University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Toward a Software Infrastructure for the Cyclops64 Cellular Architecture

Juan B. del Cuvillo
Ziang Hu
Weirong Zhu
Fei Chen
Guang R. Gao

CAPSL Technical Memo 55
April 26, 2004

Copyright © 2004 CAPSL at the University of Delaware

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

This paper presents an initial design of the Cyclops64 (C64) system software infrastructure and tools under development as a joint effort between IBM T.J. Watson Research Center, ETI. Inc. and the University of Delaware. The C64 system is the latest version of the BlueGene/C supercomputer architecture that consists of a large number of compute nodes each employs a multiprocessor-on-a-chip architecture with 160 hardware thread units. The first version of the C64 system software has been developed and is now under evaluation. The current version of the C64 software infrastructure includes a C compiler, a runtime thread library, other tools for program execution control (linker/loader, program initiation and diagnostics software, etc.) and a function accurate simulator called FAST that can simulate a multi-node C64 system.

This paper is focused on the following aspects of the C64 system software: (1) the C64 Thread Virtual Machine (C64 TVM), its API (the CThread Library) and in particular, the key components of C64 TVM: the thread model, the memory model and the synchronization model; (2) the C64 software toolchain with emphasis on the CThread run-time system library, the first implementation of the C64 TVM; (3) the validation of the toolchain/FAST through both an extensive testing that ensures its stability and a case study, which demonstrates what a C64 architect or a software/application developer can expect to learn using the FAST tool.

Contents

1 Introduction **1**

2 Cyclops64 Cellular Architecture **2**

3 Cyclops64 Thread Virtual Machine **4**
3.1 The Thread Model 5
3.2 The Memory Model 5
3.2.1 Memory address space model 5
3.2.2 Memory Consistency Model 7
3.3 The Synchronization Model 9

4 Cyclops64 Software Toolchain **11**
4.1 Implementation Details 12

5 Results **13**
5.1 Experimental Platform 13
5.2 Results 14
5.3 Discussion 16

6 Related Work **16**

7 Conclusions **17**

8 Acknowledgements **18**

List of Figures

1 Cyclops64 Computing Environment 3
2 Cyclops64 Node 3
3 Cyclops64 Supercomputer 4
4 Producer-Consumer Sample Program 10
5 Barrier Sample Program 10
6 Cyclops64 Software Toolchain 11
7 Absolute Speedup 14

List of Tables

1 Coarse Instruction Mix 15
2 Tuning Experience Report for the MxM program 16

1 Introduction

The C64 is a petaflop supercomputer project under development at IBM Research Laboratory. C64 is designed to serve as a dedicated compute engine originally designed for running high performance application such as molecular dynamics to study protein folding, or image processing to support real-time medical procedures. C64 supercomputer is attached — through a number of Gigabit Ethernet links — to a host system. The host system provides a familiar computing environment (such as Linux) to applications software developers and end users. Besides access (thru the Ethernet links) to a common file server used for storing input and output data sets used and produced by application programs, each C64 chip can be connected to a serial ATA disk drive.

A C64 is built out of tens of thousands of C64 processing nodes. Each processing node consists of a C64 chip, external DRAM, and a small amount of external interface logic. A C64 chip employs a multiprocessor-on-a-chip architecture containing 160 hardware thread units, half as many floating point units, each shared by two thread units, on-chip SRAM, on-chip instruction cache, bidirectional inter-chip routing ports, and interface to off-chip DDR SDRAM. On-chip resources are connected to a crossbar network, which also provides threads access to the routing ports that connect each C64 chip to its neighbors arranged in a 3D-mesh configuration.

Based on our previous experience from the embedded Cyclops32 project [24, 9], we now present a system software architecture, encompassing components running on the host system and on C64 nodes, for system management and applications development and execution. We introduce a low-level program execution model (called the Cyclops64 Thread Virtual Machine). In other words, system software is an "extension" of the C64 ISA to implement the thread virtual machine. We streamline our discussion of the virtual machine by defining its API, called CThread. System software is illustrated by showing how some features/functions of the API are implemented under the overall system software architecture.

This paper is focused on the following aspects of the C64 system software:

- The C64 Thread Virtual Machine (C64 TVM) and its API (the CThread Library), which will be used as the common baseline for future research on parallel programming models. In particular we describe the three key components of C64 TVM: the thread model, where thread management issues are presented; the memory model that includes a presentation on both C64 memory address space and memory consistency model; and the synchronization model that provides the functionality to implement mutual-exclusion regions, perform direct thread-to-thread and barrier-type of synchronizations.

- The C64 software toolchain with emphasis on the CThread run-time system library, the first implementation of the C64 TVM. Specifically, we discuss how our Cthread implementation takes advantage of Cyclops specific hardware features to leverage performance without imposing any additional burden on the application programmer.

- The validation of the toolchain/FAST both through extensive testing and the matrix-

matrix-multiply program case study. Although performance tuning and optimization are not the objectives of this paper, we also report some initial performance observations to demonstrate what a C64 architect or a software/application developer can expect to learn using the FAST tool.

In Section 2 we will present an overview of the C64 system architecture including the C64 chip architecture, the communication networks connecting the system components, and the control network for system initialization and reconfiguration under the control of host software. In Section 3, we describe an outline of the C64 system software architecture and the C64 Thread Virtual Machine. In Section 4, we discuss C64 software tool chain and its implementation issues. In Section 5 we report initial experimental results. In Section 6 we briefly discuss related work. Conclusions are presented in Section 7.

