell processor.

In our evaluation we use both application kernels as well as full applications. The applications we use are: FFT and LU from SPLASH-2 [18], PBPI [7], and an H.264 Encoder [1]. We re-implemented LU and FFT with single precision floating point arithmetic, replacing the original double precision version, because the SPEs exhibit significantly higher performance with single precision floating point operations. Using single-precision floating point arithmetic results in higher communication to computation ratios and a more realistic evaluation.

LU: We maintain the original algorithm [18] and modify the execution control structure of LU to employ a single *master* and multiple *worker* cores. Phases between barriers in the original code are translated to tasks, with the master core waiting between phases for all tasks to complete. Porting LU to *TPC* essentially involves converting three compute-intensive functions to *TPC*: `bdiv()`, `bmod()`, and `bmodd()`. The main modification to these functions is the identification of shared memory accesses in their body and conversion of these updates to a task argument list. We use the contiguous blocks version of LU from the SPLASH-2 suite, therefore we avoid stride arguments.

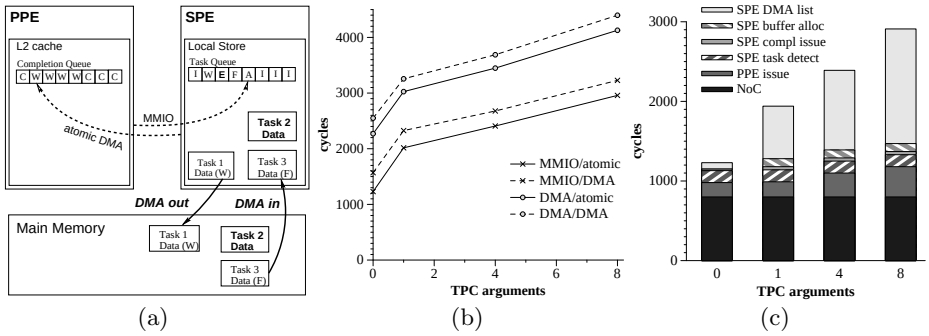
FFT: The SPLASH-2 version of FFT uses a six-step algorithm that involves alternating phases of transpose and FFT calculations. In our porting, we re-

organize the code as follows: We merge steps two and three in a single asynchronous call to reduce data transfers, as both steps modify the same data. We modify the transpose step to transpose the matrix in place. We split the original matrix into blocks in a similar way as the original SPLASH-2 FFT but we use the local storage of SPEs as an intermediate buffer to transpose each block. Although certain aspects of porting FFT to *TPC* require understanding the existing code beyond syntactic modification, eventually the changes required are simple structural changes that do not require modifying data structures or re-writing the code. Similarly to LU, this is because FFT has been optimized to avoid fine-grain accesses to shared memory, which hinder scalability in traditional shared memory multiprocessors.

PBPI: Parallel Bayesian Phylogenetic Inference [7] constructs phylogenetic trees from aligned homologous DNA sequences. The original code is implemented in MPI. The TPC version of PBPI aims at exploiting fine-grain parallelism in each MPI process by using TPC tasks. We use TPC tasks to parallelize three loops that compute the likelihood on each node of the phylogenetic tree. The three loops are separated by barriers. Each node has enough data to produce tasks for all SPEs with argument sizes that result in efficient DMA transfers. Additionally, loop portions that each task executes are unrolled and vectorized. We introduce a user-defined parameter that specifies task size in terms of loop iterations. We implement a static load balancing scheme to ensure that all SPEs execute the same number of tasks, while adjusting their size to be as close as possible to the user-defined size.

H.264 Encoder: A typical H.264 video encoder consists of three components: prediction, transformation, and entropy encoding [17]. We port an existing parallel encoder, x264 [1], originally written for shared-memory multiprocessors, to the Cell using *TPC*. Although parallelization of x264 can occur at different granularities, the limited on-chip memory leads to parallelization at the macro-block level, which allows a single frame to be processed in parallel by all SPEs. This requires satisfying macro-block dependencies in an antidiagonal-based manner [16]. We port the analyze, encode and Context-based Adaptive Variable Length Coding phases to the SPEs, leaving the rest of the code on the PPE. This allows for parallelizing about 85% of the serial execution time. Finally, we vectorize certain kernels of motion estimation for the SPEs: sum of absolute differences, sum of absolute transformed differences, and pixel average. The remaining application code that runs on the PPE is vectorized using the PowerPC Altivec extensions.

