Computer Science Department

University of Crete

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

Master's Thesis

George Tzenakis

October 2009

Heraklion, Greece

University of Crete

Computer Science Department

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

Thesis submitted by

George Tzenakis

in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________________

George Tzenakis

Committee approvals: ______________________________

Angelos Bilas

Associate Professor, Thesis Supervisor

____

Dimitrios Nikolopoulos

Associate Professor

____

Evangelos Markatos

Professor

Departmental approval: ______________________________

Panos Trahanias

Professor, Director of Graduate Studies

Heraklion, October 2009

Abstract

Increasing the number of cores in modern CPUs is emerging as the main approach for improving system performance. A central challenge in this area is the runtime support that multi-core systems ought to use for sustaining high performance and scalability without, however, 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, such as the Cell processor. We present the design and implementation of *TPC* for the Cell and we examine how the runtime system can support task management functions with on-chip communication only, without requiring accesses to off-chip memory. Through minimizing off-chip transactions in the runtime, we achieve sub-microsecond task initiation latency —which represents an order of magnitude of improvement over existing task-parallel programming frameworks on the Cell– 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.

Supervisor professor: Angelos Bilas

Περίληψη

Η αύξηση του αριθμού των πυρήνων στους σύγχρονους επεξεργαστές έχει αναδειχθεί τα τελευταία χρόνια ως η κύρια μέθοδος αύξησης της επίδοσης των υπολογιστικών συστημάτων. Η μεγαλύτερη πρόκληση στον τομέα των πολυεπεξεργαστών είναι το τι υποστήριξη χρειάζεται από το περιβάλλον εκτέλεσης έτσι ώστε το σύστημα να διατηρεί υψηλό επίπεδο επιδόσεων και κλιμακωσιμότητα χωρίς ωστόσο να αυξάνεται δυσανάλογα η προσπάθεια που απαιτείται από τον προγραμματιστή. Σε αυτήν την εργασία παρουσιάζουμε το *Tagged Procedure Calls* (*TPC*), ένα περιβάλλον εκτέλεσης για υποστήριξη προγραμματιστικών μοντέλων που βασίζονται σε εργασίες και απαιτούν ρητή περιγραφή του τρόπου πρόσβασης στα δεδομένα, όπως ο επεξεργαστής *Cell BE*. Παρουσιάζουμε τον σχεδιασμό και την υλοποίηση του (*TPC*) στον *Cell* και εξετάζουμε πως το περιβάλλον εκτέλεσης μπορεί να υποστηρίξει την διαχείριση των εργασιών χωρίς να κάνει αναφορές στην εξωτερική μνήμη και να παραμένει πάντα εντός του ίδιου *chip*. Με την ελαχιστοποίηση των δοσοληψιών με την εξωτερική μνήμη στο περιβάλλον εκτέλεσης, καταφέρνουμε να έχουμε καθυστέρηση έναρξης εργασίας μικρότερη από ένα *microsecond* –το οποίο είναι μία τάξη μεγέθους λιγότερο σε σχέση με υπάρχουσες εργασίες για προγραμματιστικά μοντέλα που βασίζονται σε εργασίες για τον *Cell*– και ελάχιστη καθυστέρηση για την αρχικοποίηση και ολοκλήρωση μιας εργασίας 385 *ns*. Αξιολογούμε το *TPC* χρησιμοποιώντας διάφορους υπολογιστικούς πυρήνες και εφαρμογές, επιδεικνύοντας ότι το *TPC* επιτυγχάνει κλιμακωτή εκτέλεση προγραμμάτων που είχαν ήδη γίνει παράλληλα για πολυεπεξεργαστές διαμοιραζόμενης μνήμης, μπορεί να αξιοποιήσει παραλληλισμό με πιο αναλυτικές εργασίες για προγράμματα που είχαν φτιαχτεί για λιγότερο αναλυτικές εργασίες, και ότι αποδίδει ανταγωνιστικά σε σχέση με άλλα προγραμματιστικά μοντέλα που βασίζονται σε εργασίες που βελτιστοποιούν στατικά την τοποθέτηση των δεδομένων και των εργασιών.

Επόπτης καθηγητής: Άγγελος Μπίλας

Acknowledgments

I feel grateful to my supervisor, Angelos Bilas, for his valuable assistance and guideline in my academic steps in the field of Computer Science. The extensive discussions about my work and his wide knowledge have been of a great value for me.

A big thanks to Dimitris Nikolopoulos for his support and help to a major part of my work. Moreover, I would like to thank my friend and colleague, Konstantinos Kapelonis, for his help, especially in the beginning of my thesis.

My warmest appreciation to the following, past and current, members of the CARV Laboratory of the ICS/FORTH, whom I feel to be friends more than just colleagues: Mihalis Alvanos, Konstantinos Koukos, Giannis Kesapides, Giannis Klonatos, Kostas Chasapis, Markos Fountoulakis, Giorgos Nikiforos, Dimitris Tsaliagkos and Vangelis Mangas.

I would like also to thank my friends : Manolis Stiligkas, Manolis Zidianakis, Sokratis Kartakis, Efthimis Kartsonakis, Giorgos Iakovidis, Kwnstantinos Kapakiotis, Panayiota Ignatiou, Leyteris Sardis, Manolis Stratakis, Giorgos Kartakis and Giannis Papadakis. Their encouragement and discussions have been a great value of me.

Last but not least, I would like to thank my family, my parents Lefteris and Poppy and my sister Niki for their support and encouragement they provided me with.

George Tzenakis

Heraklion, October 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Current technology trends indicate that future high-performance, general-purpose and embedded, systems will be built out of heterogeneous chip multi-processors (CMPs) with large numbers of cores and tightly-coupled interconnects. However, scalable CMPs will require a large degree of parallelism in applications as well as dealing with heterogeneity, without significantly increasing programming effort.

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 both 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.

1.1 Thesis Contributions

In this thesis we introduce a runtime system for the Cell processor [8], *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, e.g. task initiation and completion that now become common-path operations. In this work we focus on better understanding and minimizing the 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 argument list size. 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 reported to just under 312 ns [2].

Then 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 simple and mechanical code changes, while *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 that perform static data layout and placement optimizations.

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents the design and implementation of *TPC* and its runtime system on the Cell processor. Chapter 3 presents the hardware and software environment we used for our performance evaluation. Chapter 4 presents our experimental results. In Chapter 5 we discuss the advantages of *TPC* over previous efforts and related work. Finally, we draw our conclusions in Chapter 7.