2 Cyclops64 Cellular Architecture

The computing environment we are considering consists of a host and external file systems connected to a C64 supercomputer by means of a Gigabit Ethernet network, see Figure 1. The host system (shown as consisting of a number of control nodes and a front-end node) supports application programs development and execution, as well as system administration. The file system, which may also contain multiple (external) file server nodes, provides one means of file support for the C64 supercomputer. An internal high bandwidth distributed file system hosted by serial ATA hard drives attached to each C64 node will be also available to avoid disk bottlenecks and network congestion.

C64 nodes are arranged in a 3D-mesh network. A fraction of these nodes, labeled as I/O nodes, use the Gigabit Ethernet port (present in all C64 chips) to connect the C64 supercomputer to the host and external file systems. Each I/O node will service a number of C64 nodes, called compute nodes, and relay requests and data between the compute nodes and the host and file server systems. The I/O nodes and compute nodes communicate via packets over the 3D-mesh network only. This 3D-mesh provides the high bandwidth necessary for internode communication in running application programs.

There is a separate control network that connects the C64 system to the host system. This control network carries commands from the host system to each C64 node. A C64 node attaches to this control network via its JTAG interface. The host system uses this control network to initialize the C64 system, monitor its status while programs are in execution, and reconfigure and restart C64 after hardware failures. Details of the initialization and configuration procedures are not the focus of this paper and will be discussed elsewhere.

In Figure 2, we show the architecture of a C64 node. Each processing node consists of a C64 chip, external DRAM, and a small amount of external interface logic. Each C64 chip has 80 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

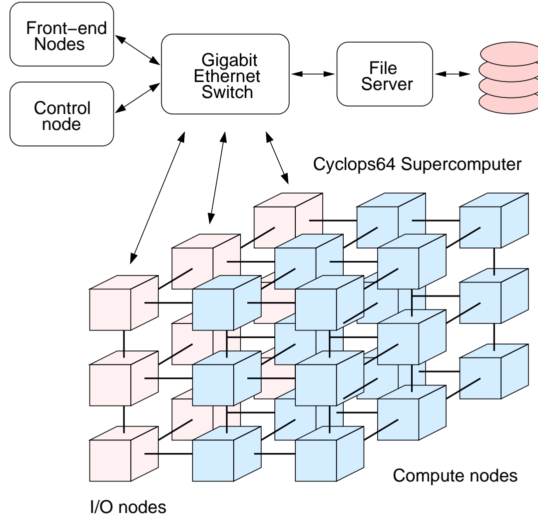

Figure 1: Cyclops64 Computing Environment

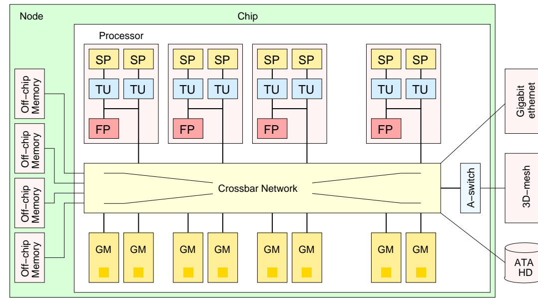

Figure 2: Cyclops64 Node

among five processors. In a C64 chip architecture there is no data cache. Instead a portion of each SRAM bank can be configured as scratch-pad memory. Such a memory provides a fast temporary storage to exploit locality under software control. Processors are connected to a crossbar network that enables intra-chip communication, i.e. access to other processor's on-chip memory as well as off-chip DRAM, and inter-chip communication via input and output ports that connect each C64 chip to its nearest neighbors in the 3D-mesh. The intra-chip network also facilitates access to special hardware devices such as the Gigabit Ethernet port and the serial ATA disk drive attached to each C64 node.

Finally, Figure 3 illustrates an instance of a C64 supercomputer architecture with $24\times24\times24$ logically arranged C64 nodes in the 3D-mesh configuration. Notice the physical distribution is somewhat different.

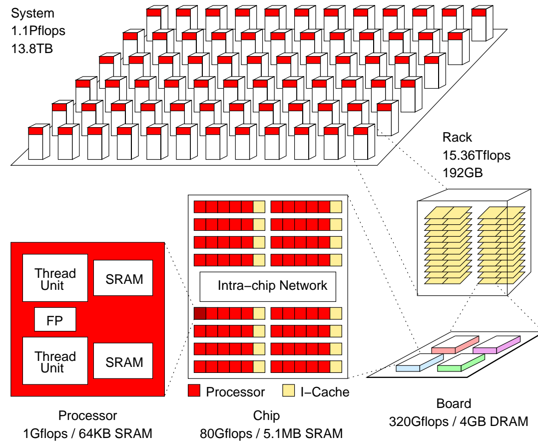

Figure 3: Cyclops64 Supercomputer

3 Cyclops64 Thread Virtual Machine

The main objective behind the C64 chip design is to build a petaflop computer by scaling up some millions of simple processing elements. On C64 the computation cell is the thread unit, a simple 64-bit in-order RISC processor core with a small instruction set architecture (60 instruction groups) operating at a moderate clock rate. The two main distinct hardware features that distinguish Cyclops from other general purpose processors are: (1) user and supervisor execution modes supported by the C64 processor in addition to a set of exceptions triggered by predefined events, provide the mechanisms required for protection. However, execution is non preemptive. That means the OS will not interrupt the user program running on a thread unless the user explicitly specifies so or an exception occurs; (2) 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 on the chip.

As we described in the introduction, one important role of the C64 system software is to implement a C64 virtual machine. This virtual machine, called C64 Thread Virtual Machine, can be viewed as a multichip multiprocessor extension to the base C64 instruction set architecture. In this section, we present an outline of the C64 Thread Virtual Machine along with the CThread (Cyclops Threads) run-time system library. The CThread library has been designed and developed to support a multithreaded programming model for a cellular multithreaded architecture such as Cyclops. Given C64 special features described earlier, it was not our intention to develop an OS for this platform that would put a considerable overhead on top of a machine that is aimed for simplicity from the bottom up. Instead we decided to implement CThread directly on top of the hardware architecture as a micro-kernel/run-time system library that takes advantage of C64 hardware features while provides an interface that shields application programmers and system software developers from the complexities of the architecture wherever possible.

