TiNy Threads: a Thread Virtual Machine for the Cyclops64 Cellular Architecture

Juan del Cuvillo Weirong Zhu Ziang Hu Guang R. Gao
Department of Electrical and Computer Engineering
University of Delaware
Newark, Delaware 19716, U.S.A
{jcuvillo,weirong,hu,ggao}@capsl.udel.edu

Abstract

This paper presents the design and implementation of a thread virtual machine, called TNT (or TiNy-Threads) for the IBM Cyclops64 architecture (the latest Cyclops architecture that employs a unique multiprocessor-on-a-chip design with a very large number of hardware thread units and embedded memory) — as the cornerstone of the C64 system software. We highlight how to achieve high efficiency by mapping (and matching) the TNT thread model directly to the Cyclops ISA features assisted by a native TNT thread runtime library. Major results of our experimental study demonstrate good efficiency, scalability and usability of our TNT model/implementation.

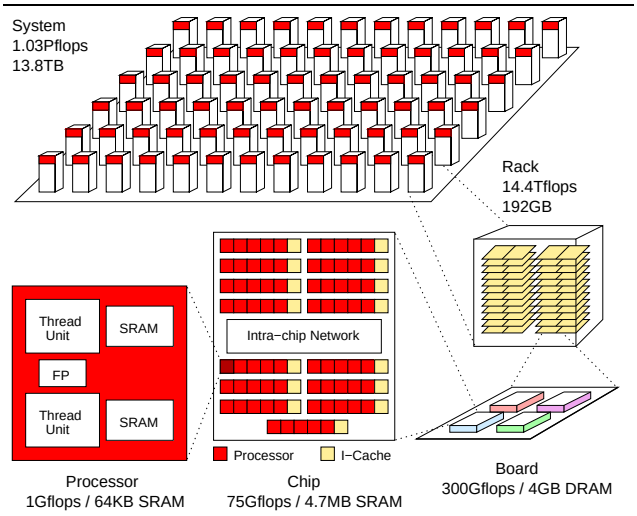

Figure 1. Cyclops64 supercomputer

1. Introduction

The C64 is a petaflop supercomputer project under development at IBM Research Laboratory. C64 is intended to serve as a dedicated compute engine originally designed for running high performance applications such as molecular dynamics to study protein folding [2], or image processing to support real-time medical procedures. A C64 is built from tens of thousands of C64 processing nodes arranged in a 3D-mesh network.

The main objective behind the C64 chip (serving as a compute node in the 3D mesh) design is to build a petaflop computer by scaling up some millions of simple processing elements and provide massive intra-chip parallelism to tolerate memory and functional unit latencies, see Figure 1. On the C64 architecture, the computational cell is the thread unit, a simple 64-bit in-order RISC processor with a small instruction set architecture (60 instruction groups) operating at a moderate clock rate (500MHz).

Perhaps the most remarkable feature of the C64 architecture is its high computation to memory ratio — 150 thread units and 4.7MB of on-chip SRAM. Although the total amount of on-chip memory is comparable to the on-chip data cache of common off-the-shelf processors, C64 has the advantage of a higher communication bandwidth between on-chip memory and thread units. However, C64 hardware does not provide any sort of data cache, as the on-chip memory is exclusively controlled by software. The conclusion from this observation is that on-chip memory is the most precious resource as computation resources, i.e. thread units, become relatively inexpensive.

Given a machine such as C64 cellular architecture, the challenge is to use this massive intra-chip parallelism to obtain high sustained (not peak) performance. Based on our previous experience in the embedded Cyclops32 project [18, 8], we believe that a key requirement from the system software standpoint is a runtime system (RTS), which efficiently manages such a large number of thread units without wasting on-chip memory. In

other words, memory footprint and bandwidth consumption incurred by the RTS must be minimum, so that applications can fully utilize this scarce resource.

This paper focuses on the design of a thread virtual machine for the C64 chip and the first implementation of its thread model as a native TNT thread runtime library. Major results of our experimental study demonstrate good efficiency, scalability and usability of our TNT model/implementation.

