Programming Experience on Cyclops-64 Multi-Core Chip Architecture

Ziang Hu Geoff Gerfin Brice Dobry Guang R. Gao
Department of Electrical and Computer Engineering
University of Delaware
Newark, Delaware 19716, U.S.A
{hu,gerfin,dobry,ggao}@capsl.udel.edu

1. Abstract

In this paper, we present the design and implementation of a novel toolchain and runtime system which support segmented memory spaces for the IBM Cyclops-64 (C64) computer system - a multiprocessor-on-a-chip architecture. With the powerful support of the toolchain and runtime system, application developers can easily manipulate C64's explicitly visible memory hierarchy and tune the performance of their applications. In addition to the discussion of the toolchain, we use a set of benchmarks to show how to tune applications on the Cyclops-64 chip architecture to achieve high performance.

2. Introduction

High-end processor architectures are rapidly moving toward the integration of a large number of multiple processing cores on a single chip. Technology has the potential to put from 10-100+ processing cores on future generation microprocessor chips. Although this might be realized in successive stages: i.e. from 2, to 4, to 8 cores and up, it has already put a great challenge on parallel programming and system software development for such multi-core processors. It has been well recognized that a key to success for such multi-core chips is developing parallel system software technology that enables productive parallel programming on computers based on such chips.

This class of multi-core architecture has the following features: (1) it may employ a shared-global address space model while likely using a very different shared memory organization (e.g. PIM, MIP, etc.) from today's popular multi-level cache-coherent architecture modes; (2) it may employ a multithreaded execution model with efficient hardware support for thread management and inter-thread synchronization and in-memory atomic memory operations. A representative architecture we will use in this paper is the IBM Cyclops-64 architecture [8], where we had the chance to develop system software for it.

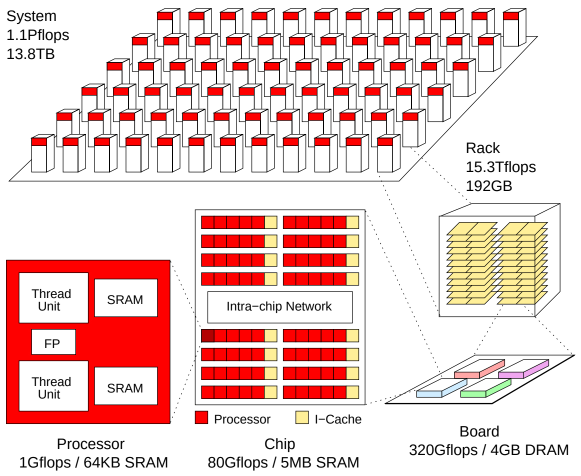

Figure 1. Cyclops-64 supercomputer

We have taken a bottom-up approach in the development of Cyclops-64 system software technology [8]. That is, a high-priority is given to the need of domain experts, enabling them to steer the machine towards high performance. To this end, we strive to provide a simple programming model and tools to achieve such goals. Although we wish to pursue long-term research in automatic compilation and optimization topics, our early system software release will expose performance critical architecture features through an API where the domain experts can directly make production use if they wish to. It is our hope that we will work closely with domain experts and learn from each other during the production algorithm/code development process. This is also the wish of our current users that they want to have the ability to manipulate the memory hierarchy by themselves and want to have the mechanisms to explicitly control data layout and memory movement.

The current C64 system software provides a platform that allows for various experiments on the novel architecture - as everything has been exposed.

So far, there is no claim that we have found the best pro-

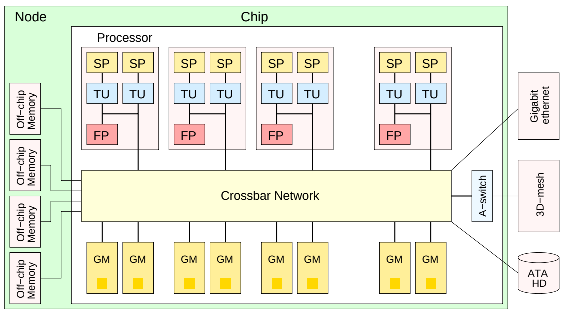

Figure 2. Cyclops-64 Chip Architecture

gramming model for C64 architecture. However, we can say that we have provided a solid foundation to explore a variety of programming models.