Kernels: We port SAXPY, SGEMV, and CONV2D directly from their original implementation in Sequoia [6] to *TPC*, with no structural or algorithmic modifications in the kernel code. SAXPY and SGEMV are communication bound. CONV2D is computation bound. CONV2D uses convolution to apply a mask to a 2D image. The initial image of size $M \times N$, is decomposed into a set of parallel 2D convolution subproblems, each computing a non-overlapping region of the output image of size $S \times T$.



(a) (b) (c)

Fig. 1. (a) *TPC* runtime operations. (b) Null task latency for the different initiation and completion mechanism. (c) Null task round-trip breakdown for MMIO initiation and atomic DMA completion.

For each application, we present execution time breakdowns for both the PPE and the SPEs. We break down the execution of the PPE in three parts: time spent in the *TPC* runtime, time waiting for SPEs to complete, and time spent in application code. SPE breakdowns consist of task compute time, library time (including data transfer time), and idle time. Also, as a reference point, we show application execution time for a single PPE, where this is possible. Finally, in this work we assume that the code to be executed by each task is already present on the target SPE and PPE distributes tasks round-robin across SPEs.

4 Experimental Results

4.1 Basic task overheads

In this section we examine the basic overheads associated with task operations in *TPC* using null tasks, which perform no computation. Furthermore, we set the task queue size to a single entry to avoid overlapping of runtime overheads.

In Figure 1(b) we see the total latency for initiation and completion of a null task. We evaluate two methods for initiation and two methods for completion. The PPE can initiate a *TPC* task with remote stores directly to an SPE's local storage. We refer to this mechanism as MMIO initiation. Alternatively, the PPE can build the task descriptor locally in its L2 cache and initiate a DMA command in the SPE's DMA controller to fetch the descriptor to the local storage of the SPE. We refer to this mechanism as DMA initiation. Completion status from SPE can be sent with a simple DMA command or an atomic DMA command. We refer to these methods as DMA and atomic (ACU) completion respectively. We use zero-byte arguments to show how the overhead of the runtime varies with the number of *TPC* arguments without including the DMA transfer costs that are not affected by the design of the runtime system. First, we see that minimum round-trip latency is about 1230 cycles or 385 ns. Second, we note

that using MMIO for task initiation and the ACU for task completion results in the lowest overhead. Using DMA instead of MMIO for task initiation adds about 1000 cycles, whereas replacing the ACU with regular DMA for task completion adds about 200 cycles.

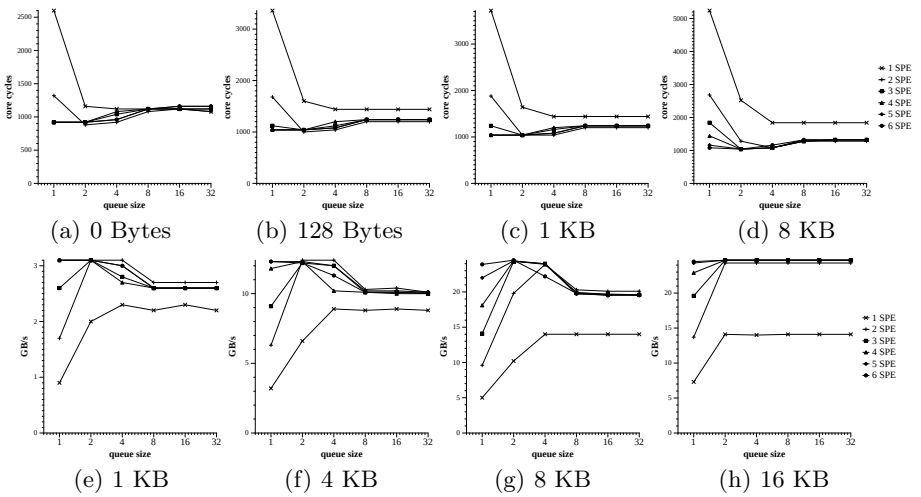
Figure 1(c) shows the breakdown of null-task latency in the best case, when using MMIO and atomic DMA, for a varying number of zero-byte arguments. PPE initiation includes building the task descriptor and issuing the remote stores. The PPE initiation overhead increases slowly with the number of arguments from 180 to 380 cycles. The EIB round-trip latency is about 800 cycles. We should note that both PPE and SPE are dual-issue, in-order processors. This makes them vulnerable to register dependencies and poor instruction scheduling. For this reason, in the PPE, we use a separate `tpc_callN()` function for tasks with N arguments. In these versions of `tpc_callN()` functions, as the number of *TPC* arguments is fixed, we perform loop-unrolling and appropriate instruction scheduling to help the compiler produce more efficient code. However, we can not apply the same method for run-time operations in the SPE as they depend not only on the number of *TPC* arguments but also on the types of these arguments. We expect that compilers will be able to deal with these issues when generating code for the TPC runtime.