2. Cyclops64 chip architecture

The work described in this paper focuses on the C64 chip, the main component of a C64 node, see Figure 2. A C64 chip has 75 processors, each containing two thread units, a floating-point unit and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors.

As we demonstrate in this paper, TiNy Threads, the implementation of our Thread Virtual Machine, takes advantage of relevant architecture features such as: (1) an instruction set architecture design that includes efficient support for thread level execution and a rich set of hardware supported in-memory atomic operations; (2) a tremendous intra-chip communication bandwidth, that can be exploited by multithreading to hide memory access and functional latencies; (3) the capability to configure a section of every SRAM bank as scratch-pad memory, which under software control can store thread local data that can be quickly accessed through a dedicated path.

The C64 instruction set architecture incorporates efficient support for thread level execution. For instance, it provides a sleep instruction, such that a thread can stop executing instructions for a number of cycles or indefinitely. If a thread is expected to wait on an external event or synchronization, i.e. a long-latency operation, it would be judicious to put the thread to sleep and get notified as soon as the long wait is over. A thread is woken up by another thread through a hardware interrupt/signal. Such a wakeup signal is generated when a store into a memory-mapped port is executed. This operation takes as little as 20 cycles when there is no contention in the crossbar network. Additionally, a rich set of hardware supported in-memory atomic operations is available to the programmer. Locks and mutexes can be efficiently implemented using this type of instructions. Unlike similar instructions available on common off-the-shelf microprocessors, when an atomic operation is executed in C64 architecture, the crossbar network only blocks the memory bank where the atomic instruction is operating. Meanwhile, the remaining on-chip memory banks operate normally.

In regard to intra-chip communication bandwidth, each processor within a C64 chip is connected to a crossbar network that can deliver 4GB/s per port, totaling 300GB/s in each direction. The bandwidth provided by the crossbar supports intra-chip communication, i.e. access to other processor's on-chip memory and off-chip DRAM, as well as inter-chip communication via the A-switch device, which connects each C64 chip to its neighbors in the 3D-mesh.

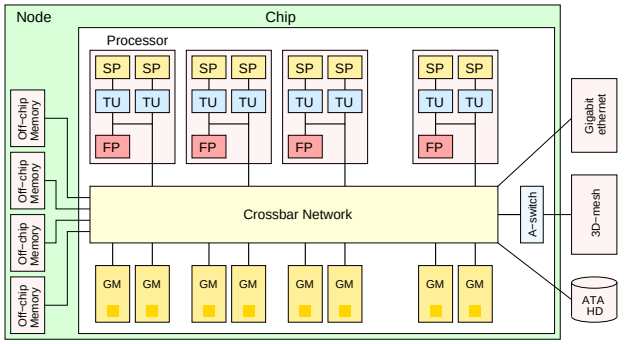

Figure 2. Cyclops64 node

In the C64 chip architecture there is no data cache. Instead a portion of each SRAM bank can be configured as scratch-pad memory and accessed through a dedicated path. Such a memory provides a fast temporary storage to exploit locality under software control.

3. Cyclops64 thread virtual machine

Cellular organization and high computation to memory ratio are probably the most noticeable features of the C64 architecture. However, after a closer look, the system software designer will find some other distinctive characteristics. For instance, similar to general purpose microprocessors, C64 supports user and supervisor operation modes as well as a set of interrupts, that together provide the mechanisms required for protection. However, execution is non-preemptive. That means that the OS will not interrupt the user program running on a thread unless the user explicitly specifies preemption or an exception occurs. Memory organization is another aspect where C64 architecture represents a major departure compared to more conventional systems. There is no hardware virtual memory manager, which means the memory hierarchy of the C64 chip is visible by the programmer. Within a chip, on-chip and off-chip memory banks form a non-uniform shared address space. Processors can directly address any memory location. Additionally, the C64 design, unlike its predecessor [1, 2, 5], does not have data cache, and the computation to memory ratio has been quadrupled as on-chip memory was reduced to one half while doubling the number of thread units.

