BDDT-SCC: A Task-parallel Runtime for the Single-chip Cloud Computer

Alexandros Labrineas

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisors: Prof. *Angelos Bilas*, Dr. *Polyvios Pratikakis*

This work has been performed at the **Institute of Computer Science (ICS) Foundation for Research and Technology – Hellas (FORTH), Heraklion, Crete, GREECE.**

University of Crete
Computer Science Department

BDDT-SCC: A Task-parallel Runtime for the Single-chip Cloud Computer

Thesis submitted by
Alexandros Labrineas
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Alexandros Labrineas

Committee approvals: ______________________
Angelos Bilas
Professor, Thesis Supervisor

____
Dimitrios S. Nikolopoulos
Professor, Committee Member

____
Manolis Katevenis
Professor, Committee Member

Departmental approval: ______________________
Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, May 2013

Abstract

This thesis presents BDDT-SCC, a task-parallel runtime system for the Intel Single-Chip Cloud Computer. The BDDT-SCC runtime includes a dynamic dependence analysis and automatic synchronization, and executes OpenMP-Ss tasks on a non cache-coherent architecture. We design a runtime that uses fast on-chip inter-core communication with small messages. At the same time, we use non coherent shared memory to avoid large core-to-core data transfers that would incur a high volume of unnecessary copying. We evaluate BDDT-SCC on a set of representative benchmarks, in terms of task granularity, locality, and communication. We find that memory locality and allocation plays a very important role in performance, as the architecture of the SCC memory controllers can create strong contention effects. We suggest patterns that improve memory locality and thus the performance of applications, and measure their impact.

Περίληψη

Η εργασία αυτή παρουσιάζει το BDDT-SCC, ένα task-parallel σύστημα χρόνου εκτέλεσης για τον επεξεργαστή Intel Single-Chip Cloud. Το σύστημα χρόνου εκτέλεσης BDDT-SCC περιλαμβάνει δυναμική ανάλυση εξαρτήσεων και αυτόματο συγχρονισμό, και εκτελεί OpenMP-Ss tasks σε μία αρχιτεκτονική με μη συνεκτικές κρυφές μνήμες. Σχεδιάζουμε ένα σύστημα χρόνου εκτέλεσης το οποίο χρησιμοποιεί γρήγορη ενδοεπικοινωνία με μικρά μηνύματα ανάμεσα στους πυρήνες μέσα στο ολοκληρωμένο κύκλωμα. Την ίδια στιγμή, χρησιμοποιούμε μη συνεκτική κοινόχρηστη μνήμη για να αποφύγουμε μεγάλες μεταφορές δεδομένων από πυρήνα σε πυρήνα, οι οποίες θα επιβαρύνονταν από υψηλή ποσότητα μη αναγκαίων αντιγραφών. Αξιολογούμε το BDDT-SCC με μια συλλογή από αντιπροσωπευτικές εφαρμογές, όσον αφορά την λεπτότητα καταμερισμού εργασίας, την τοπικότητα και την επικοινωνία. Βρίσκουμε ότι η τοπικότητα και η κατανομή μνήμης παίζουν πολύ σημαντικό ρόλο στην επίδοση, καθώς η αρχιτεκτονική των ελεγκτών μνήμης του επεξεργαστή SCC μπορεί να δημιουργήσει έντονα φαινόμενα ανταγωνισμού. Προτείνουμε πρακτικές που βελτιώνουν την τοπικότητα μνήμης και κατά συνέπεια την επίδοση των εφαρμογών, και μετράμε την επίδρασή τους.

Acknowledgements

First of all I would like to thank the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH) for providing me a graduate scholarship and a great working environment at the Computer Architecture and VLSI Systems (CARV) laboratory during my stydies.

Secondly, I would like to thank the University of Crete (UOC) and the Department of Computer Science (CSD) for providing me with high quality education during both my undergraduate and graduate studies.

Moreover, I need to express my gratitute to my advisors, Professor Angelos Bilas and Researcher Polyvios Pratikakis. Furthermore, I would like to give my appreciation to Professor Dimitrios S. Nikolopoulos and Professor Manolis Katevenis for contributing as members of my Masters committee. I would also like to thank all my colleagues for helping me through several discussions.

Last but not least, I am grateful to my family for supporting and encouraging me all these years. Finally, I dedicate this work to the memory of my father, who has always been a source of inspiration for me.

Heraklion – Crete,
May 2013

Alexandros Labrineas

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rising core counts of modern processors follow a trend towards processors with hundreds of cores in the near future. It is becoming apparent that the performance of cache-coherent shared memory does not scale well with the number of cores, leading to systems with high core counts that have either expensive cache-coherent, non-uniform memory access (cc-NUMA) or no cache-coherence at all [1].

The Single-chip Cloud Computer (SCC) is a manycore processor that best represents this trend. The SCC chip consists of 48 cores, placed in a tile formation, with two cores in each tile. The tiles are connected by a 6x4 mesh, which also links with four memory controllers that address the external system memory. The memory address space can be either private to each core or shared by all cores. Accesses to shared memory are not cache coherent. As there is no operating system that can currently use such a manycore processor, the SCC cores are completely independent: each core runs an individual operating system.

Programming such systems requires careful consideration of memory allocation, layouts, locality and access patterns, since not all memory accesses are equally expensive. The commonly used abstraction of shared memory can greatly hurt performance and even break program correctness (for non cache-coherent systems). More importantly, this trend seems to continue strong in the future; recent work from Intel predicts future manycores will not have fully coherent caches [2, 3], and will require a change in runtimes and operating system design.

Traditional threaded programming is not portable in future manycores. Implicit communication between threads using shared memory does not work through non-coherent memories and can hurt performance on cc-NUMA memory. Moreover, clusters and systems like the SCC require explicit communication among cores, which is complex for the programmer to handle. For these reasons, the "threads & shared memory" model is not suitable for these systems. Conversely, task-parallel programming models are better fit for such architectures because they lift the effort required for explicit communication from the programmer to the runtime system.

Task-based parallelism is expressed via annotations in the code that identify certain procedure calls as concurrent tasks. This is a more abstract way to express parallelism. The programmer describes all parallelism without having to manually manage thread or process communication and execution. The runtime extracts the best parallelism automatically according to the system load and the available hardware resources.

Parallel programs require synchronization mechanisms in order to produce correct executions. In early task parallel systems [4, 5, 6], the programmer must use such mechanisms to avoid conflicting memory accesses. Recent task-parallel systems introduce implicit synchronization using dependence analysis to order task execution and avoid conflicts [7, 8, 9, 10, 11, 12]. In contrast to statically expressed parallelism, dynamic dependence analysis only synchronizes tasks that actually have conflicting memory footprints allowing the runtime to discover more parallelism. However, existing task-parallel runtimes target either shared-memory multiprocessors [4, 13, 7] or clusters of nodes that communicate over a network [6, 9], both very different architectures to non-coherent manycores like the SCC.

Overall, this work makes the following contributions:

- We design and implement BDDT-SCC, a task-parallel runtime system with implicit synchronization, for the SCC, a non-coherent manycore architecture.
- We evaluate BDDT-SCC using a set of representative benchmarks and find that memory contention and task granularity play a very important role in the scalability of the benchmarks.