C64 Thread Virtual Machine includes three components: a thread model, a memory model and a synchronization model, as well as their corresponding APIs. In Section 3.1, we introduce the thread model. This is partly derived on our experience from our earlier work on a thread model for the embedded Cyclops32 project [24, 9]. In Section 3.2, we introduce the memory model that includes the specification of a shared address space model (both intra-node and inter-node) and memory consistency model for a C64 system. In Section 3.3, we present the synchronization model. The memory model and synchronization model can be viewed as an extension of the base thread model — hence the name CThread may include all three in the rest of this discussion.

3.1 The Thread Model

A program section can be declared as a thread. A thread can be activated for execution by binding to a hardware thread unit within a certain chip, a thread activation pointer 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. stack pointer, etc.)

A thread activation pointer can also be "global" if the thread handler is extended with a node (or chip) identifier — 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.

The functionality provided by the present version of the CThread library is very limited. It provides basic functions for creating, synchronizing and terminating threads; the minimum functionality required to write multithreaded programs. For this first release, an interface inspired on that of the popular PThread model is provided to ease the first hands-on experience of application and system software developers.

3.2 The Memory Model

In this section, we describe the memory model and its API supported by the first version of the CThread library.

3.2.1 Memory address space model

First, we describe the memory address space model of a single C64 chip. Then, we outline how we intend to extend the memory address space model to the entire C64 system.

- Memory Address Space Model Within a Single Cyclops64 Node

  CThread assumes a memory address space model closely associated with the underlaying C64 architecture.

The C64 chip hardware supports a shared address space model: all on-chip SRAM and off-chip DRAM banks are addressable from all thread units/processors on the same chip. That is, all threads see a single shared address space. On-chip SRAM memory space is limited in the current technology to 5.2MB — so it should be viewed and used as temporary storage during computation. There is no hardware data cache used in the C64 design. Off-chip DRAM should be considered as the main memory.

Architecturally, each thread unit has an associated 32KB SRAM bank. Each memory bank can be partitioned (configured) into two sections: one called "global" (or "interleaved") section, the other "local" (or "scratch-pad") section. The partition boundary is identical across the thread units, and is set at the hardware initialization time and cannot be altered under program control. All such "global sections" together form the (on-chip) "global memory" in an interleaved fashion that are uniformly addressable from all thread units. Logically, the address field of the global memory ranges from 0 to a maximum value, which depends on how the SRAM banks are configured (partitioned), and how many SRAM banks are available and functional [1]. The absolute maximum address is $160 \times 32KB$ (about 5.2MB). This range of address is organized such that it is free of holes.

Of the 64-bit address computed by a C64 processor, only 32 bits are needed to address all node resources. Although the high-order 32 bits are currently ignored, we anticipate these extra bits will be used to emulate a global shared memory address space across chips. The low-order 32 bits are used to access C64 on-chip SRAM, off-chip DRAM and memory-mapped special devices (e.g. interthread interrupt, wakeup signal and input/output ports). The most significant bit is used to distinguish DRAM from SRAM and other devices. When the first bit of the address is 1, it represents an address to the off-chip DRAM. The first bit 0 represents an address to on-chip SRAM and/or other devices. The second most significant bit is used to distinguish on-chip global (or interleaved) SRAM from scratch pad memory and other devices/ports. When this bit is 0, it denotes on-chip global SRAM. Otherwise, it denotes scratch pad memory and/or other devices/ports.

All local (scratch-pad) sections are globally (but non-uniformly) addressable by all thread units. The addressing scheme to scratch pad memory is such that the address consists of a processor identifier, a thread identifier and an offset field. The access is directed to the scratch pad section associated with the thread unit given by the processor and thread identifiers.

The memory space reserved for off-chip DRAM is up to 2GB. Presently, only 1GB of DRAM is installed.

- Memory Address Space Model for a Multinode C64 System

  Under the current C64 design, there is no hardware architecture support for shared address space between C64 chips. Therefore, it is up to the C64 system software to support a

[1] C64 chips are used even if not all thread units are perfect, resulting in very low cost

shared address space across chips. However, an efficient implementation would need hardware architecture support as well. One straightforward extension is to associate a node identifier to the on-chip address space. Initially 14 bits seems to be enough to provide a system-wide identifier to each of the 13,824 nodes the first C64 system will consist of.

Since C64 memory hierarchy is explicitly exposed to the users — i.e. on-chip SRAM (global and scratch-pad) and off-chip DRAMs associated with a chip or elsewhere in the system — CThread provides services that can be used to move data across regions of the memory space. Accordingly, CThread include functions such as memory put and memory get operations that can be used to perform such data movements.

3.2.2 Memory Consistency Model

As in its address space model, CThread employs a memory consistency model close to the underline C64 architecture support.

The most widely accepted memory model for the multiprocessor machine is Lamport's sequential consistency (SC) model. It was described by Lamport in the following well-known statement:

> [A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program order [19].

The above quote becomes the commonly used definition of sequential consistency in most textbooks and research papers.

Under the current C64 single-chip architecture design, the following two conditions are valid:

1. Each processor issues memory requests in the order specified by its program.
2. Two operations designated to the same memory module M will be delivered to M's input FIFO queue in the same order as they entered into the network.