Chapter 2

Design

2.1 *TPC* Semantics

TPC uses program annotations to identify certain procedure calls as concurrent tasks. Annotations currently occur at the procedure level. Annotated procedures execute in the same or another core, as asynchronous tasks, with the current core continuing execution. Procedure arguments can be *in*, *out*, or *inout*. The issuing task can wait for tasks using point-to-point or barrier synchronization. When issuing an asynchronous task, the runtime returns a handle that can be used later on 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* functions have no return values and all arguments are passed by reference. Task arguments and their sizes are determined at runtime before task initiation. *TPC* supports continuous and fixed stride arguments. We expect that interfaces and constructs for specifying memory layout for task arguments will play an important role on programmer effort.

Figure 2.1(a) shows the basic *TPC* API and a simple usage example. Figure 2.1(b) shows code example for the LU kernel.

```
tpc_handle_t tpc_call(
    task_id, total_args,
    [arg1,size,IN/OUT/INOUT],
    ... );
tpc_wait(tpc_handle_t hdl);
tpc_wait_all();

mytask1(int *x);
mytask2(int *x, int *y);
mytask3(int *x, int *y, int *z);
int main(void){
  int i, x[N], y[N], z[N];
  for(i=0; i<N; i+=B) {
    tpc_call(mytask1, 1,
        x+i,B,INOUT);
  }
  tpc_wait_all();
  for(i=0; i<N; i+=B) {
    tpc_call(mytask2, 2,
        x+i,B,INOUT,
        y+i,N,IN);
  }
  tpc_wait_all();
  for(i=0; i<N; i+=B) {
    tpc_call(mytask3, 3,
        x+i, B, OUT,
        y+i, B, INOUT,
        z+i, B, IN );
  }
  tpc_wait_all();
}
```

(a)

```
bdiv(A, D);
bmodd(A, D);
bmod(A, B, C);

int main(void){

  for all diagonal blocks D {
    factor_diagonal_block(D);

    for all column blocks C {
      tpc_call(bdiv, 2,
          C, block_size, INOUT,
          D, block_size, IN );
    }

    for all row blocks R {
      tpc_call(bmodd, 2,
          R, block_size, INOUT,
          D, block_size, IN );
    }
    tpc_wait_all();

    for all interior blocks IB {
      tpc_call(bmod, 3,
          IB, blocks_size, INOUT,
          C, blocks_size, IN,
          R, blocks_size, IN, );
    }
    tpc_wait_all();
  }
}
```

(b)

FIGURE 2.1: *TPC* API (a) and code example for LU (b).

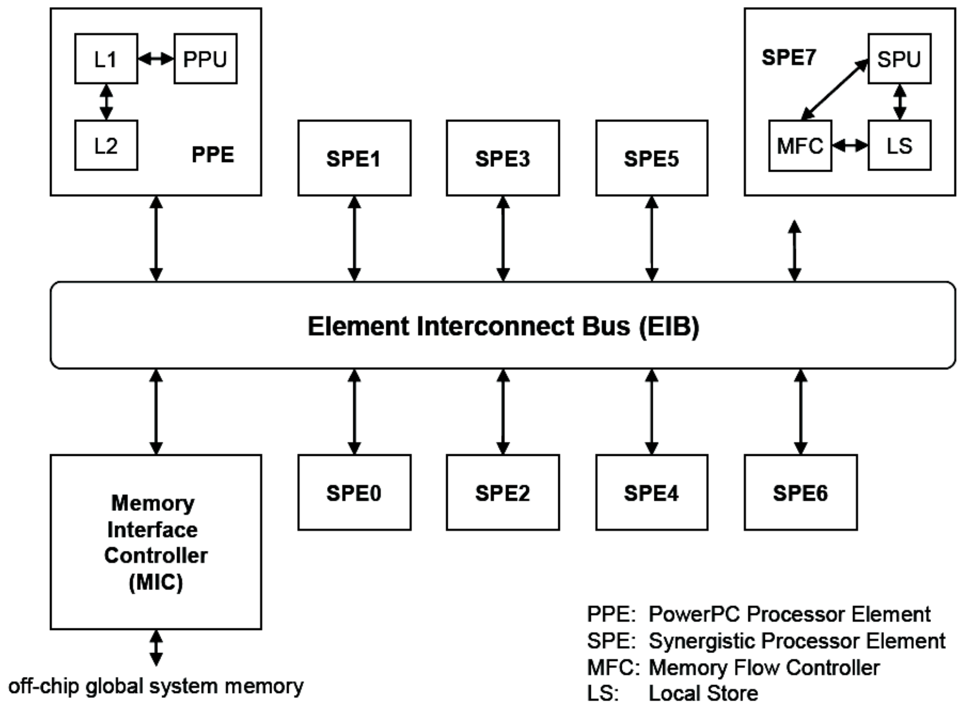

L1
PPU
L2
PPE
SPE1
SPE3
SPE5
SPE7
SPU
MFC
LS
Element Interconnect Bus (EIB)
Memory Interface Controller (MIC)
SPE0
SPE2
SPE4
SPE6
off-chip global system memory
PPE: PowerPC Processor Element
SPE: Synergistic Processor Element
MFC: Memory Flow Controller
LS: Local Store

FIGURE 2.2: Cell processor architecture.

2.2 Cell architecture

Figure 2.2 shows the architecture of Cell. The Cell processor [8] 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 (on-chip) memory without any other cache between this memory and the SPE core. There is also a global, off-chip memory. The PPE has a coherent memory hierarchy with two levels of cache prior to the single global external memory. Table 2.1 summarizes the different access mechanisms that exist between the various stores and processing elements. DMAs in the Cell are capable of scatter/gather functions and can have multiple (16 per SPE) outstanding transfers. Moreover,

	PPE	SPE(i)	SPE(j)
SPEI(i) local store	remote load/store, DMA	load/store	DMA
SPE(j) local store	remote load/store, DMA	DMA	load/store
Global Memory	cached load/store	DMA	DMA

Table 2.1: Cell processor transfer mechanisms

PPE can access the local stores of SPEs with remote load/stores as local stores are mapped to main memory address space. Finally, the PPE and SPEs can also communicate with messages via small mailboxes with and without interrupts. These options create a wealth of 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 parallel, bi-directional rings.