Given C64 special features described above, it is not our intention to develop a conventional OS for this platform. Such an approach would put a considerable stress on top of a machine aimed for simplicity from the bottom up. In-

stead, we focus our efforts in the design and implementation of a Thread Virtual Machine that provides a familiar but efficient application programming interface. As part of the C64 TVM we have identified three components: a thread model, a memory model and a synchronization model. A high level overview of the thread model and its API is presented below whereas details regarding the implementation, TiNy Threads, are left for section 4.

3.1. The thread model

A program section can be declared as a thread. A thread can be activated for execution by binding a hardware thread unit within a certain chip to a thread activation pointer. A thread activation pointer can be defined as the tuple: <program pointer, state pointer>, where program pointer is the address specified by the program counter associated with the corresponding hardware thread unit and the state pointer points to thread specific information stored in the C64 memory map (e.g. thread identifier, stack pointer, etc.)

A thread activation pointer can also be "global" if the thread handler is extended with a node (or chip) identifier — a system-wide identifier of the chip where the corresponding thread unit resides. The binding of a thread activation to a thread unit can be dynamic, as long as the binding information is properly maintained by the system software.

3.2. Thread model API

For the first release, an interface inspired by that of the popular Pthread model, is provided to ease the first hands-on experience of application and system software developers. Initially, the user is responsible for creating, terminating and synchronizing threads by inserting appropriate function calls to the TNT runtime library. In the future we expect to support other parallel programming models, for instance OpenMP, that demand less effort on behalf of the programmer to manage such a high number of threads.

tnt_create(tnt_desc_t *th, const void *(*fn)(void *), const void *arg)

Runs the user provided function in the next available thread unit. If no thread unit is available, it returns an error condition, otherwise it returns a unique identifier (descriptor) for the new thread. One parameter can be passed to the thread function.

tnt_exit(const void *rc)

Caller thread terminates its execution returning and the exit code specified by **rc** is made available to any successful join with the terminating thread.

tnt_join(const tnt_desc_t th, void **th_ret)

Caller waits for the target thread to terminate. If it returns succesfully, the value passed to **tnt_exit()** by the terminating thread will be placed in the location referenced by the parameter **th_ret**.

tnt_self(void)

Obtains the descriptor of the current thread.

tnt_get_num_threads(void)

Obtain the number of thread units available for execution in the chip.

tnt_self_id(void)

Returns the virtual thread identifier of the caller thread. The identifier is an unsigned number between 0 and the number returned by the function **tnt_get_num_threads()**-1.

4. Tiny threads design and implementation

In this section we discuss the implementation of our thread model for the C64 cellular architecture based on TiNy Threads (TNT).

In the TNT thread model, thread execution is non-preemptive and software threads map directly to hardware thread units. In other words, after a software thread is assigned to a hardware thread unit, it will run on that hardware thread unit until completion. Furthermore, a sleeping thread will not be swapped out so that idle hardware resources can be assigned to another software thread. Upon initialization, each software thread is given control over a well determined region of the scratch-pad memory, which is allocated to every physical thread unit at boot time. This enables fast thread creation and reuse. As in other thread models, a waiting thread (waiting on an external event/synchronization) goes to sleep; such a thread is woken up by another thread through a hardware interrupt/signal.

A thread in Tiny Threads is identified by a pointer that references the corresponding TNT descriptor. The use of such a descriptor simplifies the management of threads, since this fixed-sized structure (less than 100 bytes) holds all the information required to properly handle a thread, including its stack pointer. TNT structures are initialized at boot time. For example, pointers to the thread stack and wakeup memory area are set when the system starts up. Until a thread is requested to run some threaded function, its status is said to be inactive and is therefore available. Idle threads are queued in a singly-linked list so that one can be easily found when a request for spawning a new thread is received. Since multiple threads can call for TNT services simultaneously, all operations upon TNT structures are guarded by a lock (one lock per descriptor). TNT structures are allocated at the beginning of the scratch-pad memory in a memory region we call thread unique area or TUA. The scratch-pad memory area above the TUA is reserved for the thread stack. The compiler toolset and runtime system software share a general purpose register, which points to the end of the stack and beginning of the TUA. This reg-