Notice the latter refers to the time a memory request enters into the network, not when it is issued by a processor, and it is true due to the equal-latency property of Cyclops' intra-chip network.

It has been shown that the above two conditions are sufficient to ensure that the C64 architecture behaves as sequentially consistent [31]. This also suggests that Cyclops designer's conjecture is true: the C64 architecture is sequentially consistent and there is no need to issue fence-like instructions after each memory operation to ensure SC [2].

[2] In fact C64 has no sync instruction.

However, hardware cannot guarantee a "Lamport order" of the accesses to the scratch-pad memory space — hence no sequential consistency can be assumed. We will come back to discuss its impact on our programming model after we present our CThread memory model.

Below we outline the memory model that is assumed in our current thoughts on CThread programming model, for which we use OpenMP as base programming model and the Location Consistency (LC) memory model [14]. In [25] the authors showed how the OpenMP memory model can be formalized by using the *pomset* abstraction in the Location Consistency memory model, so we will not repeat it here.

We will restrict ourselves here to a class of parallel programs that feature general fork-join nested parallelism as exemplified by programs written in a Single Program Multiple Data (SPMD) style. A considerable number of real world shared memory parallel programs have been written in OpenMP, hence our CThread model will use OpenMP as a high-level parallel programming model to be implemented on C64 architecture.

Each thread T has a private memory region, which can be used by T as its local storage for shared variables that reside in the shared memory space. The allocation and management of such thread private storage are implemented by C64 system software: both compiler and runtime software layer of CThread virtual machine. The movement between private and shared memory space can be implemented using memory put/get operations. The synchronizations needed to keep the consistency between shared and private memory will be discussed below as well as in the next section.

We will introduce a CThread operation called *cthread_flush* — inspired by the OpenMP *flush* directive, which will be used to ensure the consistency between private and shared memory. Flushes occur frequently in OpenMP programs, because they are performed implicitly at the end of many common OpenMP constructs such as barriers, parallel loops, parallel sections, critical sections, etc.

A *cthread_flush* operation executed at program point P requires that a certain set of shared variables, S, must be made consistent by the executing thread at point P, i.e. all read and write memory accesses that occur before point P must be performed/completed before the thread can advance past P, and all memory operations that occur after point P must be performed/completed after the thread advances past P. By default, S refers to all shared variables in the CThread program, though the user has the option of restricting the set of variables to be made consistent. The semantics of the *flush* operation have profound implications on the software and hardware implementation of a CThread program. For example, the compiler must ensure that any variable in S that is allocated to a register must be spilled to memory before point P and reloaded from memory after point P. Similarly, the hardware must ensure that all system allocated private storage for a thread T containing variables in set S are flushed before point P, and all private copies of the values for variables in set S are invalidated after point P. Note that a single thread's execution of a *cthread_flush* operation only supports consistency between the thread's private memory and main memory. Global consistency can only be achieved when all threads perform a *cthread_flush* operation.

Stronger memory models provide stricter guarantees on unsynchronized accesses to shared variables at the cost of additional burden on the implementation, especially when scaling up to large numbers of processors. For example, both strong and relaxed models include the memory coherence assumption, which states that all updates to the same shared variable must be observed in the same total order by all processors. When scratch-pad memory is considered, the overall memory model of a C64 chip is no longer sequentially consistent! However, it can be observed that the coherence assumption still holds. Therefore, CThread will not rely on a memory model that is based on hardware sequential consistency. Instead, CThread uses a weak model where the base assumption is memory coherence. One option is to use Location Consistency (LC) to formalize the semantics of CThread memory model so as to place no restrictions on scalable parallelism [14, 25]. Other alternative memory models are also under consideration.

The previous discussion was focused on a single node C64 system. When shared address space is implemented across C64 chips, we anticipate that a weak memory model (such as the one we discussed above for a single chip) will be used.

3.3 The Synchronization Model

The C64 hardware chip architecture supports direct memory access — loads and stores — to the shared address space covering the on-chip memory (SRAM banks, scratch-pad memories) and off-chip DRAM associated with the chip. When a global shared address space across chips is provided by the system, CThread will include services to implement remote load and store operations. All above operations are considered normal memory operations that do not involve synchronization.

Several types of synchronizations are supported in CThread.

A first type of synchronization is used to ensure mutual exclusion of memory accesses to shared memory locations/space. This can be expressed using CThread mutex lock and unlock operations, which are directly implemented using C64 hardware atomic test-and-set operations. Users can declare mutex variables using the CThread library and operate upon them with the functions provided for that purpose.

A second type of synchronization in CThread is introduced to express precedence relations between operations from two different threads. In the first version of CThread, we provide a coarse-grain signal-wait type of synchronization that will be placed between a pair of specific program points within the two threads.

The sample program in Figure 4, based on a producer-consumer model, shows the basic use of the signal/wait primitives. The producer thread produces data that is consumed by the consumer thread. The latter starts calling *cthread_wait* and blocks until a signal from the thread, whose thread handler matches that given as argument to the function, is received. The former produces a datum and sends a signal to the thread, whose thread handler is specified by *cthread_signal* only parameter. Once the signal is received, the consumer thread is awakened

```
void producer(void){
  while(1) {
    produce_data();
    cthread_signal(dest_thid);
  }
}

void consumer(void){
  while(1) {
    cthread_wait(src_thid);
    consume_data();
  }
}
```

Figure 4: Producer-Consumer Sample Program

```
void worker(int set_id){
  produce_data_set(set_id);
  cthread_barrier(grp_thid);
  if (set_id == 0)
    reduce_data_set();
}
```

Figure 5: Barrier Sample Program

and consumes the datum.