The SPE portion of the round-trip overhead, excluding DMA transfers for task data, involves four steps: SPE task detection recognizes the user function to be invoked and sets up internal structures; SPE DMA list builds the DMA list elements for input and output arguments, as described in the task descriptor; SPE buffer allocation allocates the required space for task data in the local storage; SPE completion builds the completion status and issues the atomic DMA command to signal completion. The processing of tasks in the SPE is dominated by the time needed to create the DMA list for fetching inputs and writing back results. The cost for a single argument is 650 cycles and increases to 1450 cycles for eight arguments. On the other hand, the time needed for task detection, buffer allocation, and issuing the DMA for the completion status is about 280 cycles and is not affected by the number of *TPC* arguments.

4.2 Impact of Queue Size

Figure 2 shows the impact of task queue size on null task latency and throughput, when using a single argument of varying size, with the generic version of the `tpc_call()` function. The minimum average latency for null task with a zero-byte argument is about 900-1000 cycles, when using more than two SPEs and queue size of two or four, due to overlapping of tasks on multiple SPEs. Larger queue sizes increase the average latency to about 1100 cycles when using more than one SPEs. We observe similar behavior in the case of non-zero arguments for null tasks. However, latency increases when queue size increases above four.

When looking at throughput in Figure 2, we see that a single argument of 8 KBytes or more can reach maximum throughput with queue sizes of two or more on three or more SPEs. A queue size of one can reach maximum throughput only when using all six SPEs. An argument size of 4 KBytes approaches half of the



(a) 0 Bytes (b) 128 Bytes (c) 1 KB (d) 8 KB

(e) 1 KB (f) 4 KB (g) 8 KB (h) 16 KB

Fig. 2. Impact of queue size on null-task latency (top) and throughput (bottom) for different number of *TPC* arguments

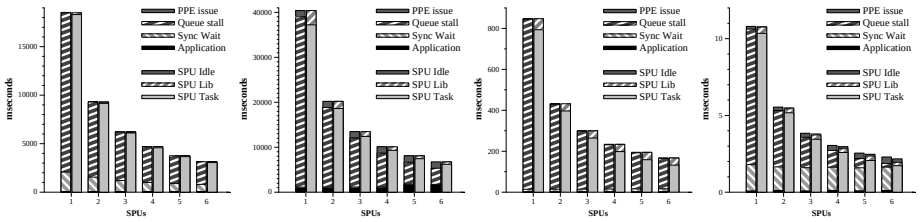
maximum throughput for two SPEs and a queue size of four. The maximum throughput achieved with a single 1-KByte argument is about 3 GBytes/s (12% of the theoretical peak of 25.6 GBytes/s) with four SPEs and a queue size of two or four.

Overall, we expect that a small task queue size of up to four will be enough for achieving all possible overlap of communication and computation in the SPEs.

4.3 Application Scaling

LU: Figure 3 shows LU execution time breakdowns for both PPE and SPEs with a $4K \times 4K$ input matrix, using 64×64 and 16×16 blocks. LU, though a shared memory application has already been optimized to avoid scattered, fine-grain accesses to shared data structures. For both block sizes, execution time scales with the number of SPEs. Maximum speedup for six SPEs is 5.98 and 5.87 for 16×16 and 64×64 blocks respectively. However, note that using 16×16 blocks is about 105% slower than using 64×64 blocks when using one SPE. With 64×64 blocks compute time dominates, as there are significantly fewer and larger DMA transfers and the larger task compute time allows the runtime to effectively pre-fetch future tasks.