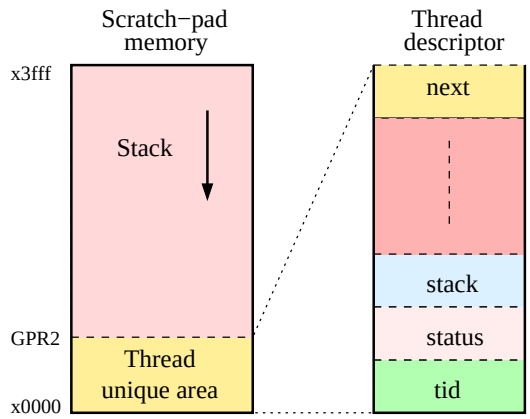

Figure 3. Scratch-pad memory organization

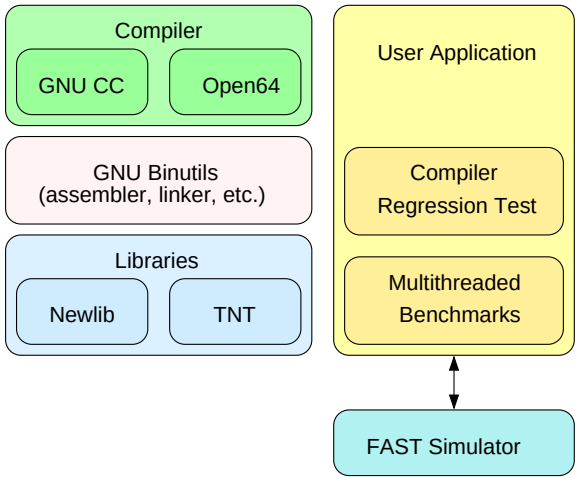

Figure 4. Cyclops64 software toolchain

ister is used both to check for stack overflows and to address the TUA, see Figure 3.

Although the synchronization model has not yet been defined, TNT provides functions for thread synchronization as part of its API. For applications that work under low memory contention, a spin lock mechanism is available. This is implemented with an in-memory atomic operation similar to the test-and-set instruction present in general purpose microprocessors. Since the C64 has no data cache, under high contention, a thread spinning on a lock interferes with other threads (or at least with those trying to access the memory bank where the lock is) by generating traffic on the crossbar network. The mutex operation is defined such that when a thread fails to lock the mutex, the thread is put to sleep. While asleep, a thread unit does not execute instructions until another thread unit generates a wakeup signal, i.e. executes a store into the wakeup port of the sleeping thread or an interrupt occurs. TNT primitives that put to sleep and wake up threads are based on the native sleep instruction and wakeup signal. They can be performed in as little as 1 and 20 cycles, respectively.

TNT barriers are implemented using the signal bus (SIGB) special purpose register. All the thread units on a chip are connected by an 8-bit bus, which is accessible through read and write operations on this register. The actual implementation uses three bits to ensure forward progress, even when arbitrarily long delays occur between instructions. However, let us assume the appropriate bit of the SIGB register is initially set to one. Upon entering the barrier, threads reset that bit to zero and wait for it to drop to zero (according to the hardware design this does not interfere with other thread units or generate excessive heat). Changes in the state of the bus propagate throughout the chip in a few cycles, providing a means for very fast global synchronization.

5. Results

We demonstrate the efficiency and usability of our TiNy Threads runtime library implementation using a diverse set of benchmark programs.

5.1. Summary

The main results of our experimental study can be summarized as follows:

- Efficiency (section 5.3): Our microbenchmarks show that TNT primitives for thread creation and termination take 286 and 60 clock cycles, respectively.
- Scalability (section 5.4): For a collection of assorted kernels, good scalability is reported for applications that have enough parallelism and run well given the high intra-chip bandwidth.
- Usability (section 5.5): The TNT programming API is simple to use, yet sufficiently general to write realistic applications. We port NAS OpenMP parallel benchmarks to TiNy Threads and report our initial experience and findings.

5.2. Experimental platform

Figure 4 illustrates the software toolchain available for application development on the C64 platform. C and Fortran compilers have been ported from the GCC-3.2.3 suite. Assembler, linker and other binary utilities are based on binutils-2.11.2. The C standard and math libraries are derived from those in newlib-1.10.0. Additionally, we wrote the TNT runtime system and a communication library to manage the C64 inter-chip communication device. Finally,

to provide developers with an experimental platform to validate the C64 toolchain, we wrote a functionally accurate simulator (FAST) that is described next.