3. Cyclops-64 Chip Architecture

Cyclops-64 (C64) is a petaflop computer system under development by IBM, ET International Inc, and University of Delaware, as well as other collaborators. The basic building cell of the system is the C64 multi-core chip.

The work described in this paper focuses on a single C64 chip, see Figure 2. A C64 chip consists 80 processors, each with two thread units, a floating-point unit, and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the fi gure, is shared among fi ve processors.

Each on-chip SRAM bank will be confi gured into two segments - one is local to its corresponding thread unit as scratchpad memory (SPM), the other contributes to the globally shared on-chip memory (SRAM). There are 4 DRAM controllers that may connect to 4 off-chip DRAM banks (256MB each - 1GB in total). Each thread unit has a dedicated path to its local SPM, with very low access latency (1 cycle for store and 2 cycles for load). Other thread units may also access this SPM section, but have to go through the crossbar. To access SRAM, the memory request goes via the one-level crossbar. The load delay is 20 cycles and store delay is 10 cycles, in case there is no crossbar and memory bank contention. The SRAM bandwidth is 320GB/s. SRAM addresses are interleaved (block-cyclic) across the 160 banks and the block size 64 bytes. For DRAM, the load delay is 36 cycles and store delay is 18 cycles. DRAM bandwidth is 16GB/s. DRAM addresses are interleaved across 16 sub-banks (each DRAM bank consists of 4 sub-banks) and block size is also 64 bytes.

In addition to the complex memory hierarchy, the C64 architecture also incorporates effi cient support of thread level execution, including a sleep/wakeup instruction, abundant atomic memory operations and hardware barrier.

Effi ciently exploiting C64 architecture features is the key to achieve high performance.

4. C64 Tool-chain and Runtime

4.1 Compiler and Toolchain

Within the C64 software toolchain, the compiler can handle a set of directives (memory segment pragmas) in C code and generate segmented assembly code. By default, code is placed into off-chip DRAM and data is placed in on-chip SRAM. Using pragmas, data and code placement can be changed. Procedure foo() may be placed into on-chip SRAM by specify the following pragma in the program.

```
#pragma sram foo
```

Data can be put into SPM and DRAM by similar pragmas.

The compiler has been enhanced to generate correct code sequences to support multiple memory segments. For instance, when one procedure calls another procedure that is not in the same memory segment, pc-relative call instruction will not be used. Instead, we use indirect call instruction. Similar problems have been handled for data segments. Data in SPM is even special, as the semantics has been changed - from sharing between all the threads to thread private data (each thread has its own copy). The address becomes offset to the beginning of TLD (thread private section).

Object code format has been extended to support multiple code and data sections. New assembler directives have been added to support the multiple data/code sections. Linker has also been expanded to support the new sections.

Runtime library has been ported to the new platform, such that they are aware of the multiple layered memory hierarchy. For instance, memcpy() has been improved to handle memory copying between different memory segments. malloc() function has multiple versions, which supports memory allocation from different memory segments.

All the above enhancements provide users a powerful toolchain to tune their application performance on C64 chip architecture. Basically, user has all the control of the system memory hierarchy and is able to manipulate them.

4.2 C64 Runtime

We provide a TiNy Thread Programming Model (TNT) for single chip C64 applications. TNT API is similar to the POSIX Threads (PThread). However, TNT API incurs much lower latency compared to PThreads on conventional SMP machines. For instance, thread creation on C64 takes about 100 machine cycles while it takes millions of cycles on a normal SMP machine.

Different from conventional OS, C64 only supports non-preemptive thread scheduling at present. Whenever a thread is scheduled, it is binded to a thread unit until its termination. Therefore, there is no context switch necessary.

C64 library routines have been optimized to be aware of the memory hierarchy. For instance, spin-lock has been carefully designed to exploit the hardware features and be implemented in a very effi cient way.

Another example is memcpy(), it has been implemented to be aware of different memory segments of the source and destination addressed. By using load multiple(LDM) and store multiple (STM) instructions, and by pipelining the loads and stores, the optimized version is more than 20 times faster than the original implementation in newlib (in average).

5. Performance Tuning on C64 Architecture

The above section described the toolchain's support for manipulation of C64 segmented memory spaces. In this section, we will use three examples to show how to explore the different memory segments to achieve high performance for different applications.

The experiments have been done with FAST, which is a functionally accurate (in terms of architecture) simulator for the C64 architecture [7].