2.3 *TPC* Design and Implementation

The *TPC* runtime library consists of two 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, in our implementation 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 are placed in task queues for execution. There is one task queue per target, located in its local store. Figure 2.3(b) shows the structure of a task descriptor. The task descriptor contains the function id and the list of arguments (16 bytes per argument). 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 store of the corresponding SPE. In addition to the task queue, the runtime maintains a completion queue for each SPE in main memory. 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. The size of the completion status structure is padded to 128 bytes for optimal DMA performance, as discussed next.

An important architectural aspect for implementing a task-based runtime is the available mechanisms for communication among different memories and cores. DMAs in the Cell are capable of scatter/gather functions and can have multiple (16 per SPE) outstanding transfers. The PPE and SPEs can also communicate very short messages via small, word-size mailboxes, with and without interrupts.

Although DMA performance on the Cell has been thoroughly analysed 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 (DMA).

SPEs can communicate with the PPE via mailbox messages, DMA, or a variant of DMA using the Atomic Cache Unit (ACU). The ACU is intended for implementing atomic synchronization primitives in the global address space, among SPEs and the PPE. Every SPE has a small cache memory with four 128-Byte cache-lines which is used by ACU commands. Using special commands the SPE can initiate a DMA from this cache memory to the global address space. A simple, non-atomic DMA transfer writes results to main memory and invalidates the PPE's cache requiring off-chip accesses. Instead, an ACU-based DMA transfer remains in the SPE's cache and when PPE touches the cache line PPE's cache is updated, eliminating off-chip operations (PPE cache misses). This mechanism supports reserve-line (load-locked), conditional-store, and unconditional-store operations. Task completions require only the "putqlluc" command that updates atomically and unconditionally the main memory location with the SPE's data via an atomic DMA transfer.

2.3.1 Task initiation

Mailboxes and PPE-initiated DMAs are not appropriate mechanisms for initiating tasks. First, sending mailbox messages incurs in the PPE the same cost as remote stores 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 when posting the mailbox message. Using PPE-initiated DMA requires five remote store operations to special SPE registers that are mapped to the PPE. Then, the DMA controller of the SPE performs the actual DMA from external 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 results in on-chip traffic only but requires a number of MMIO stores from the PPE for each task. The second approach reduces the number of stores required at the PPE but introduces a DMA transfer in the SPE. This DMA transfer will involve only on-chip traffic, assuming the task descriptor is not evicted from the PPE cache.

In both cases the SPE can simply poll to local memory and there is no requirement for round-trip communication when posting a new task. 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 for a single task descriptor. Furthermore, PPE incorporates a six-slot store combining buffer, further reducing network latencies. 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. In our evaluation we examine both options for task initiation.

2.3.2 Task pre-fetching and execution

Once a new task has been posted to the SPE 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 argument and result 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, WRITEBACK, 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 local store, 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 place the task in the write-back state. During write-back, SPE tries again to pre-fetch data for the next active tasks in the queue. After the completion of the write-back, the next task starts execution as soon as its *in* data arrives.

2.3.3 Task completion

When a task completes, the SPE sends its completion status to the SPE 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 signifies to 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 (a) when there is no more space in any task queue and (b) when 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 two issues with the implementation of the completion queue are: (a) what is the impact of polling on the PPE side and (b) what is the overhead of signaling completion from the SPE. The SPE can signal completion via a mailbox register or DMA transfer (with or without the ACU). 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 loads that incur a round-trip overhead. 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 ACU, which allows the PPE cache to be updated by the SPE DMA. Finally, the SPE DMA performs best with addresses aligned at 128-bytes (cache line size), so each completion queue entry is padded and aligned to cache line boundaries.

Overall, task management operations in *TPC* require only on-chip transfers. Task and completion queues allow overlapping of task management overheads. Moreover, different task states allow overlapping of DMA and code execution.

Figure 2.3(a) shows an example of *TPC* execution. Initially, all task

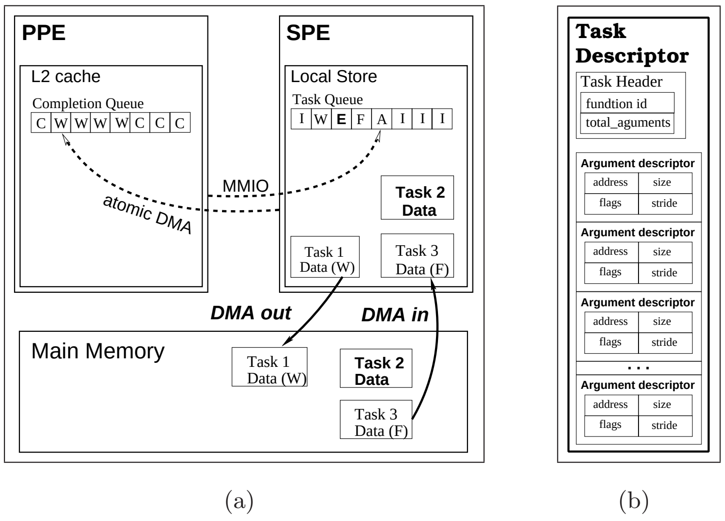


FIGURE 2.3: *TPC* runtime operations for task issue and completion (a), and task descriptor (b).

queue entries are in *invalid* (I) state and all completion queue entries are in *completed* state. PPE initiates a task by posting the task descriptor via remote stores which, sets the task queue entry to *active* (A) state and setting the corresponding entry of the completion queue to *waiting* (W) state. SPE polls task queue for *active* tasks and PPE polls completion queue until tasks are set back to *completed* state. Moreover, only one task can be in *execute* (E) state in the SPE, while there can be many tasks in *fetch* (F) or *write-back* (W) states.

Next, we discuss our evaluation methodology and the applications we use.

Chapter 3

Experimental Platform and Methodology

In our experiments we use a Playstation3 game console system, equipped with a 3.2 GHz Cell processor and 256 MBytes of main memory. On the PlayStation3, software is allowed access to only six of the SPEs. There is also 256 GBytes of global, off-chip memory.

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 implemented LU and FFT with both single and double precision floating point arithmetic, rather than the initial double precision version only, because the SPEs exhibit significantly higher performance with single precision floating point operations, resulting in higher communication to computation ratios and a more realistic evaluation.

Kernels: We ported 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 kernels have a very low computation to communication ratio and are communication 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 $S \times T$. CONV2D is computation bound.