FAST: C64 functionally accurate simulator toolset. To conduct our research before a real hardware or emulation platform is available, we wrote FAST: an execution-driven, binary-compatible simulator of a multichip multithreaded C64 system. FAST accurately reproduces the functional behavior and count of hardware components such as chips, thread units, on-chip and off-chip memory banks, and the 3D-mesh network, see Table 1.

RISC-like instructions such as integer, floating-point, branch and memory operations are modeled based on execution times expressed by x/d pairs, where x is the execution time in the ALU, and d represents the delay before the result of the instruction becomes available. Instruction timing reported in Table 2 is based on information provided by the C64 chip designer. For instance, integer division is said to take one cycle in the ALU but a subsequent instruction will not be able to use the result until 33 cycles later. During this delay, execution of independent instructions can proceed normally.

FAST allows thread units to fetch, decode and execute instructions at their own pace, following the sequence of events dictated by each thread execution. However, to properly handle C64 special operations that represent some sort of synchronization event between threads (sleep instruction, wakeup signal, inter-thread interrupt, etc.), threads can only commit instructions once the simulated chip clock reaches the time point at which the instruction is executed by the thread. In other words, instructions are executed asynchronously but committed in a synchronized fashion.

Timer alarm events and error conditions such as the execution of an illegal instruction are modeled as exceptions by the simulator. Throughout the instruction's execution, multiple exceptions may occur. FAST keeps track of all of them. Before putting the results away, it triggers an interrupt according to the priority order specified by the architecture.

Given the appropriate command line options, FAST generates the execution trace and/or an instruction statistics report to help the software developer debug, tune and optimize a program. Although FAST is not cycle accurate, we are confident of the general trends demonstrated in this paper and certain it is useful for software development and performance estimation.

5.3. Microbenchmarks

An efficient runtime library in terms of thread management must provide the minimum functionality required to write multithreaded programs with low overhead. From the API briefly described in section 3.2, it is clear that the most expensive operations are thread creation and termination.

Component	# of units	Params./unit
Threads	150	single in-order issue, 500MHz
FPUs	75	floating point/MAC, divide/square root
I-cache	15	32KB
SRAM (on-chip)	150	32KB
DRAM (off-chip)	4	256MB

Table 1. Simulation parameters

Instruction type	x	d
Branches	2	0
Integer multiplication	1	5
Integer divide	1	33
Floating add, mult. and conv.	1	5
Floating mult. and add	1	10
Floating divide double	1	30
Floating square root double	1	56
Floating mult. and accumulate	1	5
Memory operation (local SRAM)	1	2
Memory operation (remote SRAM)	1	20
Memory operation (off-chip DRAM)	1	36
All other operations	1	0

Table 2. Instruction timing

To measure the overhead imposed by these operations, we wrote a simple microbenchmark program.

The program consists of two loops. The first loop spawns a number of threads, which in turn, execute a function consisting of a delay that guarantees each software thread runs on a different thread unit. Once all threads finish, the second loop performs a join operation for each previously created thread. The execution time of each loop is divided by the number of spawned/joined threads to obtain the average thread creation and termination times. The results show that to create and join a thread takes approximately 286 and 60 clock cycles, respectively. By inspecting the trace file obtained from the execution of the microbenchmark, we additionally determined the reuse time. The reuse time is defined as the elapsed time since a software thread terminated the execution of the threaded function until the hardware thread unit goes to sleep, where it can be assigned to work on another task. As it turns out, 265 cycles are required.

For illustration purposes, the same microbenchmark program was written using Pthreads and ran on two SMP platforms: a PowerEdge 6600, 4 CPUs, 1.4GHz running Linux and a Ultra Enterprise 4500, 4 CPUs, 400MHz running Sun Solaris. From the experiment, results show Pthread creation

takes 105 thousand and 100 thousand cycles on the PowerEdge and Ultra Enterprise, respectively, while Pthread termination takes 35 thousand and 52 thousand cycles on the same platforms. As we mentioned above, TNT creation and termination take only 286 and 60 cycles, respectively. That means thread creation is almost 350 times faster while thread termination is between 580 and 860 times faster.