5.1 Tuning Scratchpad Memory

Each C64 thread unit has a dedicated scratchpad memory, which features very low access latency - 2 cycles for loads and 1 cycles for stores. The size of SPM is small compared to globally shared SRAM and off-chip DRAM. A typical partition, evenly split between SPM and SRAM, will give 16KB for each SPM. As the SPM is also used for the runtime stack, the available space for thread local data (TLD) will be even more limited. By default, the toolchain will allow at most 2KB for TLD data. Users may extend this limit by compiler and loader options. The user should be aware that when more space is requested for TLD, the stack space will be reduced.

Each SPM is independent of all other SPMs and their address spaces are not contiguous. SPM is good for applications that can partition the problem spaces into small and relatively independent segments.

Another important characteristic is that SPMs feature very large on-chip memory bandwidth, 160x4GB/s = 640GB/s.

A sample application that fi ts in SPM is the Monte Carlo simulation: a complex mathematical technique that estimates the probability of meeting specifi c goals in the future. In the context of fi nancial planning, the Monte Carlo simulation helps calculate the likelihood of successfully achieving goals, given investments set up specifi cally for those goals.

This problem is embarrassingly parallel and has little communication between multiple threads. The most important feature of this application is that the data set to be simulated is small and can be fully stored in SPM.

# Threads	1	2	4	8	16	32	64
GFLOPS	0.303	.0605	1.18	2.35	4.70	9.38	18.76

Figure 3. 128x128 Matrix Multiplication in SRAM

The simulated results show that one C64 chip is equal to 18 Opteron 250, 32 Intel Centrino or 28 Intel P4 3.2GHz Processors . (see [8] for more details).

5.2 Tuning On-chip Shared SRAM

The on-chip globally shared SRAM also has huge bandwidth (320GB/s each direction) due to the one-level crossbar support. Linear or near linear speedup can be achieved when memory accesses for a parallel application are balanced. The only issue with SRAM is the access latency (20 cycles for loads and 10 cycles for stores). Traditional compiler optimizations can be applied here to hide the latency. Loop unrolling may increase the size of basic blocks and enable more instructions to be scheduled by the instruction scheduler. Loop tiling (register tiling) may be exploited to increase data reuse in registers and reduce memory accesses, and therefore reduce data access latency.

The Fast Fourier Transform (FFT) is a well known algorithm widely used in a variety of areas, such as signal processing. On C64 we implement a one- dimensional double-precision complex transform whose data set fi ts into interleaved memory. A 2^{16}-point FFT scales up to 150 threads, at which point it delivers almost 20 GFLOPS.

We looked at the results published in http://www.fftw.org for other FFT implementations. We found that no conventional microprocessor can achieve a performance in excess of 4 GFLOPS.

Another example is dense matrix multiplication. In Figure 3, three matrices (size 128x128) are stored in SRAM. Due to the huge on-chip memory bandwidth and balanced memory access of the problem, linear speedup is achieved from 1 to 64 threads (Figure 4).

5.3 Tuning Off-chip DRAM

Any applications containing data or code sets that cannot fi t into on-chip memory must use the off-chip DRAM. Like other high performance processors, the memory wall is always the key issue. DRAM bandwidth becomes the major bottleneck, especially when compared to the massive computation power of the C64 chip (80GFLOPS floating

