# Optimizing NANOS OpenMP for the IBM Cyclops Multithreaded Architecture

David Ródenas, Xavier Martorell
Eduard Ayguadé, Jesús Labarta
CEPBA-IBM Research Institute, UPC
Jordi Girona 1-3, 08034 Barcelona, Spain
{drodenas,xavim,eduard,jesus}@ac.upc.es

George Almási, Călin Caşcaval
José Castaños, José Moreira
IBM T.J. Watson Research Center
Yorktown Heights, NY
{gheorghe,cascaval,castanos,jmoreira}@us.ibm.com

## Abstract

*In this paper, we present two approaches to improve the execution of OpenMP applications on the IBM Cyclops multithreaded architecture. Both solutions are independent and they are focused to obtain better performance through a better management of the cache locality. The first solution is based on software modifications to the OpenMP runtime library to balance stack accesses across all data caches. The second solution is a small hardware modification to change the data cache mapping behavior, with the same goal. Both solutions help parallel applications to improve scalability and obtain better performance in this kind of architectures. In fact, they could also be applied to future multi–core processors. We have executed (using simulation) some of the NAS benchmarks to prove these proposals. They show how, with small changes in both the software and the hardware, we achieve very good scalability in parallel applications. Our results also show that standard execution environments oriented to multiprocessor architectures can be easily adapted to exploit multithreaded processors.*

## 1. Introduction

Every day the number of transistors in a single chip increases, allowing to include more resources on chip. At the same time, it is more difficult to increase resource utilization through out–of–order execution, multiple instruction issue, and deeper pipelines. For that reason companies have started production of dual-core processors with multithreading capabilities (like the current IBM Power5 [5]) and processors with more than two cores in a chip (as advocated by Intel [10]).

The IBM BlueGene/Cyclops (*BG/C*) architecture [2, 3, 7] provides us with a look ahead to future multi-core machines. BG/C simplifies the architecture and hardware organization and reduces the von Neumann bottleneck. Instead of building a more complicated memory hierarchy it seeks to hide latency by integrating many thread units – essentially, simple processor cores – into the same chip sharing memory. This kind of architecture is usually a good platform for the execution of applications with high degrees of parallelism, mainly scientific and engineering simulations.

In these applications the programmer faces the complexity of managing parallelism at application level: handling work distribution, data movement and synchronization. Although a number of parallel programming models have been proposed in the last years, we believe that the OpenMP [16] standard is a simple way to parallelize numerically intensive applications: it assumes shared memory and uses a few directives to express parallelism, work distribution, and synchronization to achieve near optimal performance.

In order to prove the adequacy of the OpenMP approach we have ported the NthLib user–level thread library [15] to Cyclops, and used the OpenMP NanosCompiler [8] to generate Cyclops code for OpenMP applications. The NanosCompiler generates source code with calls to the NthLib runtime library. Our tools rely on the Cyclops developer toolkit and simulator to evaluate performance. An initial evaluation of the porting was presented in [1].

This paper proposes two different kinds of optimizations (software and hardware) to run applications on the Cyclops architecture. The paper is organized as follows: Section 2 briefly discusses the IBM Cyclops multithreaded architecture. Section 3 presents the OpenMP Nanos execution environment, and Section 4 shows an initial performance evaluation of this environment. Section 5 shows how to take advantage of the Cyclops architecture applying some software optimizations into the NthLib runtime library. Section 6 presents a simple hardware optimization to improve thread stack locality. Section 7 discusses the related work. Finally, Section 8 concludes the paper and Section 9 presents future work.

© 2005 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI 10.1109/IPDPS.2005.317

## 2. The BlueGene/Cyclops architecture

Cyclops is a multiprocessor system-on-a-chip that uses processor-in-memory (PIM) design for combination of main memory and processing logic into a single piece of silicon. The chip (Figure 1) has a large number of simple (single-issue, in-order-execution) thread units. Floating-point units and cache memory are shared among groups of multiple thread units.

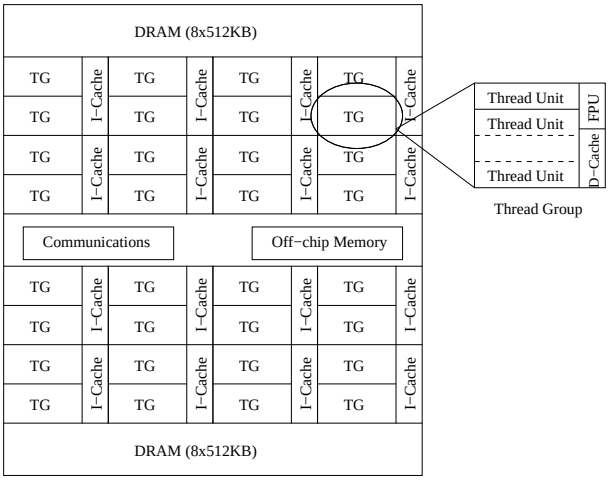


**Figure 1. Block diagram of the Cyclops architecture**

Each Cyclops thread unit has its own decoder, instruction sequencer, register file and fixed point unit. Thread units can issue an instruction in every cycle in program order if resources are available and there are no dependences with previous instructions. Instruction completion may be out-of-order.