LU: We maintain the original algorithm [18] and modify the execution control structure of LU to employ a single *master* and multiple *slave* cores. Phases between barriers in the original code are translated to tasks, with the master core waiting for completion between phases for all tasks to complete. Porting LU to *TPC* essentially involved converting three computational-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 store 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 TPC version preserves the original version and is integrated with MPI. The main purpose of the TPC port is to achieve fine-grain on-chip parallelism within an MPI task of the original PBPI implementation. Almost all execution time in PBPI is spent in 3 parallel loops which are parallelized using TPC tasks. The only adjustable parameter in these tasks is their input size, which also defines task granularity. We use this parameter to implement a static load balancing scheme for the application. We employ vectorization and we compare our implementation of PBPI against an equivalent implementation in Sequoia.

H.264 Encoder: A typical H.264 video encoder consists of three components: prediction, transformation, and entropy encoder [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 analyse and encode phases to the SPEs, leaving the rest of the code on PPE. This allows for parallelizing about 80%-85% of the serial execution time. We port most kernels responsible for motion estimation, sum of absolute differences, sum of absolute transformed differences, and pixel average. Finally, we vectorize certain kernels of motion estimation for the SPEs (sum of absolute differences, sum of absolute transformed differences, and pixel average), whereas the original code already includes vectorized versions for the PPE Altivec extensions.

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. Note that in general, the PPE wait time will be close to the average SPE breakdown, however, due to overlapping initiation of asynchronous calls with processing on the SPEs, the match is not always exact. 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 we always distribute tasks round-robin across SPEs.

Chapter 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.

In Figure 4.1(a) we see the total latency for initiation/completion of a null task. We evaluate two methods for initiation and two methods for completion. PPE can initiate a *TPC* task with remote stores directly to SPE's local store. We refer to this mechanism as MMIO initiation. Alternatively, 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 store of SPE. We refer to this mechanism as DMA initiation. Completion status from SPE can been sent with simple DMA command or using atomic DMA command. We refer to these methods as DMA and *atomic* completion accordingly. We use zero-byte arguments to show how the overhead of the runtime varies with the number of arguments without including the DMA initiation and data transfer costs that are not affected by the runtime system design. First, we see that minimum round-trip latency is about 1230 cycles or 385ns. Second, we note that using MMIO for task initiation and the atomic DMA for task completion results in the lowest overhead. Using

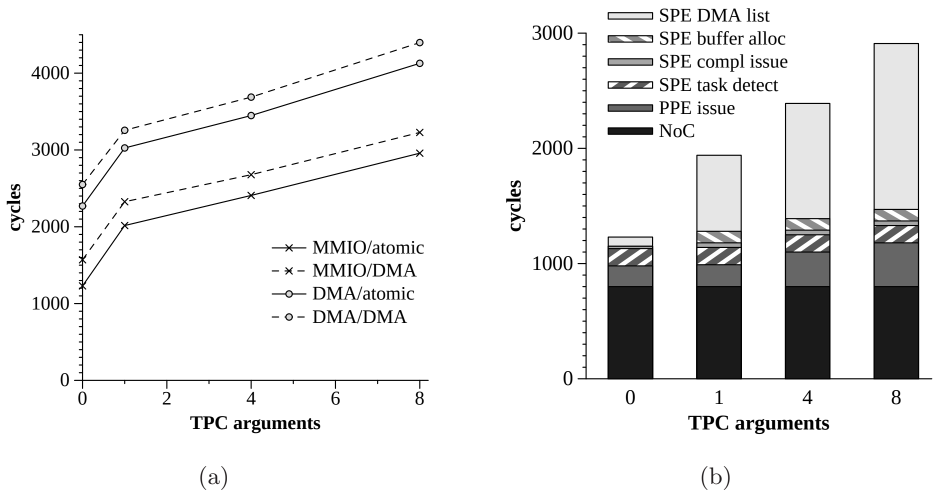


FIGURE 4.1: (a) Null task latency for the different initiation and completion mechanisms. (b) Null task round-trip breakdown for MMIO initiation and atomic DMA command.

DMA instead of MMIO adds about 1000 cycles whereas not using atomic DMA adds about 250 cycles.

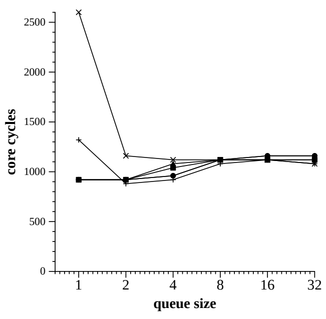
Figure 4.1(b) shows the breakdown of null-task latency in the best case (MMIO / atomic DMA) and for variable numbers of zero-byte arguments. PPE initiation includes the building of the task descriptor and issuing the remote stores. We see that PPE initiation overheads increase slowly with the number of arguments from 180 to 380 cycles. NoC 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 cannot follow the same approach for SPE as run-time operations in SPE depend not only on the number of *TPC* arguments but also depend on the types of these arguments.

The SPE portion of the round-trip overhead, excluding DMA initiation and NoC latency, involves four steps: Task detection recognizes the user function to be invoked and setups internal structures. SPE DMA list step builds the DMA list elements for input and output arguments, as described in the task descriptor. SPE buffer allocation step allocates the required space in local store and SPE completion issue step builds the completion status and issues the atomic DMA command. We see that processing 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 to build the DMA list. 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.

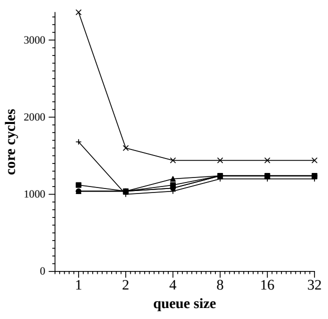
4.2 Impact of Queue Size

Figures 4.2 and 4.3 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 tpc_call() function. We see that the minimum average latency for null task with a zero-byte argument is about 900-1000 cycles, when using 2-6 SPEs and queue size of two or four, due to overlapping of tasks on multiple SPEs. Larger queue sizes increase average latency to about 1200 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.