A third type is collective synchronization that will be participated by a group of threads. For example, a barrier synchronization primitive can be invoked by a group of threads. Threads block until all participants in the operation (participants are defined by a single object passed as parameter to the barrier function) have reached this routine.

The code in Figure 5 is an extract of a CThread program that uses a barrier primitive. Multiple threads execute the *worker* routine, which starts with each thread generating some data according to a thread-specific parameter. Once all the data has been generated, an unspecified operation is applied to it (in our example it is some type of reduction). Before the operation can be applied, we must ensure all threads have produced the corresponding data. For that purpose, we call the *cthread_barrier* function, so all threads block until all participants reached the same point.

A fourth type of synchronization operations is synchronized memory access operations. For example, we can associate sync operations with memory put/get operations — *cthread_getmem_sync(src, dst, len, tid)* and *cthread_putmem_sync(src, dst, len, tid)*, similarly to block data transfer operations on EARTH architecture [29]. Here, after a memory get/put operation is completed, it will signal the designated thread (denoted by the thread handler *tid*).

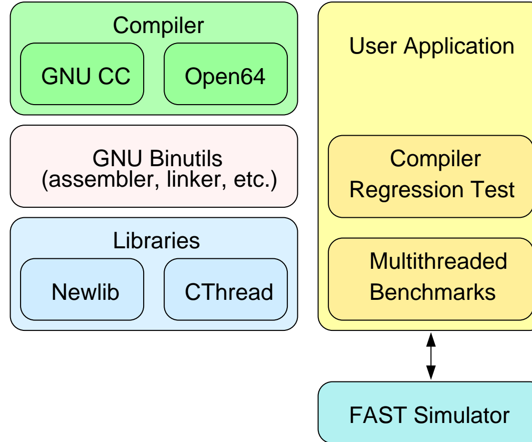

Figure 6: Cyclops64 Software Toolchain

4 Cyclops64 Software Toolchain

Figure 6 illustrates the software toolchain currently available for application development on the C64 platform.

The C/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 (libc/libm) are derived from those in newlib-1.10.0. We wrote our own runtime system, communication library and a functional accurate simulator. Additionally, a cycle accurate simulator is also being developed.

The GCC based C compiler supports C99 and most of the GCC extensions. As GCC has very good portability, this compiler version was delivered in a fairly short time period to provide developers with a basic experimental platform. The Open64 based version has most of the important features of a modern compiler, such as loop nest optimization, SSA-based global optimization, interprocedural analysis and optimization, and software pipelining. Another promising feature of Open64 is its ability to be extended, for instance for thread partitioning. However, it is still under development and only supports the O0 optimization level.

The toolchain supports the full C64 instruction set and it can be easily extended to fit our purposes. It supports segmented memory allocation, as C64 uses physical memory address space (instead of virtual memory space, because there is no hardware memory map unit). We can distinguish three memory regions in a C64 system: on-chip “global” SRAM, on-chip “scratch-pad” memory, and off-chip DRAM. For these three types of non-contiguous memory regions the toolchain supports memory allocation and management.

The library functions (libc/libm) are thread safe — multiple threads can call any of the functions at the same time. Shared resources have been protected by mutex.

The CThread runtime system library provides the software and application developer with

the functionality to write multithreaded programs: thread management, support for mutual exclusion, synchronization among threads, inter-node communication, etc. In order to achieve high performance and scalability, the implementation of such functionality tries to match as close as possible the architecture underneath the microkernel/RTS. It is not a coincidence that the first API for the thread model resembles that of PThread. PThread is a well-known parallel programming model that allowed us to write parallel programs (mainly for testing purposes) in quite a short time frame. However, such an interface is just a wrapper of the CThread inner core, which is under development to support more advanced programming models on the C64 platform.

The communication library (CNet), handles communication and synchronization between nodes. Remote memory operations are the foundation of all communication primitives. Additionally, the library provides means for synchronization, such as signal/wait and global barrier primitives.

To carry out our research until 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 memories, and the 3D-mesh network. RISC-like instructions such as integer, floating-point, branch and memory operations are modeled based on execution times expressed by x/d tuples, where x is the execution time in the ALU, and d represents a delay. The correct simulation of C64 special operations (*sleep* instruction, *wakeup* signal, *inter-thread* interrupt, etc.) requires of specific hardware status to be emulated. Timer alarm events and error conditions such as an illegal instruction are properly detected by the simulator, which then triggers the corresponding interrupt. Finally, given the appropriate command line options, FAST generates the execution trace and/or an instruction statistics report to help the software/application developer tuning and optimizing a program. Although FAST is not cycle accurate, we have shown it is useful for performance estimation.

4.1 Implementation Details

In this section we discuss a few details of the CThread implementation. Our first thread model does a direct mapping of software threads into hardware thread units. Upon initialization, a 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. A waiting thread (waiting on a external event/synchronization) goes to sleep and is woken up by another thread through a hardware interrupt/signal.

In C64, execution is non-preemptive. In other words, after a software thread is assigned to a hardware thread unit by the RTS, it will run in the hardware thread until completion. Furthermore, the RTS will not swap out a sleeping thread and reassign the idle hardware resources to another software thread (in the first release of the CThread library an operation such as *cthread_yield* is not available).

Thread units have the stack and a thread-unit-unique memory area placed in the scratch-pad memory. At the beginning of the thread-unit-unique area, four memory words have been reserved for a special purpose. The first is used as a lock to guarantee atomic transactions on the remaining three words. The thread unit status, enabled/disabled, and the software thread status, running/asleep, are stored in the second word. Words three and four hold the function address and optional argument specified by the user when a new thread is created. Upon completion the thread overwrites the optional parameter with the exit code. These are the default settings the RTS uses to initialize the thread specific information with (pointed by the status pointer of the thread activation tuple). In future implementations, this control structure will probably be preserved but in some cases the allocation could be moved to a different memory region (DRAM for instance). Then, the RTS will be allowed to swap in and out threads from the same hardware unit if needed, at some additional cost.