Cyclops thread units come in groups of four, called quads. The thread units in the quad share the quad's data cache and floating point unit. Each thread unit can dispatch a floating point operation (addition, multiplication or fused multiply-add) in every cycle (subject to resource availability). The floating point unit cannot be accessed by thread units outside its quad.

Memory is distributed on two hierarchical levels ( Figure 2): main memory and caches (instruction and data). There are multiple banks of on-chip main memory shared between all threads and the latency to any bank is uniform. Memory addresses are interleaved between different memory banks. The hardware does not support paging, so logical memory maps directly into physical memory. Two quads share an instruction cache and they cannot have access to other instruction caches. In addition, each thread has a Prefetch Instruction Buffer (PIB) to improve instruction fetching.

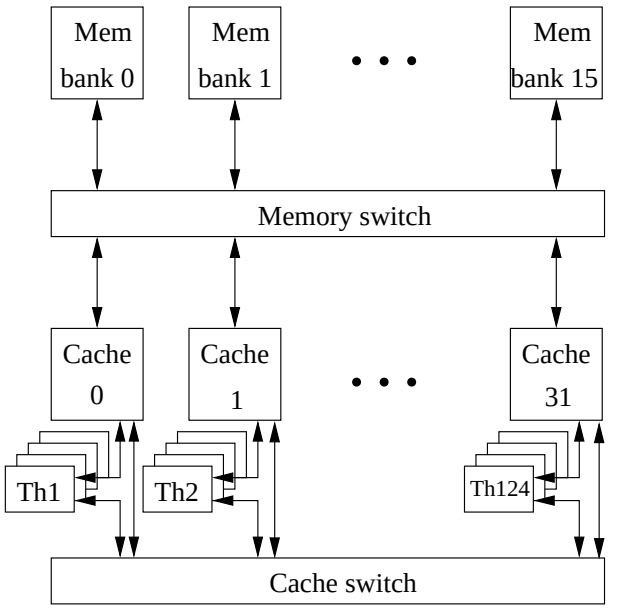


**Figure 2. Cyclops memory hierarchy**

Any thread unit in the chip can access data stored in any of the data caches. Threads units have faster access to their local data cache than to remote caches. The hardware ensures that each main memory address maps directly to a single data cache where it can be accessed by any thread. The hardware uses a static bijective scrambling function to map memory addresses to the cache. We will discuss the particulars of this mapping function later in the paper.

The BG/C environment provides a complete set of tools targeted to work with Linux Red Hat 7 and gcc 2.95.3. Components provided are the simulator of the processor, a kernel, an external debugger, a subset of the standard C library, and a cross–compiler based on the GNU toolkit (gcc 2.95.3 and binutils 2.11.2). All these tools have been re–targeted for the BG/C instruction set architecture.

The Cyclops simulator allows many configurations with different memory sizes and thread units count. Table 1 shows the actual simulation parameters used in all the experiments presented in this paper.

## 3. OpenMP for BG/C

OpenMP [16] is an application programming interface that may be used to explicitly direct multithreaded, shared memory parallelism. It is based on compiler directives, runtime library routines and environment variables. OpenMP uses the fork/join model of parallel execution. This means

(a) Memory latency

| Memory access type | Execution | Latency |
|---|---|---|
| Local cache hit | 1 | 6 |
| Local cache miss | 1 | 24 |
| Remote cache hit | 1 | 17 |
| Remote cache miss | 1 | 36 |

(b) components

| Component | # units | Params/unit |
|---|---|---|
| Quads | 32 | 4 threads, 1 FPU, 1 data cache |
| Threads | 128 | single issue, in-order, 500 MHz |
| FPUs | 32 | 1 add, 1 multiply, 1 div/sqrt |
| Data-cache | 32 | 16KB, 8 way assoc., 64-byte line |
| Instr.-cache | 16 | 32KB, 8-way assoc., 32-byte line |
| Memory banks | 32 | 8MB each for a total of 256 MB. |

**Table 1. Actual simulation parameters used in the experiments done in this paper**

that there are two kinds of threads: the main thread and slave threads. Execution of OpenMP programs starts in the main thread. The slave threads are spawned when a parallel region directive is found. Work distribution directives are used to distribute the work among all participating threads; OpenMP supports loop and task parallelism. Implicit and explicit synchronization mechanisms are provided to ensure the correct behavior of the parallel execution.