Microbenchmarks analysis. After demonstrating the efficiency of TNT primitives, the key question is: what features of the C64 architecture can be exploited by the TNT runtime library to yield such results?

1. Direct use of C64 ISA:

   Sleep instruction: a thread is put to sleep in 1 clock cycle only. While a thread is asleep, no instructions are executed. Hence, no memory bandwidth is wasted.

   Wakeup instruction: featured as a store into a memory-mapped port. It takes 20 clock cycles to wake up a thread.

   In-memory atomic operations: used to implement locks and mutexes. While such instructions are executed, only the memory bank where the operation takes place is locked. All others operate normally.

2. Managing hardware threads in user mode: If TNT primitives were trapped into the supervisor mode to service a request, they would had experienced an additional 60-cycle delay due to a context switch (saving and restoring a few registers) and calling a dispatcher.

3. The scratch-pad memory advantage: The TNT memory footprint is small (approximately 100 bytes). It fits very well in scratch-pad memory without affecting the thread stack. Access to the scratch-pad memory is fast, similar to a L1 data cache. However, the use of a conventional cache in this case would not guarantee the thread-specific data to always be in the cache, unless the thread descriptor is locked. If TNT descriptors were not allocated on SPM, create and join primitives would have taken 414 and 77 cycles, respectively. This means a 20% and 28% increase in the execution time!

5.4. Assorted scientific kernels

In this section, we present results for a small set of scientific kernels, each representing a typical application class with different computation to communication ratios:

Matrix-matrix-multiply: The result matrix is partitioned among threads which carry out the computation independently of each other. Sobel is an edge detection algorithm widely used in computer vision. Each pixel is computed independently from the others, based on the first derivative of the intensity. Laplace is the parallel implementation of a hypothetical 1D Laplace solver. At each iteration, every position of a single single-dimension array is updated with a value function of its neighbors. However, all the elements of the array need to be updated before the following iteration can start. Heat simulates the heat conduction over a solid plate (modeled as a $N \times N$ grid) using finite differential equations. First, the grid is initialized with one side of the plate being heated. Then, the simulation starts with the heat transfer from that side to the rest of the plate. The program runs until a convergence condition is reached. Nqueens counts the number of ways in which N queens can be arranged on a $N \times N$ chess board so that no queen can attack any other queen under normal chess rules. In this implementation, a number of independent tasks are created after the first three rows of the board are filled with queens in valid positions. Then, the tasks are distributed among threads in a round-robin fashion.

We chose the problem size such that the input data sets fit in the on-chip memory (i.e. global memory, not scratch-pad): MxM, 256×256; Sobel, 1024×1024; Laplace, 2^{17}; Heat, 2048×2048; Nqueens, 12. Since applications have enough parallelism, they should run well given the high bandwidth provided by the crossbar switch despite the C64 architecture's lack of data cache.

Figure 5 shows the absolute speedup achieved by the TNT kernel programs. As expected, the results show a decline in the speedup as the synchronization/communication increases. Nonetheless, it also demonstrates the efficiency of the thread management system that allows Sobel to achieve almost linear speedup, even for 128 threads. We also like to highlight that despite a global barrier, Heat achieves a speedup of 90 on 128 threads.

5.5. NAS parallel benchmarks

To demonstrate that TNT programming API is simple to use, yet sufficiently general to write realistic applications, we port the OpenMP C version of NAS NPB-2.3 parallel benchmarks written by the Omni project into TiNy Threads. TNT versions are hand-coded with no optimizations, mapping OpenMP parallel constructs and library calls to TNT function calls. For example, an OpenMP parallel region results in code that forks and joins threads; critical sections are directly translated to lock and unlock operations on a mutex, etc. Such an exercise took a reasonable effort and was completed fairly quickly.