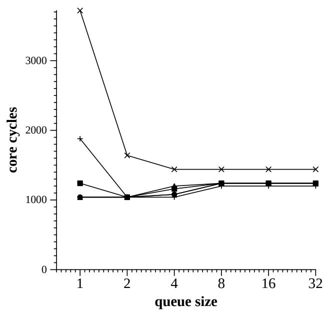
When looking at throughput, we see that a single argument of 8 KBytes or more can reach maximum throughput with queue sizes of two or more for three or more SPEs. A queue size of one can reach maximum throughput only when using all six SPEs. An argument size of 4 KBytes approaches half of the maximum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



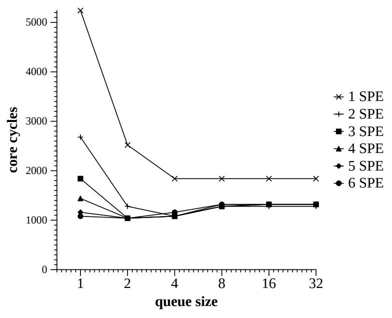
(a) 0 Bytes



(b) 128 Bytes

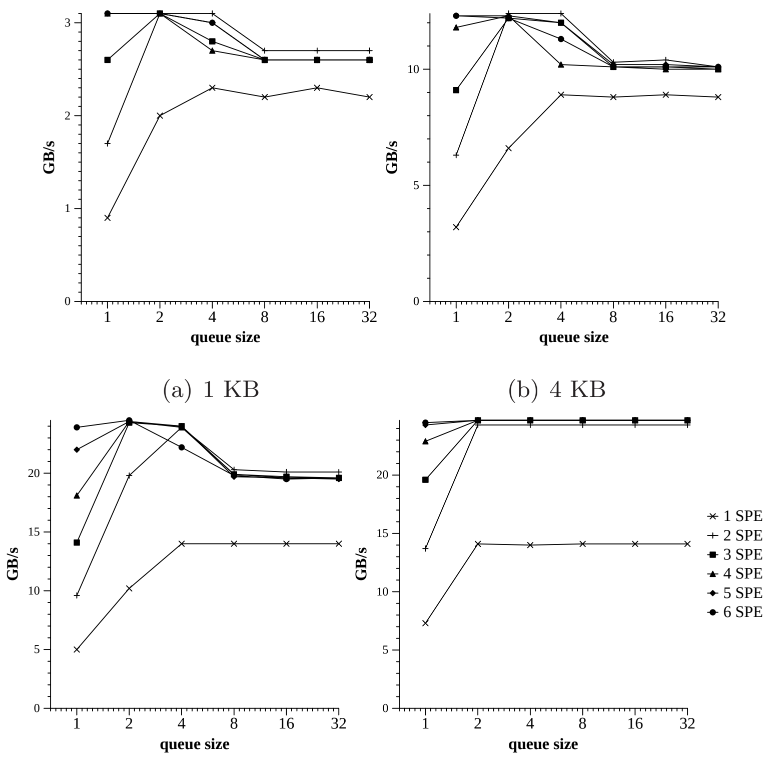


(c) 1 KB



(d) 8 KB

Figure 4.2: Impact of queue size on null-task latency for different argument sizes.



(a) 1 KB (b) 4 KB

(c) 8 KB (d) 16 KB

FIGURE 4.3: Impact of queue size on null-task throughput for different argument sizes.

maximum throughput achieved with a single 1-KByte argument is about 3 GBytes/s (12% of the theoretical maximum) with two 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: Table 4.1 shows statistics about the number of tasks as well as the total memory traffic for different problem sizes and block sizes. We note

LU	Block	# tasks	Memory Traffic	
size	size		Total(MB)	KB/task
$4K \times 4K$	16	5624960	34204	6.2
	64	89376	8274	94.8
$2K \times 2K$	16	707136	4284	6.2
	64	11408	1041	93.5

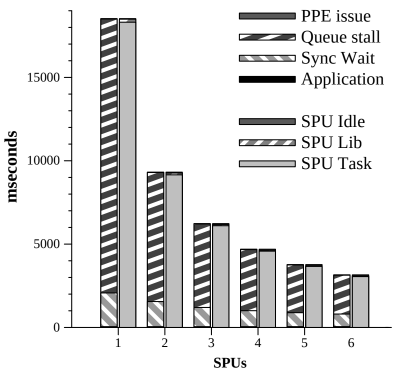
Table 4.1: LU execution statistics.

FFT	# tasks	Memory Traffic	
Size		Total(MB)	KB/task
64K	620	8.2	13.5
256K	1432	32.3	23.1
1024K	3632	128.9	36.3
4096K	10336	514.5	51.0

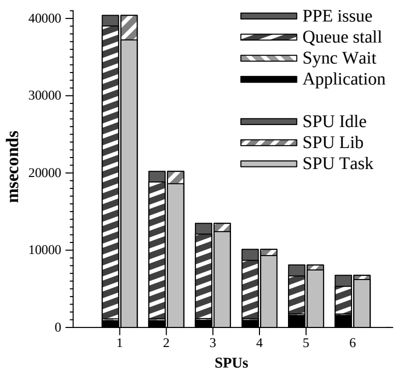
Table 4.2: FFT execution statistics.

that the total number of task increases proportionally to the problem size. On average, the amount of data passed to and returned by each task is between 6.2 and 94.8 KBytes depending on the block size. LU, although a shared memory application, has already been optimized to avoid scattered, fine-grain accesses to shared data structures.

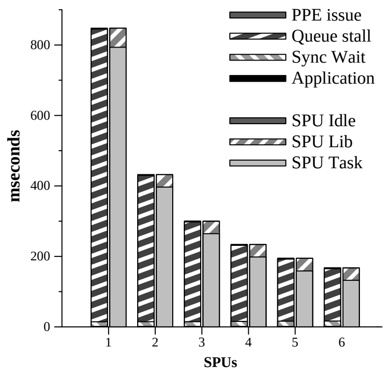
Figure 4.4 shows LU execution time breakdowns with 64×64 and 16×16 block sizes for both PPE and SPEs. We see that for both block sizes, execution time scales with the number of SPEs. Maximum speedup for six SPEs is 5.98 and 5.87 for 16×16 and 64×64 blocks respectively. However, note that using 16×16 blocks is about 115% slower than using 64×64 blocks for the same problem size when using one SPE. With 64×64 blocks compute time dominates, as there is significantly fewer and larger DMA



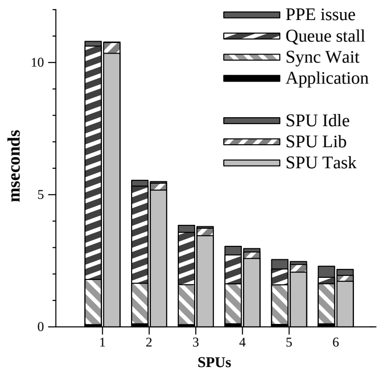
(a) LU, B=64



(b) LU, B=16



(c) FFT, 4M



(d) FFT, 64K

FIGURE 4.4: 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.

transfers and the larger task compute time allows the runtime to effectively pre-fetch future tasks.