Chapter 2

The SCC Processor

2.1 Architecture

The entire system consists of the SCC board and a Management Console PC (MCPC) used to control the board. The SCC chip is installed on the SCC board and communicates with the MCPC via a System Interface. Figure 2.1 shows the top level architecture of the SCC many-core processor [14]. The SCC chip consists of 48 cores, placed in a tile formation with two cores in each tile. Each core has a unique ID ranging from 0 to 47. A 6×4 mesh connects the tiles to each other and to four Memory Controllers (MCs) that address the external system memory. The chip features extensive frequency and voltage control on a per tile and voltage island basis. This is done by a Voltage Regulator Controller (VRC) inside the die. For all measurements in this work the cores are clocked at 533MHz, the mesh network at 800MHz and the memory controllers at 800MHz.

Figure 2.2 shows the tile level architecture of the SCC. The cores are P54C Pentium IA with an additional L2 cache. Each core has a private L1 instruction cache of 16KB, a private L1 data cache of 16KB and a private unified L2 cache of 256KB. All caches are 4-way set associative with a pseudo-LRU replacement policy. The L1 caches are integrated into the core while the L2 cache is on the tile. Each dual-core tile has 16KB of SRAM dedicated to message passing. This amounts to an on-chip Message-Passing Buffer (MPB) of 8KB for each core. The MPBs are memory-mapped and accessible from all cores. A write combine buffer is added to each core to accelerate the inter-core message transfers. The tile has also a five-port crossbar router to communicate with external components, a Mesh Interface Unit (MIU) to handle all memory requests and two Memory Look Up Tables (LUTs) for translating each core's physical addresses to the extended memory map of the system.

The SCC processor uses four Memory Controllers to address off-chip memory. The controllers address the external DRAM using a physical-to-physical translation through programmable LUTs. By default, each core gets a separate partition of the available DRAM and runs a separate instance of the Linux kernel. The on-chip

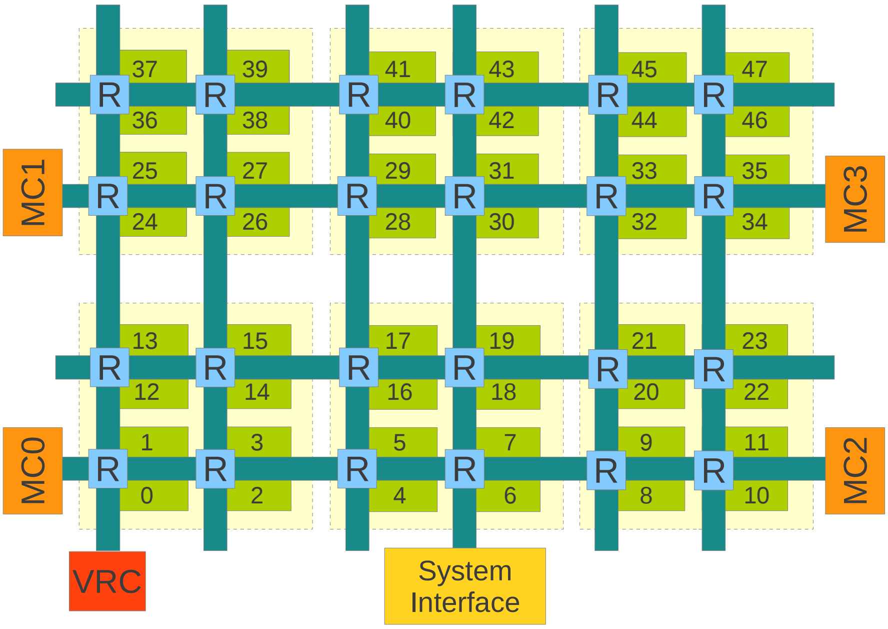


Figure 2.1: The SCC architecture at top level

LUTs can also be programmed so that all cores in the SCC can physically address a set of shared 16MB pages, split among the four memory controllers [15]. These can be memory-mapped by user-level processes running on separate cores so that they share up to 512MB of DRAM[1]. Figure 2.3 shows how the 16MB shared memory pages are assigned to the memory controllers: The first page is assigned to MC0, the second to MC1, the third to MC2, the forth to MC3, the fifth to MC0 and so on (round-robin)[2].

The SCC does not implement cache coherency, so it is the programmer's responsibility to flush the write-combine buffers (equivalent to a write fence) and invalidate the caches (equivalent to a read fence) of cores that access shared memory so that written values become visible to readers correctly.

2.2 Communication

The programming environment for the SCC is RCCE, a small library for many-core communication. RCCE provides a basic interface which is high level and a gory

[1] Unfortunately, the 512MB shared-memory configuration of the SCC overlaps some physical pages used by the Linux kernel of four cores causing these kernels to panic. We omit the crashed Linux kernel cores from our benchmarks.

[2] The memory addresses are notional and do not correspond to actual SHM address ranges.

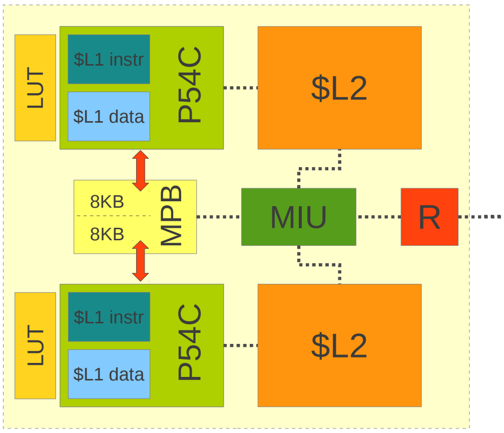


Figure 2.2: The SCC architecture at tile level

interface that exposes more details of the system for more control over the chip. It also supports power management through a special API.

We use a customized version of the gory interface for sending and receiving messages. Moreover we expand the API to support receiving messages that match with anyone among several senders. The interface provides methods to initialize and shut down the environment. In addition, it allows allocation of shared memory via collective calls from the participating processes. The processes can be synchronized with barriers.

A RCCE parallel program can utilize one or more cores. It executes as a set of parallel execution units that are mapped to the cores. Once assigned to a core, an execution unit remains pinned to that core. Each execution unit is a separate process with its own program counter that makes progress in a computation part of the total parallel program.

By default the message-passing mechanism for synchronization and data transfer is implemented via remote reads and local writes. Messages are sent as 8KB chunks of data. Upon a data transfer, the receiver spins on a flag allocated on the sender's MPB as a read barrier. The sender copies the data from its private memory to its local MPB and then sets the synchronization flag. Then the receiver copies the data from the remote MPB to its private memory. We would expect that this scheme floods the on-chip network. We modify the RCCE library in order to use remote writes and local reads. Then we evaluate it with a simple ping-pong application. However, we find that the customized version achieves similar performance to the original RCCE.