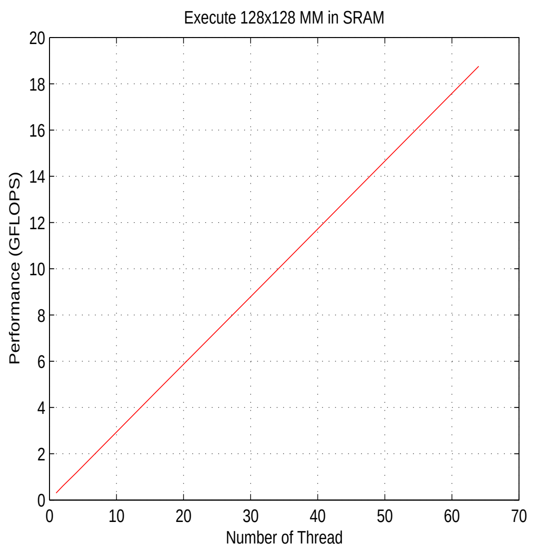

Figure 4. Performance Curve for 128x128 MM in SRAM

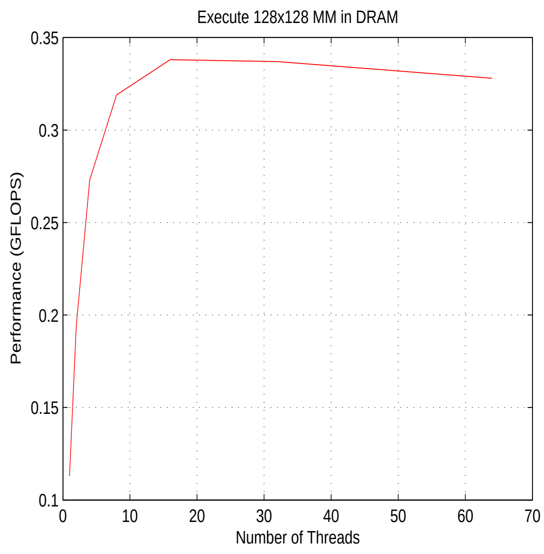

Figure 6. Performance Curve for 128x128 MM in DRAM

# Threads	1	2	4	8	16	32	64
GFLOPS	0.113	.0194	0.273	0.319	0.338	0.337	0.328

Figure 5. 128x128 Matrix Multiplication in DRAM

# Threads	1	2	4	8	16	32	64
GFLOPS	0.30	0.60	1.17	2.33	4.46	8.32	14.22

Figure 7. 256x256 MM in DRAM with Buffering

point peak performance, and 80GOPS fixed point peak performance). DRAM bandwidth is 16GB/s in the current C64 chip design, which means 32Bytes per cycle (peak DRAM bandwidth). It can supply 4 thread units with one double-word (8 bytes) per cycle. If all 160 thread units access a DRAM bank at the same cycle, most of them will be stalled.

The key optimization here is to reduce the bandwidth requirement to DRAM for user applications. Without bandwidth optimization, applications may not achieve good performance. For instance, when matrices are stored in DRAM, the 128x128 matrix multiplication gets very poor performance (see Figure 5 and Figure 6).

One technique to get around the DRAM bandwidth problem is to explore the on-chip memory bandwidth. By buffering data/code into on-chip memory and maximizing the data/code reuse in on-chip memory, we can get very impressive performance results, even when data resides in DRAM. We tried a 256x256 matrix multiplication with data initially stored in off-chip DRAM. By double buffering data into on-chip SRAM (buffer 128x128 sub-matrices), the computations can then get data from SRAM. Three helper threads are employed to move data between DRAM and SRAM; one for each matrix. The final performance number (see Figure 7 and Figure 8) is very close to the case when matrices are stored in SRAM.

Matrix multiplication is one application that can achieve good performance with large data sets in DRAM. This is possible due to the large amount of data reuse in this application. For problems with little data reuse, such as streaming applications, we may need to take other approaches.

6. Related Work

The CELL processor [17] is similar to C64 in that both of them use local memory instead of cache. One distinct architecture difference is that CELL moves data/code between global memory and local memory via DMA, while C64 uses load/store instructions.

DMA can be faster when moving large chunks of regular data structures, but the C64 design has much more flexibility. DMA could be easily incorporated into the current C64 design since it is already used to transfer data between A-switch buffers and C64 memories.

Optimizations for the CELL architecture [13, 12] may

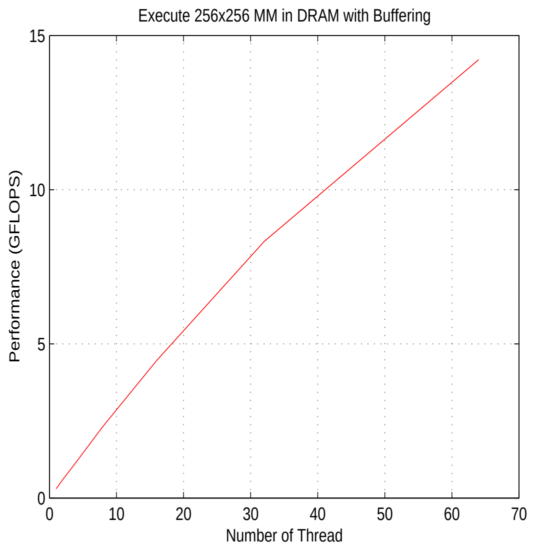

Figure 8. Performance Curve for 256x256 MM in DRAM with Buffering

also be applied to the C64 architecture, but the CELL toolchain has no support to statically allocate code or data into the local memory of each SPE.