To verify the correctness of the implementations we select NAS class S benchmarks. All programs run successfully on a number of threads between 1 (sequential) and 150 (number of threads per chip). Except for EP, these programs do not have enough parallelism to show good scalability, mainly due to their small data sets. For instance, the outermost loops of BT, LU and SP have only 12 iterations,

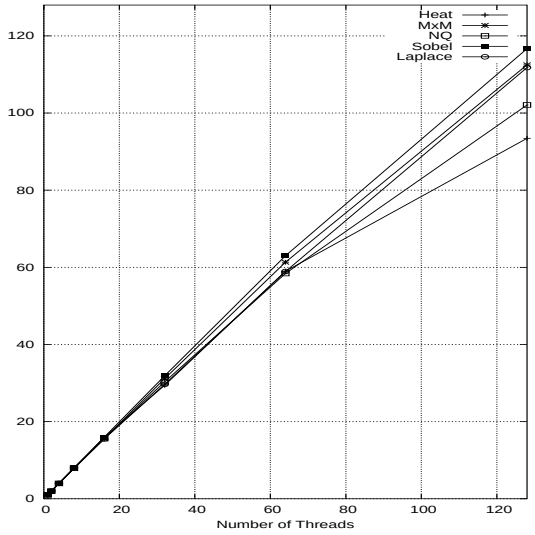

Figure 5. Assorted kernels: absolute speedup

preventing our non-optimized implementation from achieving significant scalability when run on more than 8 threads (data not shown). We also try NAS class W benchmarks. However, only EP, MG, BT and CG run successfully. The reason for missing some of the benchmarks is the limited hardware memory size. More specifically, the stack, which is allocated in the scratch-pad memory, can not be dynamically extended in the current version.

6. Related work

Previous work related to TNT are thread packages that are either found in multithreaded runtime systems or provided as stand-alone thread libraries such as: Coda LWP [17], Pthreads [14], QuickThreads [13], TAM [6], Converse [12], Nano-Threads [15], OpenThreads [11], Active Threads [21], Cilk [10], EARTH [19], NPTL [9].

These thread packages have been developed as part of the runtime system for multithreaded parallel programming languages with goals in terms of portability, interoperability and open implementation in regards to design decisions (e.g. scheduling and preemption). To achieve portability across parallel machines and environments, a number of them assume a common software substrate to be provided by the OS or a machine-dependent layer. Unlike them, TNT is part of a microkernel for the C64 that runs directly on top of the C64 architecture aimed to high efficiency at the expense of portability.

Some thread packages such as TAM, Cilk and EARTH implement a Thread Virtual Machine, similar to that of TNT. However, the resulting TVMs are influenced to different extents by dataflow execution models whereas TNT is not.

Another work on a thread package for a cellular architecture is the BlueGene/L runtime system. However, C64 employs a radically different chip architecture, which presents a different set of challenges and opportunities. For instance, the C64 has no data cache, a multilevel memory hierarchy visible by the programmer, a larger computation to memory ratio, etc.

Our work was initially influenced by QuickThreads because its hardware efficiency and by Pthreads in the sense of usability. We soon realized that the C64 chip architecture is fundamentally unlike those considered on previous work based on QuickThreads. Therefore, it turns out to be more convenient to develop a TNT runtime software infrastructure from scratch. In this first version, TNT machine-specific functionality has been wrapped to provide an interface similar to that of Pthreads, which shall facilitate the initial application development and testing of the C64 toolchain.

There are many available emulation and simulation platforms such as MicroLib [16], PearPC [3], Liberty [20], SimpleScalar [4], etc. However, the C64 chip is not a general purpose microprocessor. Although its ISA is similar to that of the PowerPC, the lack of an out of order engine, as well as other dissimilarities enumerated in Section 2, make the use of PowerPC-based emulation/simulation tools impractical. Instead, FAST is inspired by SimpleScalar's design to achieve good performance, but was written from scratch to accurately reproduce the behavior of the C64 architecture details not present in other simulators (e.g. the hardware wakeup signal, the sleep instruction and the crossbar network).

Due to space restriction of this paper, previous work relevant to Cyclops64 cellular architecture is not described in this paper. A review can be found in [7].

7. Conclusions

We have reported our work in developing a thread model for the Cyclops64 architecture as the first component of a Thread Virtual Machine targeted to cellular architectures in general.