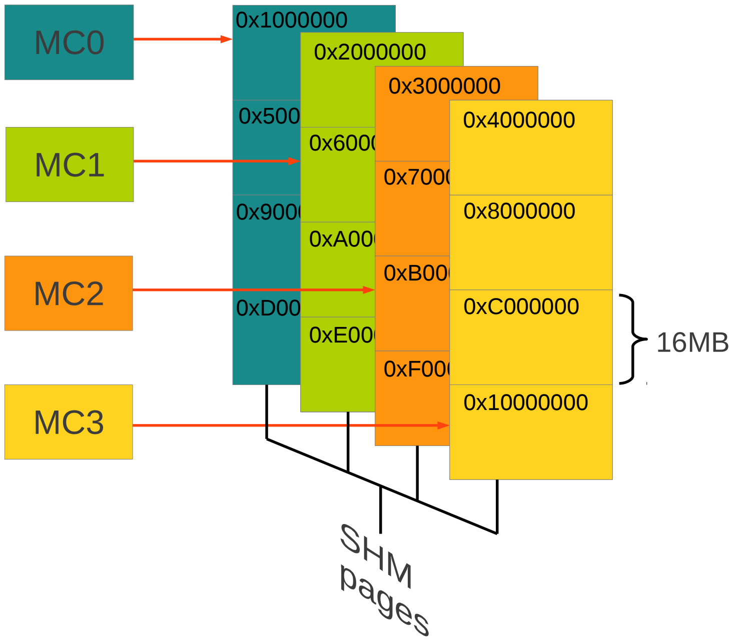


Figure 2.3: The SHM page mapping

Send: On each chunk of data that fits in the MPB the sender does the following. First it waits until the flag on the local MPB that corresponds to the receiver indicates that it is ready. To avoid reading stale data from the cache instead of the new flag value from the MPB, the sender issues MPB cache invalidation before each read of the spin cycle. Then it copies the data from its private memory to the remote MPB. Afterwards it sets the flag in the remote MPB that corresponds to the sender with a value indicating the transfer completion. It needs to write to another line to make sure the write-combine buffer gets flushed. Finally it polls again on the receiver flag placed in its local MPB until the data is received.

Receive: The receiver on the other hand does the following. First it sets the remote flag to indicate that is ready. It also has to flush the write-combine buffer. Then it polls on its local flag until the sender notifies the transfer completion. During polling it repeatedly invalidates the MPB cache. Afterwards it unsets the local flag, copies the data from the local MPB to its private memory and finally sets the remote flag to inform the sender that it has copied the data out of its buffer.

Figure 2.4 shows the execution time (y-axis) of the ping-pong application for

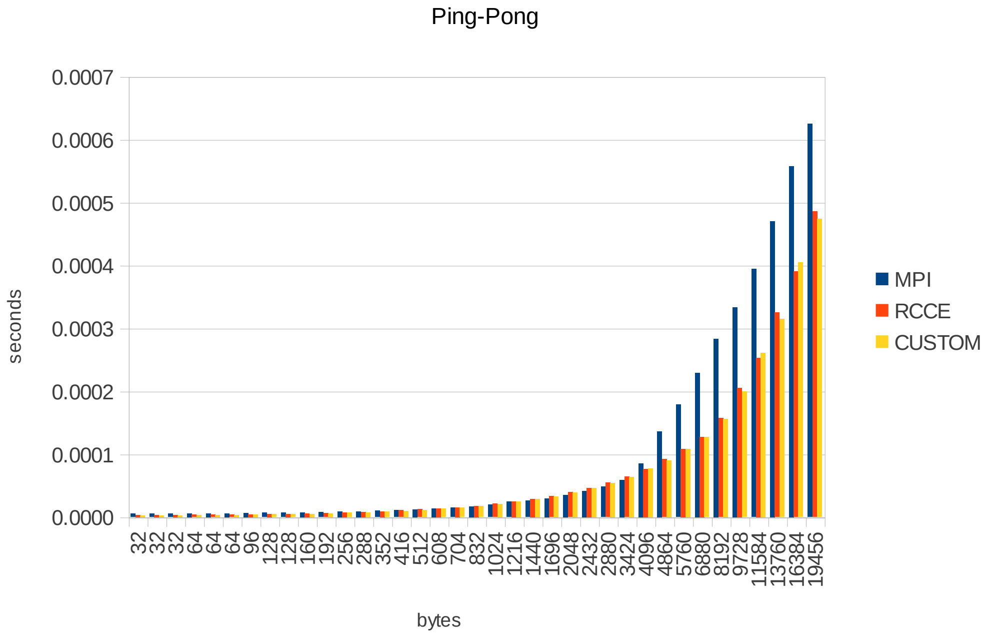


Figure 2.4: Comparison of the available message-passing libraries on the SCC

several data volumes (x-axis). We compare the default RCCE library with our customized version and also with MPI. The customized version performs similar to the original one, both better than the MPI.

Chapter 3

Design and Implementation

3.1 Programming Model

BDDT-SCC, like BDDT [7], implements the OmpSs pro- gramming model [13] for the SCC processor. In OmpSs, the programmer specifies function calls as tasks to be spawned using compiler pragma directives. A *master* core executes the main program and creates tasks to be executed in parallel by *worker* cores. The programmer also specifies task footprints as memory address ranges or multidimensional array tiles. Every task argument is described with a specific data access attribute, corresponding to three access patterns: read (IN), write (OUT) and read/write (INOUT). Dynamic analysis uses these attributes to discover task footprints that overlap in memory and detects dependencies between tasks.

To detect dependencies, BDDT-SCC performs block-level dependence analysis on the task arguments, similarly to BDDT. The block-level dependence analysis uses a custom allocator to split all allocated memory into blocks and discovers task dependencies by detecting whether any arguments of any two tasks contain the same block.

Every task spawned by the application creates a new task instance which goes through four stages in the runtime:

- *Task initiation*, in which the master creates a new task descriptor, detects its dependencies and either adds it to the task graph to wait for its arguments, or marks it as *ready* to run if it has no dependencies.

- *Task scheduling*, in which the master assigns a ready task to an available worker.

- *Task execution*, in which the worker runs the task to completion on its specified arguments and marks it as *complete*.

- *Task release*, in which the runtime removes any dependencies on the completed task –possibly creating new ready tasks– and recycles its descriptor data.

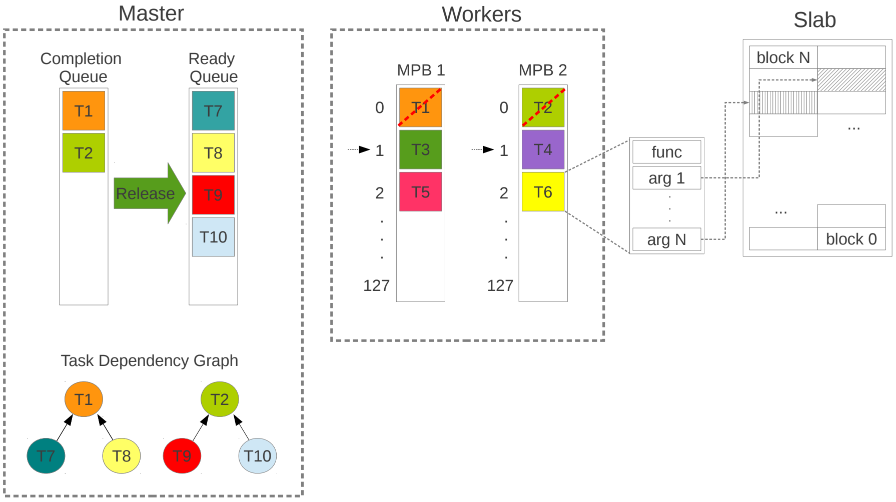


Figure 3.1: Example state during execution

For example, the lower left part of Figure 3.1 shows a task dependency graph with two immediately ready tasks (T1 and T2) and four dependent tasks (T7, T8, T9 and T10) that cannot run before T1 and T2 have completed.