Many embedded C compilers and toolchains support segmented memory spaces via language extensions, including attributes or pragmas. Among them, ”near” and ”far” pointer attributes are very popular. A ”near” pointer points to the same memory segment, while a ”far” pointer points to a distinct segment (or segments).

OnBoardC for Palm OS uses ”#pragma segment n” to start a new segment of code. Each segment will not directly access the internal code/data of other segments - only the function entry will be accessed (see http://onboardc.sourceforge.net for detailed information).

ISO working group (JTC1/SC22/WG14) proposes C language extensions to support multiple address spaces. Attributes are introduced to describe memory segments.

To support language extensions via attributes, the compiler may need considerable modifi cations while pragmas can be easily implemented with better portability. Language attributes, however, are much more strict and consistent in terms of semantics.

To get a fast prototype of the toolchain that supports segmented memory spaces, we use pragma instead of attribute, although the latter may be the choice of further development. The purpose is to provide users with quick access to the architecture so that they may tune their software on the C64 platform. More experience will allow us to choose and implement advanced programming models on top of the current toolchain.

Cache based locality optimizations have been extensively studied in the literature. Loop transformations have been investigated to exploit computation parallelism and data locality for scientifi c applications (see [1, 18, 5, 2, 20, 15] and their references). Loop tiling is a well known loop transformation used to increase cache locality (see [19, 18, 2, 3, 21] and their references). The feasibility of those optimizations to the C64 architecture needs to be re-examined.

Bandwidth optimization has also been extensively exploited in [4, 9, 14, 6, 16, 10, 11] and their references. Although most of these studies are targeted to cache-based systems, the ideas might be applied to C64 like architectures with an explicit memory hierarchy, which will be among our future research topics.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kauffmann Publishers, October 2001.

[2] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, New Mexico, June 23–25, 1993. *SIGPLAN Notices*, 28(6), June 1993.

[3] R. Andonov, H. Bourzoufi , and S. Rajopadhye. Two-dimensional orthogonal tiling: from theory to pratice. In *HiPC 1996*, Trivandrum, India, 1996.

[4] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, California, October 3–7, 1998. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News*, 26, October 1998; *Operating Systems Review*, 32(5), December 1998; *SIGPLAN Notices*, 33(11), November 1998.

[5] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, California, October 4–7, 1994. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News*, 22, October 1994; *Operating Systems Review*, 28(5), December 1994; *SIGPLAN Notices*, 29(11), November 1994.

[6] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure defi nition. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 13–24, Atlanta, Georgia, May 1–4, 1999. *SIGPLAN Notices*, 34(5), May 1999.

[7] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. In *Workshop on Modeling, Benchmarking*

and Simulation (MoBS'05) of ISCA'05, Madison, Wisconsin, June 2005.

[8] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Towards a software infrastructure for cyclops-64 cellular architecture. In *HPCS 2006*, Labroda, Canada, June 2005.

[9] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 229–241, Atlanta, Georgia, May 1–4, 1999. *SIGPLAN Notices*, 34(5), May 1999.

[10] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Parallel and Distributed Computing*, 64(1), 2004.

[11] C. Ding and M. Orlovich. The potential of computation regrouping for improving locality. In *SuperComputing 2004*, Pittsburgh, PA., November 2004.

[12] A. Eichenberger and E. Al. Optimizing compiler for the cell processor. In *ICS 2005*, Cambridge, Massachusetts, USA, May 2005.

[13] A. Eichenberger and E. Al. Using advanced compiler technology to exploit the performance of the cell broadband engine (tm) architecture. *IBM System Journal*, 45(1), Jan. 2006.

[14] K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4), 1998.

[15] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affi ne transforms. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 201–214, Paris, January 15–17, 1997.

[16] N.Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5), 1999.

[17] D. Pham and E. Al. The design and implementation of a fi rst-generation cell processor. In *ISSCC 2005*, Piscataway, NJ, Feb. 2005.

[18] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, June 26–28, 1991. *SIGPLAN Notices*, 26(6), June 1991.

[19] M. Wolfe. Iteration space tiling for memory hierarchies. *(SIAM) Parallel Processing for Scientific Computing*, pages 36–361, 1987.

[20] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing, Boston, MA, 1995.

[21] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.