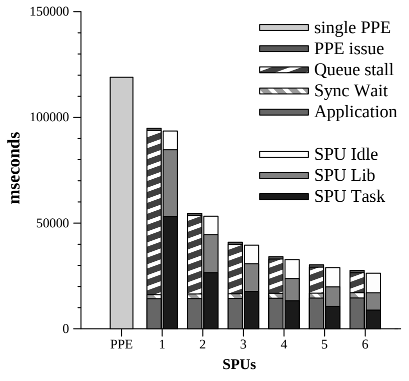
FFT: Table 4.2 shows FFT statistics about the number of tasks and size of transfers during execution for various problem sizes. The larger FFT problem size of 4M complex reals (single precision) requires about 64 MBytes of memory. The number of *TPC* tasks depends only on the problem size, as the task granularity is fixed to a single row of the matrix. Figure 4.4 shows the execution time breakdowns for the PPE and the SPEs for 4M and 64K

SPEs	Total time (ms)	Transpose fraction	Computation speedup	Transpose speedup	Overall speedup
1	847.6	8.8%	1.00	1.00	1.00
6	167.8	23.0%	5.98	1.93	5.05

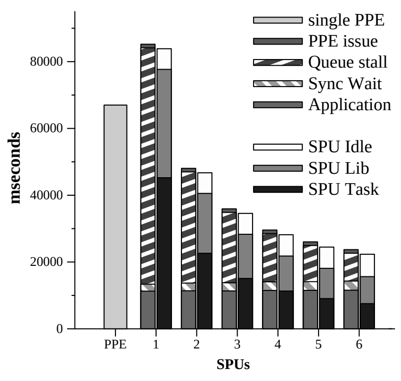
Table 4.3: FFT execution time break down and speedup over 6 SPEs for 4M complex reals. Speedup is measured for overall execution time, for computation time alone and for transpose time alone.

elements. We see that FFT exhibits good performance and scalability. For 4M FFT, *TPC* achieves speedup of 5.2 over 6 SPEs and for 64K FFT *TPC* achieves speedup of 5.1. For the 4M problem size there are enough tasks to fill the task queue of the SPEs. On the other hand, the 64K problem size creates 256 tasks during the computation phases and 36 tasks during the transpose phases. Thus, FFT does not create enough tasks to take advantage of task pre-fetching and incurs higher sync wait times for the PPE. 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 4.3 shows that for the 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 over 6 SPEs. However, the transpose step varies between 8.8% (one SPE) and 23% (six SPEs) of the total execution and has a lower impact on application scalability.

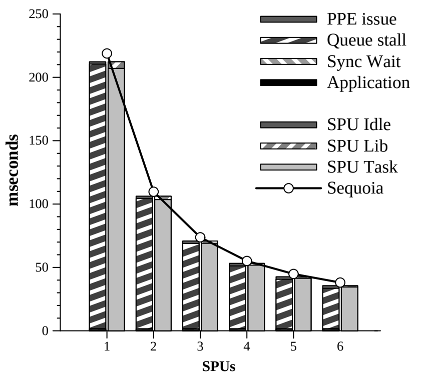
H.264 Encoder: In our experiments we use a number of full high definition (1920×1088) video inputs taken from the HD-VideoBench [3]. Although the size of a single macro-block is the same for every task, the amount of computation involved in processing it is different. Figures 4.9(a) and 4.9(b)



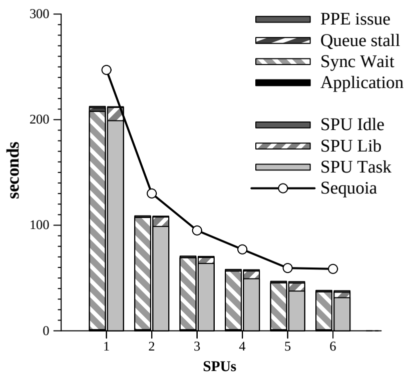
(a) x264:riverbed



(b) x264:blue sky



(c) CONV2D



(d) PBPI

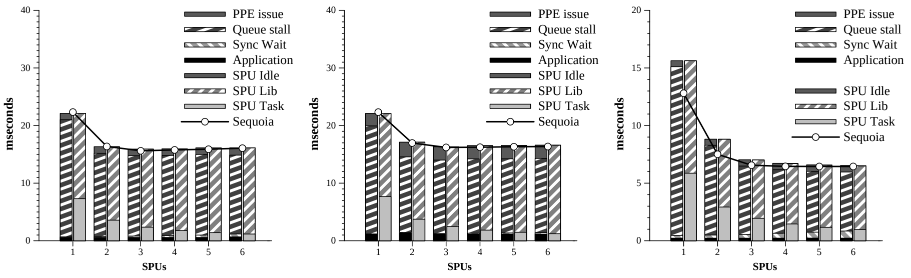
FIGURE 4.5: *TPC* execution time breakdowns for x264, 2D convolution and PBPI.

present execution time breakdowns for both PPE and SPEs for three different videos. Each video has different computation 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 130 KBytes of code). In our experiments we use three B-frames and one reference frame with 48×48 maximum motion vector search range. Finally, for the entropy encoding we use the Context-based Adaptive Variable Length Coding. The achievable speedup depends on the complexity of the input video sequence, since the input stream affects the computation to communication ratio. Overall, using 6 SPEs results in a speedup of about 2.9 for the *blue sky* stream and 4.2 for the *riverbed* stream, 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 *TPC* and compare with its Sequoia implementation [15]. SAXPY is a communication bound kernel that performs only two floating operations for every three floats. SGEMV is similar to SAXPY but multiplies a matrix with a vector rather than multiplying two vectors. The output of each task is only a few floats. Figure 4.6 shows that *TPC* and Sequoia scale similarly with both communication bound and compute bound kernels. For less than 32 floats, SGEMV performs transfers that are not cache aligned, causing serious DMA performance penalties in the *TPC* runtime. The input we use for CONV2D is a $4K \times 4K$ matrix of single precision float numbers. Matrix is constructed in row-wise form, therefore we use stride arguments for element blocks. In order to acquire better DMA performance, we split the matrix into 32×64 element blocks where each task processes one block. The performance differences between *TPC* and Sequoia for SAXPY and SGEMV kernels for more than one SPEs are under 3%. Figure 4.8(c) shows that computation time dominates the SPE execution time in CONV2D. Sequoia performs about 7% worse than *TPC* 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 4.5(d) shows that with 6 SPEs we achieve a maximum speedup of 5.6 while Sequoia achieves a maximum speedup of



(a) SAXPY, B=4K (b) SAXPY, B=2K (c) SGEMV

FIGURE 4.6: PPE and SPE execution time breakdowns for 24M element vectors for SAXPY and SGEMV with 32M element matrix.