FFT: Figure 3 shows the execution time breakdowns for the PPE and SPEs for 4M and 64K elements. The larger FFT problem size of 4M complex reals requires about 64 MBytes of memory. FFT exhibits good performance and scalability. For 4M FFT, *TPC* achieves speedup of 5.05 in 6 SPEs and for 64K FFT *TPC* achieves speedup of 5.1. The number of *TPC* tasks depends only on the problem



(a) LU,B=64 (b) LU,B=16 (c) FFT,4M (d) FFT,64K

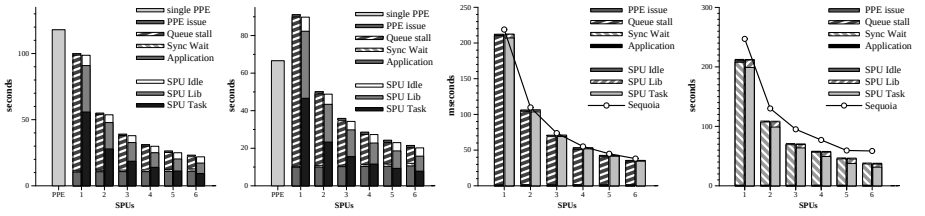
Fig. 3. LU and FFT execution times. LU uses $4K \times 4K$ matrix, with block sizes 64×64 and 16×16. FFT computes 4M and 64K complex elements respectively

size, as the task granularity is fixed to a single row of the matrix. For the 4M problem size there are enough tasks to fill the task queue of every SPE. On the other hand, the 64K problem size does not create enough tasks to take advantage of task pre-fetching and incurs higher sync wait times on the PPE for more than four SPEs. On the SPE side, compute time dominates the total execution time, whereas argument transfer overheads are less than 4% and 7% for the 64K and 4M problem sizes respectively. Overall, scalability of FFT is currently limited mainly by the transpose steps of the algorithm. Table 1 shows that for 4M FFT the computation and transpose times scale differently. Computation time alone scales by a factor of 5.98 over 6 SPEs while the transpose time scales only by a factor of 1.93 because memory throughput is saturated for more than two SPEs. However, the time of the transpose step varies between 8.8% and 23% of the total execution time and has a lower impact on scalability.

H.264 Encoder: In our experiments we use a number of full high definition (1920×1088) video inputs taken from the HD-VideoBench [3]. Although the size of a single macro-block is the same for every task, the amount of computation involved in processing is different. Figures 4(a) and 4(b) present execution time breakdowns for both PPE and SPEs for two different videos. Each video has different computational complexity. We have set the queue size to two slots for this application due to the high memory requirements for code in the SPEs (about 150 KBytes of code). In our experiments we use three B-frames and one reference frame with 128×128 motion vector search window. The achievable speedup depends on the complexity of the input video sequence, since the input stream affects the required computations. Overall, using 6 SPEs results in a speedup of up to 5.0 compared to the initial version of the encoder running on the PPE.

4.4 Comparison to Sequoia

Finally, we compare *TPC* to Sequoia using the SAXPY, SGEMV, and CONV2D kernels that come with Sequoia. We port them to *TPC* using the same computation functions and the same data partitioning schemes. We also port PBPI to

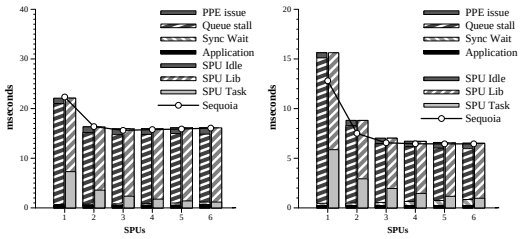


(a) x264:riverbed (b) x264:blue sky (c) CONV2D (d) PBPI

Fig. 4. *TPC* execution time breakdowns for x264, 2D convolution and PBPI

Table 1. FFT speedup

SPEs	Transpose fraction	Speedup		
		Computation	Transpose	Total
1	08.8%	1.00	1.00	1.00
6	23.0%	5.98	1.93	5.05