Besides thread management, CThread library provides a mutex object that allows mutual-exclusion via acquiring and releasing locks. As we mentioned earlier, in C64 there is no data cache. Spinning on a lock, waiting for it to be released does interfere with other threads (or at least with the threads that try to access to the same memory bank where the lock is) by generating traffic on the crossbar network. Hence, a thread that fails to grab a lock 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" memory area corresponding to the sleeping thread. Needless to say that sleeping and awaking CThread functions, which are based on the native *sleep* and *wakeup signal* instructions, both take a few cycles.

Barriers are implemented using the "Signal Bus" special purpose register. All the thread units on a chip are connected by an 8-bit bus, which is accessible thru read from and writes to this register. Let us assume the appropriate bit of the SIGB register is initially set to 1. Upon entering to the barrier, threads reset that bit to 0 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 propagates throughout the chip in a few cycles, providing a means for very fast global synchronization.

5 Results

5.1 Experimental Platform

C64 software toolchain runs on a Linux environment. We wrote three sequential programs and their corresponding parallel versions using CThread's API and ran the executables generated by GCC compiler version 3.2.3 on our Functional Accurate Simulator Toolset (FAST) to demonstrate that the toolchain is fully functional. The communication library is currently under evaluation. For this reason the benchmark programs used for this study run on a single C64 chip.

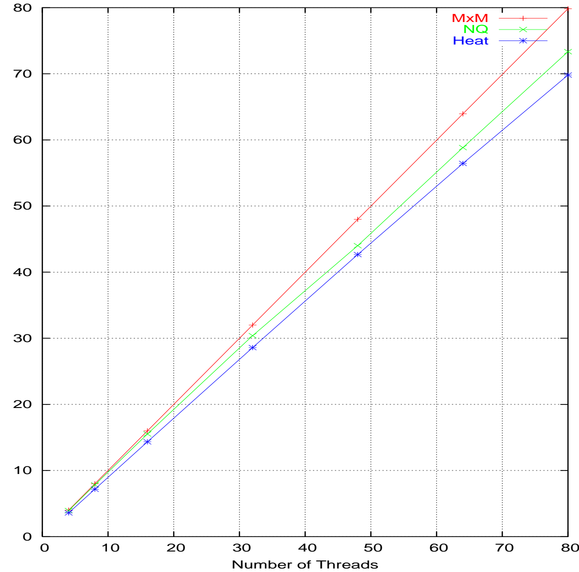

Figure 7: Absolute Speedup

The benchmark programs used in this study represent each a typical application class. These are:

- Matrix-matrix-multiply (MxM) is an embarrassingly parallel application on a shared memory system such as a C64 node. The result matrix is partitioned among threads which then carry out the computation with no dependences between any two threads.
- Nqueens (NQ) 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, the first three rows of the board are filled with queens in valid positions. After a number of independent tasks is created by this sequential search, tasks are then distributed among threads in a round-robin fashion.
- Heat is a finite differential based algorithm used to simulate the heat conduction over a solid plate. The surface of the plate is modeled as a $N \times N$ grid. The grid is initialized with one side of the plate to be heated and then simulate the heat transfer from this side to the whole plate. The program halts when a convergence condition has been reached. There is a barrier after each iteration. Hence, as the thread number increases so does the synchronization overhead.

5.2 Results

We run the experiments on the C64 simulator for the following problem sizes: matrix-matrix-multiple, 320×320; nqueens, 13; heat, 960×960. Figure 7 shows the absolute speedup achieved by each CThread program.

Program	Load/Store	Integer	Float	Branch
MxM	10.5%	79.2%	5.1%	5.2%
NQ	31.8%	36.8%	0.0%	31.4%
Heat	20.7%	73.00%	2.7%	3.6%

Table 1: Coarse Instruction Mix

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 MxM to achieve almost linear speedup even for 80 threads. We also like to highlight that despite having a global barrier after each iteration, the parallel version of Heat achieves a speedup of 70 on 80 threads.

The simulator can also produce abundant statistics information as well as a complete execution trace. For instance, Table 1 shows the percentage of each instruction class executed by the sequential programs.

As an example of what a C64 architect or a software/application developer can expect to learn using the FAST tool, we hereby report a tuning experience using the MxM program. Throughout this exercise we use the simulator's accurate time counter, the instruction statistics file and execution trace to measure the execution time, derive MIPS and MFLOPS numbers and determine the cause of delays and/or bottlenecks that may prevent the program from achieving higher performance. Table 2 summarizes the results obtained from the experience [3].

We start with a sequential implementation that follows the algorithm described in textbooks, and try different compiler optimization levels (rows 1–4 in Table 2). Based on the information generated by the simulator, we then proceed with the manual implementation of several optimizations. First, we change the variable where the matrix size is stored from integer (32 bits) to long long (64 bits). Although it uses 4 extra bytes of memory, it reduces the number of integer operations (row 5 in Table 2). Second we unroll the inner loop 4 and 8 times. The former resulted on an improvement (6th row in Table 2) while the later caused a performance drop due to register pressure and spilling (result not shown). At this point it was clear the main issue was the main memory (DRAM) latency access. We manually did loop interchange and tiling. We allocate in the stack (remember the stack is placed in the scratch-pad memory) blocks to where data is copied from memory first and then used for the computation. This software cache optimization brought us a significant improvement. However, the execution trace immediately revealed the implementation of *memcpy* is not optimized for the C64 architecture (row 7 in Table 2). We wrote our *memcpy* function emulating what the *ldm* instruction (*load multiple* was recently added to the C64 instruction set architecture) will be able to do. We tried data transfers of 8 and 16 consecutive double words (8th row in Table 2). At this point we realized we had removed all memory delays and that the program had hit the hardware thread

[3]Notice that FMAD instructions (floating-point multiply-add double) are reported as a single instruction in the second column, whereas it is counted as two floating-point operations in the MFLOPS column.