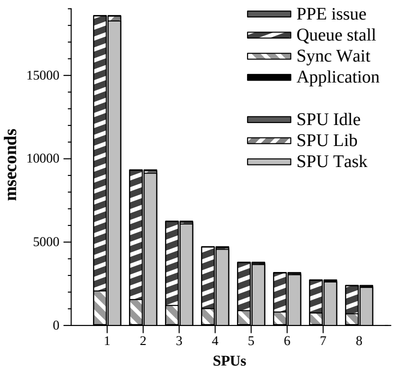
4.2 with the same setup. *TPC* benefits from dynamic task execution and better load balancing in PBPI. Overall, contrasting *TPC* to Sequoia using kernels and applications shows that both runtime environments seem to scale similarly.

4.5 Runs on QS20 BladeCenter

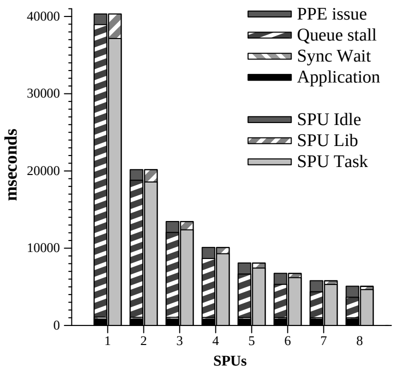
For complete evaluation of *TPC* runtime, we made additional runs for the applications mentioned previously on QS20 BladeCenter machines. In QS20 the application has access to all 8 SPEs of the Cell processor. Computational resources on QS20 are identical to those of PS3, but there are some minor differences in the configuration of the memory interface. The Cell processor runs at 3.2 GHz and includes 1 GByte of RAM.

In Figure 4.7, we can see that LU and FFT performs very close to PS3, with QS20 being better by just 1.5% for the same number of SPEs. Both LU and FFT scale naturally, achieving speedup of 7.73 and 6.96 accordingly over 8 SPEs.

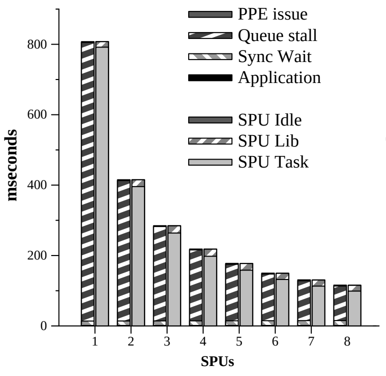
Figure 4.8 shows execution time breakdowns for SAXPY and CONV2D. CONV2D is computation bound kernel, achieving speedup of 7.94 over 8 SPEs. On the other hand, SAXPY is communication bound and cannot



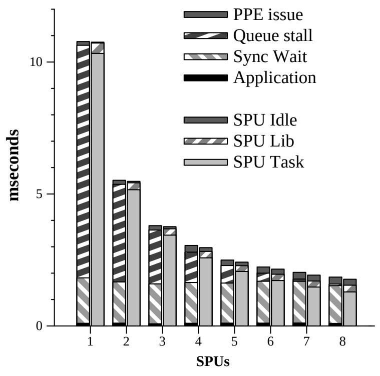
(a) LU, B=64



(b) LU, B=16



(c) FFT, 4M

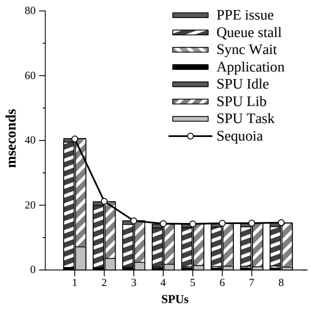


(d) FFT, 64K

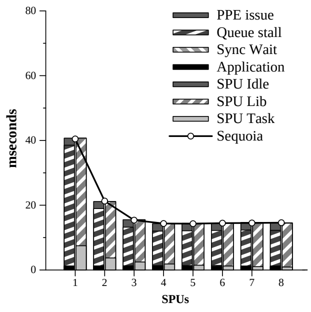
FIGURE 4.7: LU and FFT execution times. LU uses $4K \times 4K$ matrix, with block sizes 64×64 and 16×16. FFT computes 4M and 64K complex elements respectively. Runs on BladeCenter

achieve speedup more than 2.78 over 8 SPEs. The memory interface on QS20 imposes higher DMA latencies for 1-2 SPEs, compared to PS3. For more than 2 SPEs, the QS20 is favored, as it handles better the bus contention than PS3. Sequoia performs similarly to *TPC* on QS20 for all applications with small variations that are always under 4%.

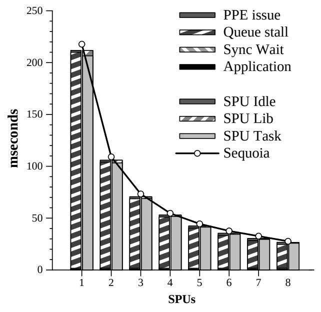
Figure 4.9 shows execution time breakdowns for x264. Library overheads in SPE are lower in QS20 than PS3 due to memory interface and task times remain the same. For single SPE, performance is better in QS20 by 20%



(a) SAXPY, B=4K



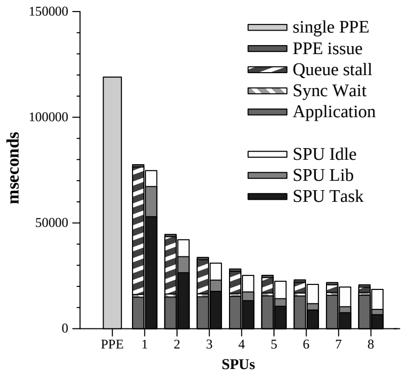
(b) SAXPY, B=2K



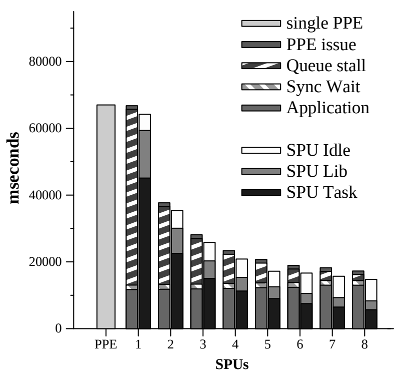
(c) CONV2D

FIGURE 4.8: PPE and SPE execution time breakdowns for 24M element vectors for SAXPY. Runs on BladeCenter

in average. For more than 4-5 SPEs, overall performance remains the same because the reduced DMA latencies are traded with greater idle times in SPEs. This happens because of limited concurrency in the application.