(a) SAXPY (b) SGEMV

Fig. 5. SAXPY and SGEMV breakdowns

TPC and compare with its Sequoia implementation [15]. SAXPY and SGEMV are both communication bound kernels and saturate memory bandwidth with more than two or three SPEs. For CONV2D, in order to achieve better DMA performance, we split the $4K \times 4K$ image into 32×64 blocks where each task processes one block. The matrix is constructed in row-wise form, therefore we use stride arguments. Computation time dominates the SPE execution time in CONV2D. Figures 5 and 4(c) show that *TPC* and Sequoia scale similarly with both communication bound and compute bound kernels. The performance difference between *TPC* and Sequoia for SAXPY and SGEMV when the kernels use more than one SPEs is about 3%. *TPC* performs about 5% better than Sequoia in CONV2D due to better overlapping of DMA transfers in the *TPC* runtime.

For PBPI we use various task sizes in *TPC*. We find that tasks with argument sizes larger than 4 KBytes reach almost maximum speedup at queue sizes of 4 or higher. Figure 4(d) shows that with 6 SPEs we achieve a maximum speedup of 5.6 while Sequoia achieves a maximum speedup of 4.2 with the same setup. *TPC* benefits from dynamic task execution and better load balancing in PBPI.

5 Related Work

The introduction of multi-core processors in mainstream computing environments has given rise to numerous proposals and associated research efforts on

parallel programming models. We concentrate our discussion of related work on task-level parallel programming models targeting heterogeneous multi-core processors with explicitly managed local memories and cover briefly other related work due to space considerations.

Sequoia [6] is a programming language which relies on explicit data accesses and is similar to *TPC* in that locality is exploited through annotation of data with in-out clauses. Sequoia follows a static execution model where the programmer statically optimizes the mapping of data and tasks relatively to the memory hierarchy. *TPC* implements a dynamic execution model where the programmer expresses parallelism and locality without considering the mapping of tasks and data to cores. *TPC* is optimized towards achieving low-overhead dynamic task management mechanisms in order to exploit fine-grain task-level parallelism, whereas Sequoia is optimized for explicit, static locality management.

CellSs [13] is a programming model for expressing task-level parallelism with code annotations. Contrary to *TPC*'s RPC-style programming model, CellSs uses compiler directives to annotate tasks and data with in-out clauses. The distinguishing feature of CellSs is the use of a helper thread that dynamically analyzes dependencies between tasks and schedules tasks dynamically after resolving their input dependencies. Dynamic dependence analysis incurs high overhead, which can be amortized if the analysis can increase the degree of available parallelism. *TPC* does not perform runtime dependence analysis although this is not precluded by its design. *TPC*'s task queues enable aggressive lookahead optimizations, such as pre-fetching via multi-buffering, similarly to CellSs. On the other hand, CellSs's scheduling model assumes coarse task granularity to mask the overhead of runtime data dependence analysis, whereas *TPC* targets fine-grain task-parallel execution. *TPC*'s measured task initiation/completion times are one order of magnitude lower than those currently reported for CellSs.

OpenMP has been extended to support task parallelism [12]. OpenMP tasks require the programmer to specify only the code region that will execute in parallel as a task. Instead, *TPC* requires specification of both code and data accessed by the task. The *XLC* [11] compiler for the Cell offers an OpenMP abstraction for loop level parallelism, using *DBDB* [9]. XLC splits loop iterations across SPEs and predicts statically the ideal number of grouped iterations in order to overlap communication with computation. On the other hand, *TPC* generates tasks dynamically and uses task queues to overlap DMA transfers of upcoming tasks with current task execution. Our evaluation shows that *TPC* is successful in hiding DMA latencies. Furthermore, *TPC* maps non-contiguous accesses always to DMA-list elements to minimize DMA initiation overheads in the runtime. On the other hand, DBDB uses an analytical model to predict whether those accesses should be mapped to a single DMA, including unnecessary data, multiple individual DMAs, or a single DMA list. The authors of DBDB find that DMA lists offer the best performance in most applications. Overall, *TPC* aims at minimizing the *runtime overhead* for preparing and initiating task and data transfers on both the PPE and SPEs, whereas DBDB aims at optimizing data transfer *time*.