3.2 Memory Management

Within BDDT-SCC, each instance of a spawned task corresponds to a *task descriptor*: a struct that includes a reference to the spawned function, its arguments, and a representation of the task footprint. To keep track of tasks throughout their lifetime, BDDT-SCC inserts each task descriptor into an appropriate data structure. The *ready queue* of the master core contains descriptors of tasks that are ready to run but have not been scheduled for execution to any of the worker cores. The *completion queue* of the master core contains task descriptors of executed tasks, whose dependencies are not yet released. The *task graph* of the master core contains descriptors of tasks with unresolved data dependencies, that cannot be executed before these dependencies are released. BDDT-SCC allocates the ready queue, completion queue and task graph in the master core's private memory.

In the example of Figure 3.1, tasks T1 and T2 are in the completion queue of the master core after they are finished executing on workers 1 and 2 and before their dependencies have been released. When the dependencies of T1 are released, tasks T7 and T8 enter the ready queue of the master. When T2 is released, T9 and T10 enter the ready queue of the master.

BDDT-SCC allocates a queue of task descriptors per worker core. We allocate a task queue as an array in each worker's Message-Passing Buffer. As all cores' MPBs are accessible from all other cores, the master core writes directly to each worker's MPB to enqueue tasks to that worker's queue or collect tasks that have finished executing. Note that each MPB consists of 512 32-byte cache lines and writing a single byte in an MPB will update all 32 bytes of that cache line. So, we align task descriptors to MPB cache lines to avoid false sharing among the master and worker cores.

For example, the middle part of Figure 3.1 shows the message-passing buffers of two worker cores. In MPB of worker 1 the master core has scheduled tasks T1, T3 and T5, whereas in MPB of worker 2 the master core has scheduled T2, T4 and T6. In the state shown, when tasks T1 and T2 finish executing at workers 1 and 2, they are marked as complete. Then, the master core will collect T1 and T2 into its completion queue and reuse their position in MPBs 1 and 2.

Unlike task queues and runtime metadata, application data is often much larger than the available on-chip memory. This means that core-to-core message passing of application data will result in DRAM-to-core (cache misses), synchronous core-to-core communication, and core-to-DRAM (cache evicts) communication[1]. Instead, BDDT-SCC allocates all application data in SCC shared memory, using a custom slab allocator.

For instance, the right part of Figure 3.1 shows the contents of a task descriptor. Each argument of the task references several blocks of data allocated in the shared memory.

3.3 Task Initiation

To spawn a new task, BDDT-SCC allocates and initializes a new task descriptor. To avoid the overhead of allocating and deallocating task descriptors and also improve the locality and cache performance of the runtime system, BDDT-SCC uses a pre-allocated memory pool of task descriptors and recycles deallocated tasks. If there are no free task descriptors, the master core blocks until a task is complete.

After creating a new task descriptor, the BDDT dependence analysis detects any data dependencies between the new task and previous tasks [7]. If the new task depends on existing tasks that have not completed yet, its task descriptor is added to the dependence graph to wait until all the dependencies are resolved. If there are no dependencies, then the task is ready to run.

To detect dependencies, BDDT-SCC uses the BDDT block-based dependence analysis. In short, BDDT-SCC uses a custom allocator to split the application memory into memory blocks and keeps metadata for each block. The runtime creates block metadata for each task that operates on any given block and uses the metadata to order tasks that use the same data. When a task is first in this ordering for all the blocks of its arguments, it has no dependencies and it is ready

[1] An early version of the runtime used solely message passing; we found this scenario caused unnecessary memory traffic, limiting performance.

to run. For details on the BDDT block dependence analysis, we refer the reader to the corresponding technical report [16].

3.4 Task Scheduling

The master core can be in one of two modes: (i) *running*, or (ii) *polling.* Initially in running mode, the master core starts executing the main program and schedules spawned tasks that are *immediately ready* to worker cores. To schedule a task to a worker core in this mode, the master core tries to append the task to the task queue in the worker's MPB[2]. To do that, the master keeps a local index of the next available entry in the MPB queue for each worker and checks the state of this entry. If the entry is *empty*, the master writes the task descriptor in that entry. If the entry holds a completed task, the master enqueues the completed task in the completion queue and replaces it with the ready task. If the next available entry is full, the master adds the task to a local queue of ready tasks and continues with main program execution. This way, the master never blocks at a spawn and will resume the application execution until either all tasks are spawned and it reaches a synchronization point, or it runs out of task descriptors.

At all points where the execution of the main program blocks, the master enters the polling mode. This can happen at synchronization points, which include explicit barriers and the end of the main program, or during task creation, if there are no available task descriptors. During its polling mode, the master performs three functions: (i) It removes ready tasks from the ready queue, as long as it is not empty, and schedules them; (ii) it polls the queue entries of each worker to discover task descriptors marked as completed; and (iii) it removes completed tasks from the completion queue and releases their dependencies.

When on polling mode, scheduling is similar to that on running state. The master appends the descriptor to the next available entry. However, if this entry is full, the master does not return the task back in the ready queue as during the running mode, but continues with the next worker. If all worker queues are full the master dequeues a completed task from its completion queue, *releases* its dependencies and then retries scheduling of the first task.

3.5 Task Execution

To execute a scheduled task, the worker core reads the next ready task descriptor from its local MPB and executes the task. Task execution is simply a call of the task function on the task arguments. The task arguments are allocated in the external shared memory and are thus accessible by all cores. Note that the shared memory is cacheable, but the SCC caches are not coherent. Thus, BDDT-SCC requires every worker core to invalidate its L2 cache before task execution and

[2] This communication is asynchronous: the master core writes directly to the remote MPB without blocking or interrupting the worker

flush it after it, in order to make the task output visible to all subsequent tasks running on other cores and maintain program correctness.

After the worker executes the task function, it marks the task descriptor as completed in the worker's MPB buffer task queue and continues with the next ready task in the queue. To avoid a race between the master and the worker on the MPB task queue of the worker, we use L1 invalidation as a read barrier and flushing the write-combine buffer as a write barrier. Specifically, the worker invalidates its L1 cache before polling each task entry in the queue, and flushes its write-combine buffer after changing a task descriptor from ready to completed. Conversely, the master invalidates its L1 cache before reading a worker's queue. As an optimization, the master does not flush its write-combine buffer after putting a ready task in a worker's queue. This may mean that the worker will not observe the transition from completed to ready or from empty to ready for that entry immediately. That is not an issue, however, since it can only cause the worker to poll its queue again.