(a) riverbed



(b) blue sky

FIGURE 4.9: *TPC* execution time breakdowns for x264 on *riverbed* and *blue sky* streams. Runs on BladeCenter.

Chapter 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 which dynamically analyzes dependencies between tasks and schedules tasks dynamically upon resolution of 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 measure task initiation/completion times are 10 times lower than those currently reported for CellSs.

OpenMP has recently been extended with a task directive for supporting task parallelism [12]. OpenMP tasks require the programmer to specify the code region that will execute. Instead, *TPC* tasks require specification of both the code and data involved in the computation (as arguments to the call). The *XLC* [11] compiler for the Cell offers an OpenMP abstraction for loop level parallelism, using *DBDB* [9]. Compared to XLC and its runtime, *TPC* has significant differences. XLC splits loop iterations across SPEs and then tries to calculate statically the ideal number of grouped iterations of a loop in order to utilize all available local store and overlap as much communication with computation as possible. On the other hand, *TPC* generates tasks dynamically, where a task can be a group of iterations provided by the programmer. *TPC* uses task queues to overlap the transfer of arguments for upcoming tasks with current task execution and hide the added overheads of dynamic task management. The degree of pre-fetching depends on the amount of available memory and is dynamically determined. Our evaluation shows that *TPC* is successful in hiding DMA latencies, when adequate space is available in the local stores. *TPC* relies on the programmer to specify the type (simple, stride) and size of each argument, based on the argument usage in the task code. DBDB automatically identifies access patterns to data buffers and select the appropriate transfer scheme. Both *TPC* and DBDB map contiguous accesses to a single DMA transfer of a contiguous memory region (flat buffers in DBDB or simple arguments in *TPC*). *TPC* maps non-contiguous accesses always to DMA-list elements, as described by a stride *TPC* argument. This approach reduces argument processing and minimizes transfer initiation overheads in the runtime when dynamically fetching data before task execution. On the other hand, DBDB uses an analytical model to predict whether those accesses should be mapped to single DMA, including unnecessary data, multiple DMAs of individual elements, or a single DMA list. However, author in DBDB find that DMA lists offer almost always the best performance. Overall, the main difference is that *TPC* aims at minimizing the runtime overhead for preparing and initiating data transfers on both the PPE and SPEs, whereas DBDB aims at optimizing the data transfer time. Moreover, *TPC* is a runtime that aims mostly on efficient dynamic task management mechanisms for the Cell, rather than a high-level programming abstraction and automatic parallelism extraction.

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 anecdotal evidence suggests that there are several ongoing efforts for extending these models to support heterogeneous systems in the future.

Chapter 6

Discussion and Future Work

Besides the issues we address in this work, we believe that runtimes for future multicores will need to deal with three additional, broad issues: Minimization of data transfers, mapping of application natural task size to fine-grained tasks for memory efficiency, scheduling of fine-grained tasks, and code management.

In explicitly parallel programming models, executing a task on a core typically involves reading data on the local memory, executing the task, and then propagating updates to the location of the corresponding program variables in a globally accessible external memory. However, fine-grain tasks may have an adverse impact on the amount of data transferred between various types of memories in explicitly managed memories in multi-core architectures. Small tasks may require transferring multiple times the same (control) data that could otherwise be transferred only once for each coarser-grained task. Thus, there is a need for mechanisms that will minimize unnecessary data transfers transparently, in the light of the additional knowledge provided by explicit data accesses in each task. This problem is reminiscent of two other situations: (a) Coherent caches that use invalidations to move the data on demand between different (fast and local) memories that map to the same address range. (b) Register allocation that reduces the number of load/store operations from/to memory by scheduling instructions appropriately and ensuring appropriate use of data already in registers. These solutions rely either on extensive hardware support or extensive static program analysis. Similar decisions will be required in future multicores.

Explicit data accesses and fine-grained parallelism provide the opportunity to hide low-level synchronization primitives from the programmer. In principle, conflicts to data accesses can either be resolved by appropriate synchronization or scheduling. Fine-grained task-based programming models seem to favor using scheduling to avoid waiting on conflicting accesses. However, this requires dependence analysis at the task level, that although possible at runtime, traditionally it has resulted in high runtime costs. This issue will need to be re-examined carefully as the number of heterogeneous tasks and cores scales over time.

Code management for future multicores is an important issue. On-chip memory is limited, the large number of cores may require multiple copies of a code region wasting memory resources or may result in frequent code replacement operations and thus, high overhead and increased non-determinism during execution. Storage management for code and data is undesirable for programmers and significantly increases complexity and effort. Instead it should lie in the responsibility of the architecture and runtime system. Transparent solutions to these problems will rely on knowledge about how frequently and dynamically cores will need to be re-programmed, how static are the requirements of an application in each type of core, and even possibly the relative positioning of each core used by a single application in a CMP. Explicit specification of data accesses at a fine granularity can help the compiler and runtime environment manage task code more efficiently.

Chapter 7

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 the data accessed by them.

TPC implements task management with on-chip operations only for task creation, initiation, assignment, and completion. *TPC* achieves 385ns null task initiation overhead on the Cell and null task initiation/completion overhead close to the round-trip DMA latency. Furthermore, 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 on Cell. 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 that perform static optimizations for data layout and task placement.

Finally, based on our experience with *TPC*, runtime support for future

CMPs will need to deal with three additional, broad issues: Mapping of application natural task size 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.

Bibliography

[1] 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*, Nov. 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 Multipro-

cessors 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, July 1995.