Optimization	Float	MIPS	MFLOPS
-O0	2.28%	261	6
-O1	3.77%	223	17
-O2	3.95%	241	19
-O3	4.47%	241	22
adjust var.	7.58%	204	31
unroll 4	8.55%	324	55
sw cache(memcpy)	20.00%	385	149
sw cache(ldm)	20.54%	456	181
2 threads	20.74%	913	363

Table 2: Tuning Experience Report for the MxM program

unit peak performance limit (500 MIPS). For the sake of completeness we parallelized our tuned MxM program and ran it on 2 thread units in an attempt to fully utilize the floating-point unit shared between threads. Again, the low overhead imposed by the CThread RTS allowed the program to double the performance (last row in Table 2).

5.3 Discussion

The purpose of the experiments was to show the usefulness of the toolchain. Previously, extensive testing have been performed to ensure the stability of the toolchain. In this paper, we have selected three programs to demonstrate what the software toolchain can provide a C64 architect or a software/application developer. Now that we have the toolchain, more interesting research may be carried out. For instance, we plan to study software cache strategies to get the best utilization of the on-chip memory. We may also study compiler optimization to partition a sequential program into a multithreaded program.

6 Related Work

As the semiconductor and VLSI technology rapidly advances to allow us to integrate a billion transistors on a single chip, it becomes important to exploit parallelism at all levels to utilize the chip capacity effectively [5, 6]. Within the last two decades, microprocessors achieved dominant success by exploiting instruction-level parallelism (ILP) [23] and improving memory access latency and bandwidth with multi-level memory hierarchy.

However, applications have inherent limits on ILP [30, 22]. To go beyond ILP, Hammond et. al. [17] suggested that simultaneous multithreading (SMT) and chip multiprocessor (CMP) can be two alternative architectures and CMP is preferred. Three different multiprocessor architectures are evaluated in [21] and Stanford Hydra CMP architecture is proposed [16]. Meanwhile, to overcome the memory bandwidth limitation of microprocessors [7], Processor-

in-Memory (PIM) is considered an effective way. Several PIM architectures are presented, like *Gilgamesh* [28, 27, 26], *Terasys* [15].

Prior work most relevant to Cyclops64 is the IBM BlueGene/C architecture [10, 1] and related system software research [2, 3, 8]. Almási et. al. demonstrated that a massive amount of parallelism can be exploited for applications, such as molecular dynamics, on the cellular architecture [2]. They also provided a detailed analysis on the cellular architecture and proved that several scientific kernels can reach the theoretical peak performance on different configurations of the architecture [3]. Caşcaval et. al. evaluated two important hardware features of the Cyclops cellular architecture: memory hierarchy with flexible cache organization and fast barrier synchronization and demonstrated the advantage of the single-chip multiprocessor with integrated multiple memory banks [8]. All the above work is based on a preliminary version of the BlueGene/C chip design, the Cyclops32 (C32), which is targeted to application-specific (embedded) domain.

The IBM BlueGene/L is another effort to deliver massive parallel system which exploits a more conventional system-on-chip technology, coupled with a highly scalable cellular architecture [12, 2, 1]. A BlueGene/L system consisting of 65,536 nodes is designed to deliver a peak performance of 180 to 360 teraflops. The system software architecture for BlueGene/L is arranged hierarchically according three levels: a computation core, a control infrastructure and a service infrastructure [4, 18]. There have also been several interesting evaluation studies of BlueGene/L system with regards to hardware performance monitoring [20] and scientific application frameworks [13, 11].

7 Conclusions

In this paper we presented the first version of the C64 system software. First, we described the C64 Thread Virtual Machine, as well as its key components: the thread model, the memory model and the synchronization model. The C64 software toolchain was also presented with emphasis on the CThread run-time system library, the first implementation of the C64 TVM. Finally, we demonstrated the correctness and usefulness of the toolchain/FAST simulator through three parallel benchmarks which results range from an almost linear speedup for an embarrassingly parallel applications to a speedup of 70 on 80 threads for a barrier-based program. As an example, we also tuned the matrix-matrix-multiply sequential code to show how the toolchain can provide the C64 system software and application developer with the insight to increase the performance from 22 to 363 MFLOPS (a 16.5 improvement!) on a single C64 processor.

8 Acknowledgements

We acknowledge Monty Denneau for his leadership, insightful advice and willingness to answer our numerous questions. The authors also acknowledge Alban Douillet for his effort porting the Open64 compiler to the C64 platform and for allowing us to borrow the diagram in Figure 2; Hirofumi Sakane, who patiently answered all our questions on Cyclops architecture and provided us the first draft of Figure 3; Yuan Zhang, her work on memory consistency models helped to write section 3.2.2; Clement Leung from ETI for his strong support during this project; Henry Warren for his timely evaluation of the toolchain and valuable feedback; the support from IBM Bluegene software team led by Manish Gupta and José Moreira – in particular José Castaños who has had several meetings and many email exchanges in sharing the experience of the C32 system software as well as BG/L and Christos Georgiou and George Chiu at IBM for their encouragement and support. We also acknowledge the strong support from our sponsors. Finally, we wish to thank many present and formal CAPSL members who have given us various support during this project without which the results presented in this paper would not be possible.