3.6 Task Release

The master core locates completed tasks in workers' task queues during scheduling of newer ready tasks, or during polling its mode. To avoid extending the critical path, the master core does not process the completed tasks immediately, but collects them in its completed queue, recycling that space in the workers' queues into new task descriptors. When the master core idles because all worker queues are full, it has reached a barrier, or it needs to recycle the task resources, it iterates the completed queue and lazily releases the completed tasks' dependencies. Releasing a completed task decrements a dependency counter for each of its dependent tasks. If a counter reaches zero, the master removes the newly ready task from the dependency graph and marks it as ready to run.

Chapter 4

Evaluation

4.1 Core Placement

The SCC architecture results in a different latency for accessing DRAM depending on a core's distance from the respective memory controller [17, 18]. We measured the impact of the latency difference using a microbenchmark that repeatedly accesses a 16MB array allocated to take exactly one shared memory page managed by controller 0. Figure 4.1 shows the total execution time depending on how many hops away the core running the microbenchmark was from the controller. Similarly, the MPB access latency varies depending on the core's distance from the respective MPB. Table 4.1 shows the latencies for reading a cache line from a local MPB, from a remote MPB and from the off-chip DRAM.

We took the variable latency into account when placing the cores in BDDT-SCC, so that (i) the master core is one of the middle cores, having almost uniform distance from all memory controllers and worker cores, and (ii) each worker core is placed as close as possible to the master. Therefore, every additional worker has higher communication cost with the master and its distance from the memory controllers deviates from uniform. For instance, a configuration with 31 workers uses all the cores that a configuration with 30 workers uses, plus an additional core

Memory Access	**Latency**
Local MPB	$45C_c + 8C_m$
Remote MPB	$45C_c + 4 \times n \times 2C_m$
DRAM	$40C_c + 4 \times n \times 2C_m + 46C_r$

C_c : core clock cycles
C_m: mesh network clock cycles
C_r : DRAM clock cycles
n : number of hops over the mesh network ($0<n<10$)

Table 4.1: The memory access latencies of the SCC

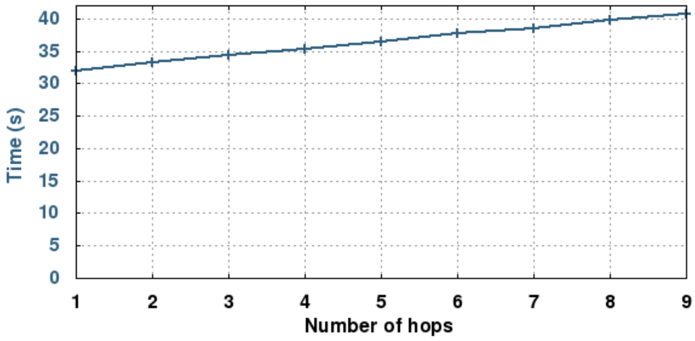


Figure 4.1: Measured memory access latency

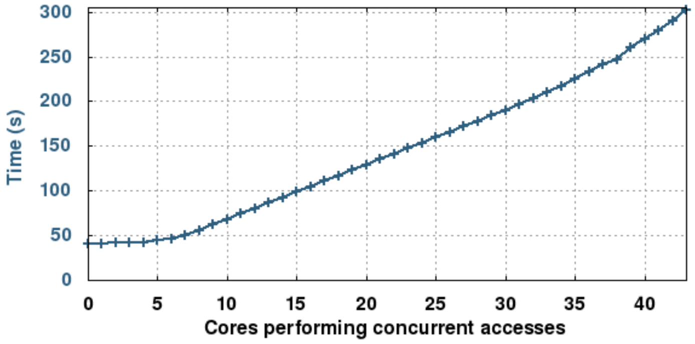


Figure 4.2: Memory contention effects

that is as close to the master as possible.

We placed the master at core 16, one of the middle cores on the SCC (cores 16, 17, 18 and 19 in Figure 2.1). This position minimizes the maximum distance to worker cores to 5 hops, and the sum of hops from the master to all remote MPBs at full chip utilization to 120 hops. Similarly, the closest memory controller is 4 hops away from the master and the furthest is 5 hops away. The total distance from the master core to all memory controllers is 18 hops. Placing the master at any other position results higher number of total hops and increases the runtime's communication overhead.

Moreover, we found that the latency for accessing DRAM increases proportionally to the number of cores performing concurrent accesses through the same memory controller. We use the microbenchmark mentioned above to measure the impact of concurrent memory accesses through the same memory controller. Figure 4.2 shows the total execution time (y-axis) of the microbenchmark run on a reference core, while the same microbenchmark is running on various other cores (x-axis). We select the reference core to be the most distant (9 hops) from the memory controller 0 as a worst-case scenario. The total execution time on the reference core increases with the number of accessing cores, due to contention effects at the memory controllers. This effect does not occur under the standard configuration of the SCC, where each core has access to a disjoint part of the physical memory, accessible only via the nearest memory controller. With shared-memory configuration, however, contention effects become more pronounced, as all cores access all memory controllers.

4.2 Benchmarks

We use 5 well-known applications to evaluate the runtime. *Black-Scholes* is a financial application; we use a data set of 2M options, split into tasks of 512 options. *Matrix Multiply* is a tiled parallel implementation of matrix multiplication; we used 1K×1K floats, split into 64×64 tiles. *Fast Fourier Transform* computes the FFT of a 2D matrix; we used 1M complex doubles, split into blocks of 32 rows at the transformation phase and 32×32 tiles at the transposition phase. *Jacobi Method* computes a Jacobian determinant; we used 4K×4K floats, split into 512×512 tiles, for 16 iterations. *Cholesky Decomposition* computes a matrix factorization; we used 2K×2K doubles, split into 128×128 tiles. All applications except for Black-Scholes have task dependencies. Some benchmarks have small, concentrated datasets that fit within the shared-memory segment of a single memory controller. This creates strong contention effects when all cores access memory through the same memory controller. In these cases, we use padding and non-unit strides during allocation, to distribute application data across all memory controllers as uniformly as possible.

4.3 Results

Figure 4.3 shows the execution time (left y-axis) and scalability (right y-axis) for each benchmark. The x-axis shows the number of worker cores used (*i.e.*, we do not count the master core). We show the performance of the original, sequential program at point 0. The sequential program runs at the master core and allocates all its memory at the nearest memory controller. We exclude initialization time from all measurements and report total time of parallel execution. Black-Scholes and Matrix Multiply scale to 16× and 33× speedups, respectively, compared to the sequential execution.

For each application, we present execution time breakdowns for the worker cores. We break down the execution of each worker in three parts: (i) time waiting the master (idle), (ii) time spent in application code, and (iii) time spent for L2 cache flush and invalidation. Figure 4.4 shows the cumulative breakdowns for all participating cores. FFT, Jacobi and Cholesky feature strong memory contention effects. The cumulative time spent in application code grows as core count increases, since each individual memory access or cache miss costs more.

Figure 4.5 shows the load balance per worker for the configuration with 43 workers for each benchmark. Again, we show the breakdown of the total time spent by each worker, into: (i) time waiting the master (idle), (ii) time spent in application code, and (iii) time spent for L2 cache flush and invalidation. Note that idle time in the workers is always caused by too fine a task granularity, where the master cannot spawn and schedule tasks fast enough to keep all workers busy.