Related work targeting heterogeneous multi-core architectures outside the context of task-level parallel programming models includes data-parallel programming models, such as RapidMind [10], and libraries for expressing and managing communication between heterogeneous components, such as IBM ALF and DaCs [5]. Other commonly used programming models for shared-memory multiprocessors, such as Cilk [4], do not provide support for heterogeneous systems with explicitly managed local memories, although there are ongoing efforts for extending these models to support heterogeneous systems in the future.

6 Conclusions

We present *Tagged Procedure Calls* (*TPC*) a programming model for the Cell processor, designed to exploit fine-grain parallelism and reduce programmer effort for scaling to large numbers of cores. *TPC* requires the programmer to annotate programs at the procedure level for specifying parallel tasks and their data accesses.

TPC implements task management using only on-chip operations for task creation, initiation, assignment, and completion. *TPC* achieves null task initiation and completion in 385 ns on the Cell, which is close to the round-trip DMA latency. We find that applications previously implemented and optimized for shared-memory multiprocessors can be ported with manageable effort that involves mostly mechanical code changes and achieve high parallel efficiency using *TPC*. In addition, our results show that *TPC* enables the exploitation of additional fine-grain parallelism on-chip in applications parallelized previously at coarse granularity. Through a comparison with the Sequoia programming language and its runtime we demonstrate that *TPC* performs competitively to existing task-level parallel programming frameworks.

Finally, based on our experience with *TPC*, runtime support for future CMPs will need to deal with three additional, broad issues: Mapping of the natural task sizes of applications to fine-grained tasks for memory efficiency, scheduling of fine-grained tasks, and code management. We believe that addressing these issues at the runtime and architectural levels can result in efficient and scalable task-based programming models for future CMPs.

Acknowledgments. We would like to thank Manolis Katevenis for discussions at early stages of the work. We are thankful to the Barcelona Supercomputing Center (BSC) and the Polytechnic University of Catalonia (UPC) for making available the QS21 boards for performance debugging experiments. Finally, we thankfully acknowledge the support of the European Commission through the SARC IP (Contract No. SARC-27648), HiPEAC NoE (Contract No. IST-004408 and IST-217068) and the MCF IRG project I-Cores (Contract No. IRG-224759)

References

1. VideoLAN - x264, http://www.videolan.org/developers/x264.html.

2. J. Abellán, J. Fernández, and M. Acacio. Characterizing the Basic Synchronization and Communication Operations in Duall Cell-Based Blades. In *Proc. of the 8th International Conference on Computational Science*, pages 456–465, June 2008.
3. M. Alvarez, E. Salami, A. Ramirez, and M. Valero. HD-VideoBench. A Benchmark for Evaluating High Definition Digital Video Applications. In *Proceedings of the IEEE Workload Characterization Symposium*, pages 120–125, 2007.
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. In *PPOPP'95: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 1995.
5. C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating Computing with the Cell Broadband Engine Processor. In *CF '08: Proceedings of the 5th ACM Conference on Computing frontiers*, pages 3–12, 2008.
6. K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of ACM/IEEE Supercomputing'2006*, 2006.
7. X. Feng, K. W. Cameron, C. P. Sosa, and B. E. Smith. Building the Tree of Life on Terascale Systems. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, pages 1–10, Mar. 2007.
8. H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, 2005.
9. T. Liu, H. Lin, T. Chen, K. O'Brien, and L. Shao. DBDB: Optimizing DMA Transfer for the Cell BE Architecture. In *ICS*, pages 36–45, 2009.
10. M. D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In *GSPx Multicore Applications Conference*, Santa Clara, CA, Oct. 2006.
11. K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. *International Journal of Parallel Programming*, 36(3):289–311, 2008.
12. OpenMP Architecture Review Board. Draft version 3.0 for public comments. http://www.openmp.org/mp-documents/spec30_draft.pdf, Jan 2008.
13. J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5):593, 2007.
14. A. Rico, A. Ramirez, and M. Valero. Available Task-level Parallelism on the Cell BE. *Scientific Programming*, 17:59–76, 2009.
15. S. Schneider, J.-S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos. A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies. In *PPOPP'09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 131–140, 2009.
16. E. van der Tol, E. Jaspers, and R. Gelderblom. Mapping of H.264 Decoding on a Multiprocessor Architecture. In *Image and Video Communications and Processing*, 2003.
17. T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. volume 13 of *Circuits and Systems for Video Technology*, pages 560 – 576, July 2003.
18. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.