References

[1] F. Allen, G. Almási, W. Andreoni, and D. Beece et. al. Blue Gene: A vision for protein science using a petaflop supercomputer. *IBM System Journal*, 40(2), November 2001. Deep Computing for the Life Science.

[2] George Almási, Daniel K. Beece, and Ralph Bellofatto et. al. Blue Gene/L, a system-on-a-chip. In *2002 IEEE International Conference on Cluster Computing (CLUSTER 2002)*, pages 349–350, Chicago, IL, September 2002.

[3] George Almási, Călin Caşcaval, José G. Castaños, Monty Denneau, Derek Lieber, José E. Moreira, and Henry S. Warren, Jr. Dissecting Cyclops: A detailed analysis of a multi-threaded architecture. *ACM SIGARCH Computer Architecture News*, 31(1):26–38, March 2003. Special Issue: MEDEA workshop.

[4] George Almási and Ralph Bellofatto et. al. An overview of the Blue Gene/L system software organization. In *Ninth International Euro-Par Conference of Parallel Processing (Euro-Par 2003)*, volume 2790 of *Lecture Notes in Computer Science*, pages 543–555, Klagenfurt, Austria, August 2003.

[5] Doug Burger and James R. Goodman. Billion-transistor architectures. *IEEE Computer*, 30(9):46–49, September 1997. Guest Editors Introduction.

[6] Doug Burger and James R. Goodman. Billion-transistor architectures: There and back again. *IEEE Computer*, 37(3):22–28, March 2004.

[7] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, Philadelphia, May 22–24, 1996.

[8] Calin Cascaval, José G. Castaños, Luis Ceze, Monty Denneau, Manish Gupta, Derek Lieber, José E. Moreira, Karin Strauss, and Henry 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, Massachusettes, February 02–06, 2002. IEEE Computer Society.

[9] Juan B. del Cuvillo, Robert Klosiewicz, and Yingping Zhang. A software development kit for DIMES. CAPSL Technical Note 10, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, September 2003.

[10] Monty Denneau. Blue Gene. In *Proceedings of SC2000: High Performance Networking and Computing*, pages 35–35, Dallas, Texas, November 4–10, 2000. ACM SIGARCH and IEEE Computer Society. URL http://www.supercomp.org/sc2000/proceedings/.

[11] M. Eleftheriou, José E. Moreira, Blake G. Fitch, and Robert S. Germain. A volumetric FFT for BlueGene/L. *Lecture Notes in Computer Science*, 2913:194–203, 2003.

[12] Adiga et. al. An overview of the BlueGene/L supercomputer. In *Proceedings of SC2002: High Performance Networking and Computing*, Baltimore, Maryland, November 2002.

[13] B. G. Fitch, R. S. Germain, M. Mendell, J. Pitera, M. Pitman, A. Rayshubskiy, Y. Sham, F. Suits, W. Swope, and T. J. C. Ward et al. Blue Matter, an application framework for molecular simulation on Blue Gene. *Journal of Parallel and Distributed Computing*, 63(7–8):759–773, July/August 2003.

[14] Guang R. Gao and Vivek Sarkar. Location consistency – a new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 52(1):798–813, August 2000.

[15] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The Terasys massively parallel PIM array. *Computer*, 28(4):23–31, April 1995.

[16] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The stanford hydra CMP. *IEEE Micro*, 20(2):71–84, March 2000. IEMIDZ.

[17] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, September 1997.

[18] Elie Krevat, José G. Castaños, and José E. Moreira. Job scheduling for the BlueGene/L system. *Lecture Notes in Computer Science*, 2537, 2002.

[19] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[20] Pedro Mindlin, Jose Brunheroto, Luiz DeRose, and José E. Moreira. Obtaining hardware performance metrics for the BlueGene/L supercomputer. In *Ninth International Euro-Par Conference of Parallel Processing (Euro-Par 2003)*, volume 2790 of *Lecture Notes in Computer Science*, pages 109–118, Klagenfurt, Austria, August 2003.

[21] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 67–77, Philadelphia, May 22–24, 1996.

[22] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 24, October 1996; *Operating Systems Review*, 30(5), December 1996; *SIGPLAN Notices,* 31(9), September 1996.

[23] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7:9–50, May 1993.

[24] Hirofumi Sakane, Levent Yakay, Vishal Karna, Clement Leung, and Guang R.Gao. DIMES: An iterative emulation platform for multiprocessor-system-on-chip designs. In *Proceedings of the IEEEInternational Conference on Field-Programmble Technology*, Tokio, Japan, December 15–17, 2003. University of Tokio.

[25] Vivek Sarkar and Guang R. Gao. Analyzable atomic sections: Integrating fine-grained synchronization and weak consistency models for scalable parallelism. CAPSL Technical Memo 52, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, February 2004. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[26] Thomas Sterling. An introduction to the gilgamesh PIM architecture. *Lecture Notes in Computer Science*, 2150, 2001.

[27] Thomas Sterling. The gilgamesh MIND processor-in-memory architecture for petaflops-scale computing. *Lecture Notes in Computer Science*, 2327, 2002.

[28] Thomas L. Sterling and Hans P. Zima. Gilgamesh: A multithreaded processor-in-memory architecture for petaflops computing. In *Proceedings of SC2002: High Performance Networking and Computing*

[29] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill University, Montréal, Québec, May 1999.

[30] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2),

April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[31] Yuan Zhang, Weirong Zhu, Fei Chen, Ziang Hu, and Guang R. Gao. Lamport order revisit: A study on how to efficiently achieve sequential consistency on a modern multiprocessor-on-a-chip architecture. CAPSL Technical Memo 53, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 2004. In ftp://ftp.capsl.udel.edu/pub/doc/memos.