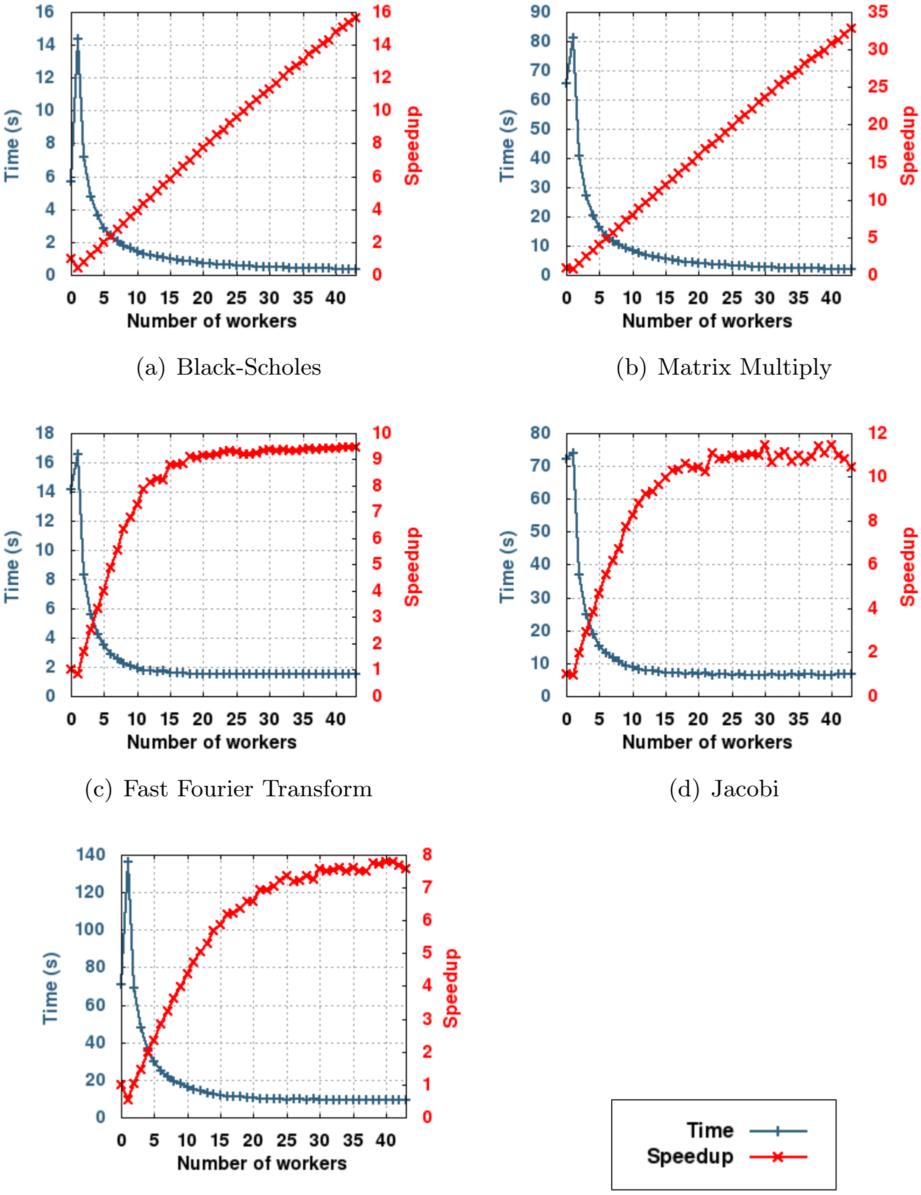
Black-Scholes scales linearly to all the available worker cores (Figure 4.3(a)). However, its speedup is not equal to the number of cores, due to the high flush time to execution time ratio (Figure 4.4(a)). In this case the master core is idle most of its time, waiting for the workers to finish. Black-Scholes also produces a very balanced schedule, where all workers perform an almost equal amount of work (Figure 4.5(a).

Similarly, Matrix Multiply scales proportionally to the number of workers (Figure 4.3(b)), achieving better speedup than Black-Scholes as the constant overhead of cache flushing is minimal compared to the execution time (Figure 4.4(b)). Matrix Multiply is also very well balanced, with all workers performing an almost equal amount of work.

On the other hand, the rest of the applications do not exhibit similar scalability. In all cases, their scalability is limited by the memory contention effects that we demonstrate at the previous subsection.

FFT scales up to 16 worker cores (Figure 4.3(c)), with performance being almost unaffected by a larger number of workers. Figure 4.4(c) demonstrates the effect of contention: the total execution time increases with the number of workers because memory accesses become more expensive, although the total actual work remains the same. The flush time is also slightly affected by the same contention effect, although the number of flushes is constant, equal to the total number of tasks. Moreover, the total idle time starts increasing when the number of workers reaches 10, indicating that the master core is not fast enough to serve all workers beyond that point. This also affects load balancing (Figure 4.5(c)), as the master core cannot keep up with workers that execute faster tasks.

Similarly, Jacobi and Cholesky reach a maximum speedup at 22 worker cores (Figures 4.3(d) and 4.3(e)). Again, this is a combination of memory contention that increases the total task execution time with the number of workers, although the total work remains the same (Figures 4.4(d) and 4.4(e), and also of the master becoming a bottleneck at 13 and 3 worker cores, respectively, increasing the worker's idle time. In turn, this reduces the load balancing of the problem for high core counts (Figures 4.5(d) and 4.5(e), as the master cannot schedule more tasks fast to workers that finish early.