Our TiNy Threads implementation, which presents distinctive features such as direct mapping of software threads to hardware thread units and preallocation of resources (thread-unique memory area assigned to each thread upon initialization), blends nicely with the Cyclops64 hardware resources. We demonstrate that this approach offers high efficiency, scalability and usability.

8. Future work

We believe that TiNy Threads is a good starting point in the development of a portable thread package targeted to cellular architectures. In addition, we do not discard porting other thread packages to the Cyclops64 platform using TNT as interface.

Acknowledgments

We acknowledge support from IBM, in particular, Monty Denneau, Henry Warren, José Castaños and Christos Georgiou. We thank ETI for support of this work, especially Clement Leung. Thanks to many CAPSL members for helpful discussions, in particular, Fei Chen, Hirofumi Sakane, Alban Douillet, Geoff Gerfin and Brice Dobry

References

[1] G. Almási, C. Caşcaval, J. G. Castaños, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren, Jr. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. *ACM SIGARCH Computer Architecture News*, 31(1):26–38, March 2003.

[2] G. S. Almási, C. Caşcaval, J. G. Castaños, M. Denneau, W. Donath, M. Eleftheriou, M. Giampapa, H. Ho, D. Lieber, J. E. Moreira, D. Newns, M. Snir, and H. S. Warren, Jr. Demonstrating the scalability of a molecular dynamics application on a petaflops computer. *International Journal of Parallel Programming*, 30(4):317–351, August 2002.

[3] S. Biallas. PearPC - PowerPC architecture emulator, May 2004.

[4] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin at Madison, Madison, Wisconson, June 1997.

[5] C. Cascaval, J. G. Castaños, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H. S. Warren, Jr. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 311–321, Boston, Massachusetts, February 02–06, 2002.

[6] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM – a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[7] J. B. del Cuvillo, Z. Hu, W. Zhu, F. Chen, and G. R. Gao. Toward a software infrastrucutre for the Cyclops64 cellular architecture. CAPSL Technical Memo 55, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, April 2004.

[8] J. B. del Cuvillo, R. Klosiewicz, and Y. Zhang. A software development kit for DIMES. CAPSL Technical Note 10, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, September 2003.

[9] U. Drepper and I. Molnar. The native POSIX thread library for linux. Technical report, Read Hat, Inc., January 2003.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montréal, Québec, June 17–19, 1998.

[11] M. Haines and K. Langedoen. Platform-independent runtime optimizations using OpenThreads. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 460–466, Geneva, Switzerland, April 1–5, 1997.

[12] L. Kalè, J. Yelon, and T. Knauff. Threads for interoperable parallel programming. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, number 1239 in Lecture Notes in Computer Science, pages 534–552, Ithaca, New York, August 8–10, 1996.

[13] D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report UW-CSE-93-05-06, Department of Computer Science and Engineering,University of Washington, May 1993.

[14] F. Mueller. Pthreads library interface. Technical report, Department of Computer Science, Florida State University, July 1993.

[15] D. S. Nikolopoulos, E. D. Polychronopoulos, and T. S. Papatheodorou. Efficient runtime thread management for the Nano-Threads programming model. In *Proceedings of the 2nd IPPS/SPDP Workshop on Runtime Systems for Parallel Programming*, pages 183–194, Orlando, Florida, March 30, 1998.

[16] D. G. Perez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 43–54, Portland, Oregon, December4–8, 2004.

[17] J. Rosenberg. LWP user manual. Technical Report CMU-ITC-85-037, Information Technology Center, Carnegie-Mellon University, June 1985.

[18] H. Sakane, L. Yakay, V. Karna, C. Leung, and G. R.Gao. DIMES: An iterative emulation platform for multiprocessor-system-on-chip designs. In *Proceedings of the IEEE International Conference on Field-Programmble Technology*, Tokio, Japan, December 15–17, 2003.

[19] K. B. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, McGill University, Montréal, Québec, May 1999.

[20] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with liberty. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 271–282, Istanbul, Turkey, November18–22, 2002.

[21] B. Weissman. Active threads: An extensible and portable light-weight thread system. Technical Report ICSI TR-97-036, International Computer ScienceInstitute, University of California at Berkeley, October 1997.