(a) Black-Scholes

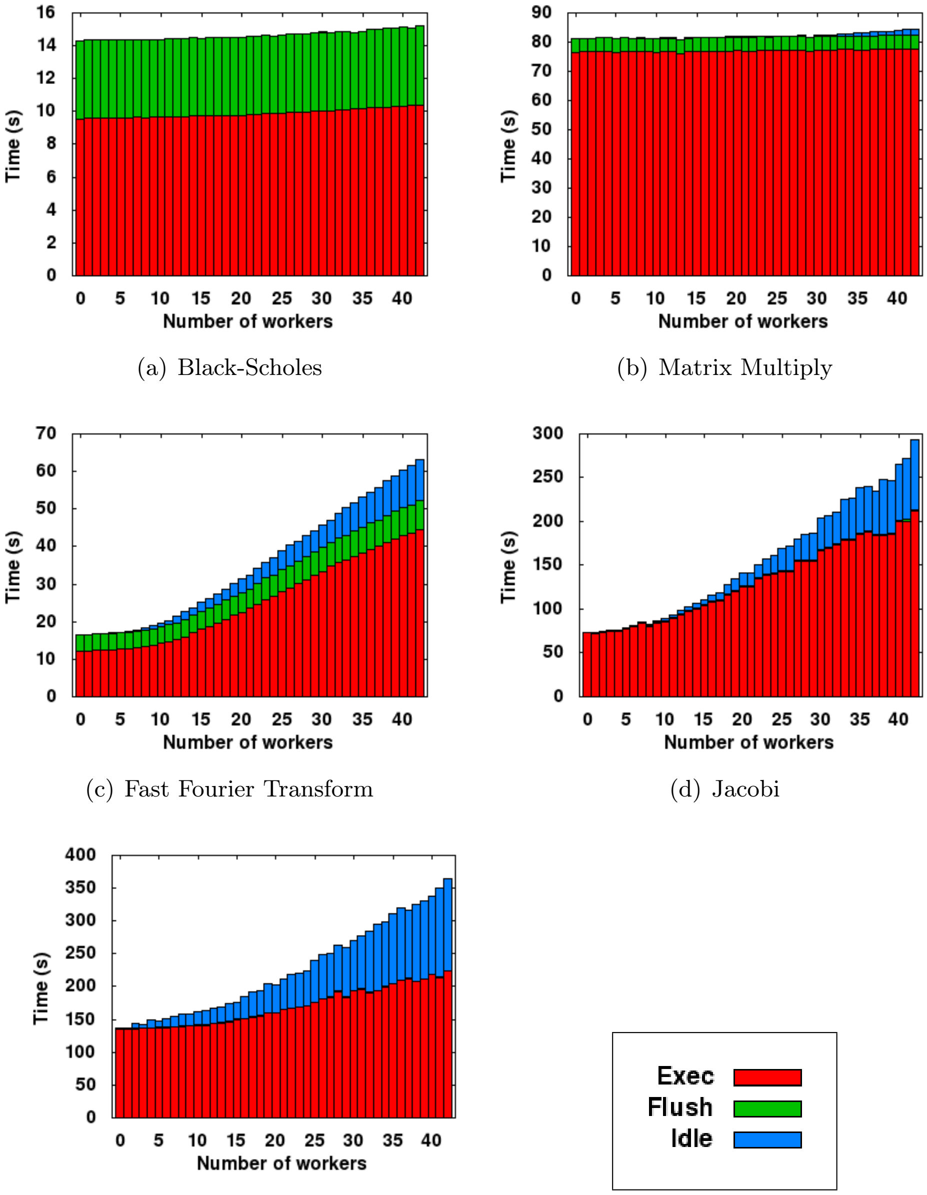
(b) Matrix Multiply

(c) Fast Fourier Transform

(d) Jacobi

(e) Cholesky

Figure 4.3: Benchmark execution time and speedup



(a) Black-Scholes

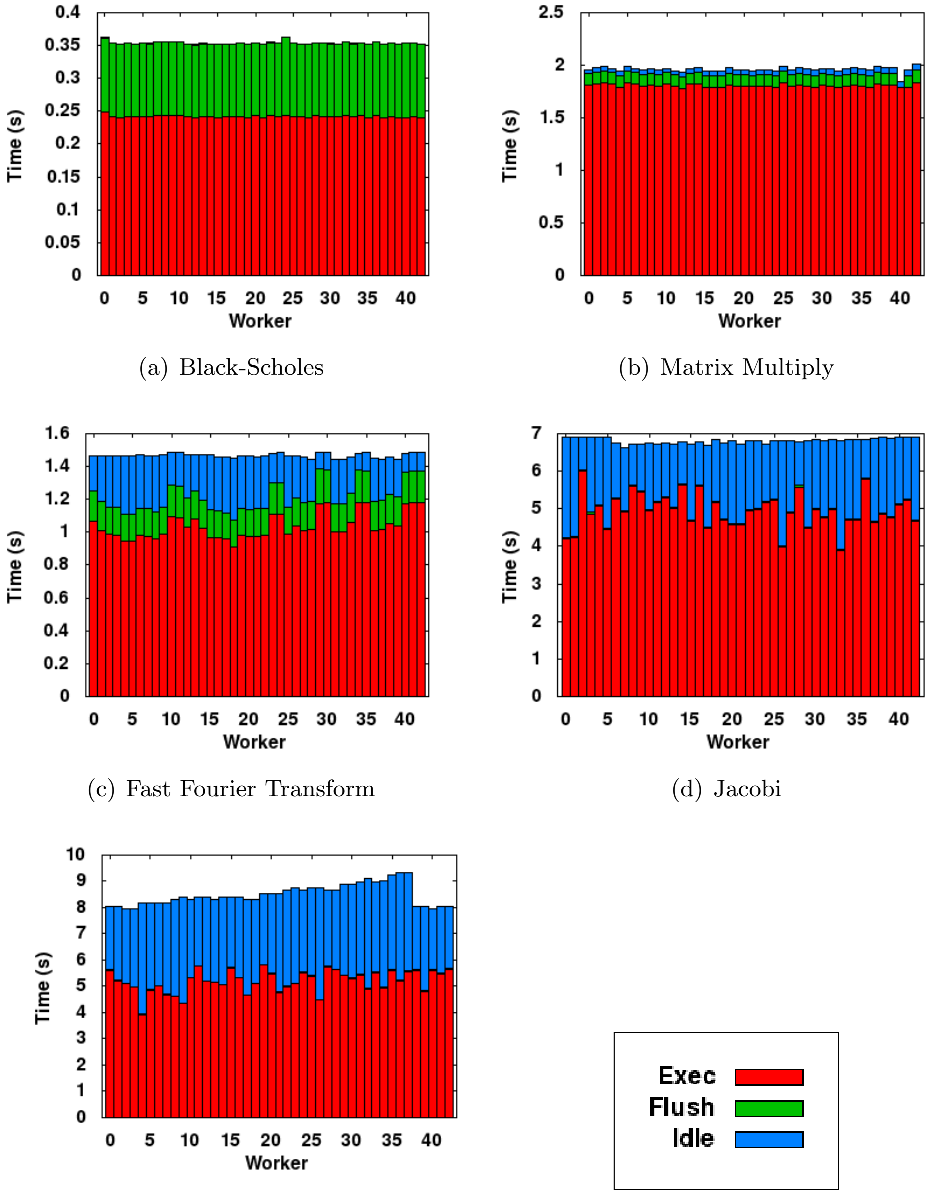
(b) Matrix Multiply

(c) Fast Fourier Transform

(d) Jacobi

(e) Cholesky

Figure 4.4: Breakdown of total processor time per benchmark



(a) Black-Scholes

(b) Matrix Multiply

(c) Fast Fourier Transform

(d) Jacobi

(e) Cholesky

Figure 4.5: Load Balance for 43 workers

Chapter 5

Related Work

Task-parallel programming models have risen as a high-level alternative to thread programming. Task-parallelism allows the programmer to specify scoped regions of atomic code, without specifying synchronization or communication during a task. Early task-parallel programming models do not perform dependence analysis or implicit synchronization [5, 4, 19, 6]. Recent task-parallel systems add either static [20, 11, 21] or dynamic dependence analysis [22, 23, 24, 9]. Most of these systems target shared memory architectures, where cache-coherency automates most of the communication, or clusters, where everything is communicated through message-passing. In comparison, BDDT-SCC targets the SCC, a manycore processor that does not support cache coherency or asynchronous message-passing communication, although all cores can share the physical memory.

The SCC processor relaxes hardware cache-coherence to improve scalability and energy consumption [1, 25, 17, 14]. Early runtimes treat the SCC as a message-passing system [15, 26], use distributed and cluster languages to program it [27], or implement software cache-coherence [28]. However, these approaches fail to take advantage of the non-coherent shared memory of the SCC and also the granularity of tasks that does not require coherence traffic for individual loads and stores. Recent runtimes introduce hybrid address spaces on the SCC via LUT remapping as an alternative method for efficient memory copy operations [29], or a design methodology for implementing scalable runtime systems on many-core architectures without hardware support for cache coherence [30]. Lastly, the SCC has been a platform of baremetal development. A multikernel OS model, that employs message passing instead of data sharing, manages the complete collection the non-cache-coherent SCC cores as a single, unified platform [31].

Chapter 6

Future Work

6.1 BDDT-SCC

Work-stealing: In applications with too fine-grained parallelism we find that the master becomes a bottleneck for the workers. This reduces the load balancing of the problem for high core counts 4.5(c) 4.5(d) 4.5(e). With work-stealing the workers would autonomously fetch task descriptors from remote queues to their local queue without having to wait for the master. However, this requires locks to protect concurrent MPB accesses, or lock-free queues.

Nesting: In many applications (i.e. recursion) parallelism is naturally expressed with nested tasks. In nested parallelism, tasks can spawn other tasks and create task hierarchies. To support nested parallelism the runtime probably requires changes on the dependence analysis mechanism.

Regions: In addition to nesting, BDDT-SCC could support dynamic data structures (linked lists etc), allocated on memory regions. The runtime currently allows allocating data on initialization. Moreover, these data are contiguous pieces of memory. An improvement would be to allow data allocation at runtime inside tasks.

6.2 Other runtimes

Porting BDDT on the SCC helped us understand the strengths and the weaknesses of the chip, as well as the difficulties of developing a parallel runtime system. A challenge would be to port BDDT programming model on other platforms, such as a cluster or a future many-core processor.

Chapter 7

Conclusions

Technology trends dictate that the number of computer processors will increase to hundreds of cores per chip in the near future. However, cache-coherence traffic can limit the performance of such systems as well as worsen their power profile. This is likely to lead manycore manufacturers towards processors with limited relaxed coherence or no cache-coherence at all ([2, 1]). The Single-chip Cloud Computer is a representative processor of this trend.

This paper presents BDDT-SCC, a runtime system for executing task-parallel programs written in the OmpSs programming model on the SCC. We demonstrate that BDDT-SCC scales up to a factor of 33× in applications where the memory traffic is balanced over the four memory controllers of the chip and the tasks' footprint features good cache locality (*i.e.,* Matrix Multiply). Furthermore, we demonstrate that applications with dense, stencil computations reach a scalability limit due to strong memory contention effects before taking full advantage of the chip. We conclude that task-parallel programs can take advantage of the SCC manycore processor through careful consideration of locality and data placement, and load balancing of data across memory controllers. Projecting to the future of manycore processors, scalability could be greatly improved by:

- A mechanism for asynchronous bulk communication between processors will lift the limit of 8KB through-MPB messages.

- Hardware support for fine-grained management of the cache could reduce the amount of cache misses and consequently contention effects. For instance, the SCC uses an older P54C core that does not support L2 partial flushing or separate invalidation.

Overall, we found that the SCC performs better on data-parallel applications (*i.e.,* map-reduce [30]) and coarse-grained parallel programs. Although fine-grained parallelism has greater potential for speed-up, in our current design, a too-fine granularity could make scheduling tasks the bottleneck, limiting scalability.

Bibliography

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 2010, pp. 108–109.

[2] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu, "Runnemede: An architecture for ubiquitous high-performance computing," in *HPCA*, 2013.

[3] R. Knauerhase, R. Cledat, and J. Teller, "For extreme parallelism, your os is sooooo last-millennium," in *HotPar*, 2012.

[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*, 1995.

[5] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, January 1998.

[6] "The sequoia programming language," http://http://sequoia.stanford.edu.

[7] G. Tzenakis, A. Papatriantafyllou, J. Kesapides, P. Pratikakis, H. Vandierendonck, and D. S. Nikolopoulos, "BDDT: Block-level dynamic dependence analysis for deterministic task-based parallelism," in *PPoPP*, 2012, poster paper.

[8] A. Pop and A. Cohen, "OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs," *TACO*, vol. 9, no. 4, pp. 53:1–53:25, Jan. 2013.

[9] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *SC*, 2012.

[10] P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos, “A programming model for deterministic task parallelism,” in *MSPC*, 2011.

[11] J. C. Jenista, Y. H. Eom, and B. Demsky, “OoOJava: Software out-of-order execution,” in *PPoPP*, 2011.

[12] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, “Hierarchical task-based programming with StarSs,” *IJHPCA*, vol. 23, no. 3, pp. 284–299, 2009.

[13] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, “Ompss: a proposal for programming heterogeneous multi-core architectures,” *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[14] Intel Labs, “SCC external architecture specification,” 2010.

[15] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, “Light-weight communications on intel’s single-chip cloud computer processor,” *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 73–83, Feb. 2011.

[16] G. Tzenakis, A. Papatriantafyllou, F. Zakkak, H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos, “BDDT: Block-level dynamic dependence analysis for deterministic task-based parallelism,” FORTH, Tech Report 426, Feb. 2012. [Online]. Available: http://www.ics.forth.gr/~polyvios/forth-tr-426.pdf

[17] Intel Labs, “The SCC programmer’s guide,” 2012.

[18] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, “The 48-core scc processor: the programmer’s view,” in *SC*, 2010.

[19] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2007.

[20] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword, “Synchronization via scheduling: Techniques for efficiently managing shared state,” in *PLDI*, 2011.

[21] R. Bocchino, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, “A type and effect system for deterministic parallel Java,” in *OOPSLA*, 2009.

[22] J. M. Pérez, R. M. Badia, and J. Labarta, “Handling task dependencies under strided and aliased references,” in *International Conference on Supercomputing*, 2010.

[23] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta, “CellSs: Making it easier to program the Cell Broadband Engine processor,” *IBM J. Res. Dev.*, vol. 51, September 2007.

[24] *SMP Superscalar (SMPSs) v2.3 User's Manual*, 2010.

[25] M. Baron, "The single-chip cloud computer."

[26] I. A. C. Ureña, M. Riepen, M. Konow, and M. Gerndt, "Invasive mpi on intel's single-chip cloud computer," in *Proceedings of the 25th international conference on Architecture of Computing Systems*, ser. ARCS'12, 2012.

[27] K. Chapman, A. Hussein, and A. L. Hosking, "X10 on the single-chip cloud computer: porting and preliminary performance," in *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, ser. X10 '11, 2011.

[28] J. Kim, S. Seo, and J. Lee, "An efficient software shared virtual memory for the single-chip cloud computer," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, ser. APSys '11, 2011.

[29] M. W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. R. Jesshope, "Efficient memory copy operations on the 48-core intel scc processor," in *Intel Many-core Applications Research Community Symposium, (MARC)*, 2011.

[30] A. Papagiannis and D. S. Nikolopoulos, "Scalable runtime support for data-intensive applications on the single-chip cloud computer," in *Intel Many-core Applications Research Community Symposium, (MARC)*, 2011.

[31] S. Peter, A. Schüpbach, D. Menzi, and T. Roscoe, "Early experience with the barrelfish os and the single-chip cloud computer," in *Intel Many-core Applications Research Community Symposium, (MARC)*, 2011.