OpenMP for BG/C is built on the NanosCompiler and the NthLib components. The OpenMP NanosCompiler is a source-to-source translator for Fortran77 based on Parafrase-2 [17] that replaces OpenMP directives by calls to NthLib runtime. This translator is platform independent. The translation step is performed in a Linux workstation. The NthLib Runtime is designed to provide an efficient support to the OpenMP execution model on shared-memory multiprocessors. This library is implemented efficiently to support fine grain parallel tasks. The execution environment offers a master thread and as many slave threads as Cyclops threads are available. Applications start executing in the master thread. OpenMP directives cause slave threads to be activated. The master thread controls parallelism by providing a shared work descriptor [14] specifying the slave threads involved in the computation. The mapping of slave threads to Cyclops thread units is supplied by the user, allowing control of e.g. how FPU use is distributed and what thread units will have to share resources. The NthLib Runtime creates a fixed and aligned stack for all threads, mapped on top of the stack provided by Cyclops for the hardware thread.

# 4. Initial Performance Evaluation

## 4.1. Benchmarks

In our previous work we had ported to Cyclops, and evaluated, a set of micro benchmarks and a subset of NAS benchmarks, version 2.3 [4]. For this paper we have ported the NAS 3.0 benchmark suite [11] to the OpenMP environment for BG/C with no architecture dependent optimizations and we have evaluated its performance and speedup. We simulated classes S and W: for every benchmark we chose the largest class that could be simulated subject to Cyclops architectural limitations (memory size) and simulation time. In a further bid to keep simulation time reasonable we reduced the number of iterations for some of the class W benchmarks.

We analyzed the results taking advantage of detailed performance and resource use data provided by the simulator.

| Benchmark | Class S Size (nIter) | Class W Size (nIter) |
|---|---|---|
| LU | 12 (50) | 33 (300) |
| MG | 32 (4) | 64 (40) |
| SP | 12 (100) | 36 (400) |
| BT | 12 (60) | 24 (200) |
| CG | 7000 (15) | 14000 (15) |

**Table 2. NAS benchmarks characteristics**

Table 2 shows the characteristics of the NAS benchmarks used in the evaluation of this paper. Two different classes of the benchmarks have been used. *Class S* is the smallest benchmark dataset size, usually used for testing purposes only. *Class W* is the next class in size and we use it as a reference in our simulations. The simulation time of the larger classes of the NAS benchmarks (classes A, B, C and D) take too long and most of them do not fit in the Cyclops main memory.

The parameters provided in the table are the following:

- *Size* is the number of elements in the relevant dimension of the matrices. Parallelization is based on distributing this dimension across the different threads. As the number is usually small, sometimes the evaluation shows the effect of an unbalanced distribution of iterations among threads.
- *nIter* is the number of iterations in the outer (sequential) loop, usually representing a time–step.

### 4.2. Analyzing the current results

In [1] the authors evaluated the performance of some of the NAS benchmarks on Cyclops. The performance metric used to evaluate performance is speedup with respect to the sequential execution.

The results of this previous evaluation are shown in Figures 3 and 4. They present the scalability of LU, MG, and SP (class S) and LU, MG, SP, BT and CG (class W), respectively. Notice that with class S, applications do not scale beyond 16 threads. For class W this point is moved up to 32 threads. Figure 4 also shows that the applications SP and BT suffer from unbalanced work distribution when run with 16 and 32 threads. For instance, when SP (class W) distributes 36 columns of the matrices among 16 or 32 threads, there are always 4 threads executing more work than the others. However, when running on 36 threads unbalance is reduced and performance increases.

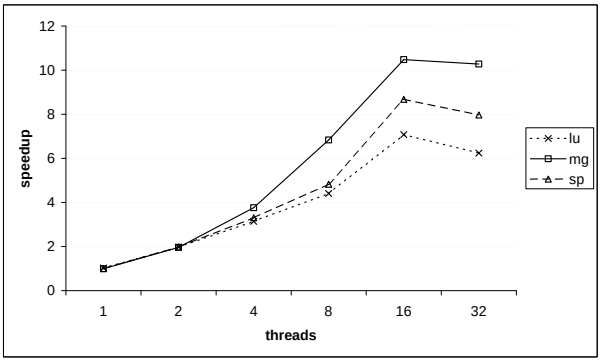


**Figure 3. Initial performance of the NAS Benchmarks (class S)**

We suspected that the poor scalability could be due to memory problems, in particular how data caches are used during the execution. If we take a look at the data cache access patterns, we will find a characteristic signature for each application and data size. Changing the number of threads, we will observe that some data caches, like the ones holding thread stack data, will increase the number of accesses, and others, like the ones holding code or global data, will stay mostly constant. The distribution among data caches is determined by the (previously discussed) scrambling function. Figure 5 presents the characteristic signature given by the MG benchmark with 4 and 64 threads. Program global data is usually distributed across all data caches. This is shown by all data caches being referenced around 2,500,000 times. The most used stack area of the master thread is located in the 7th cache. Slave threads stacks are mapped in the 28th cache. This causes a big unbalance between the number of

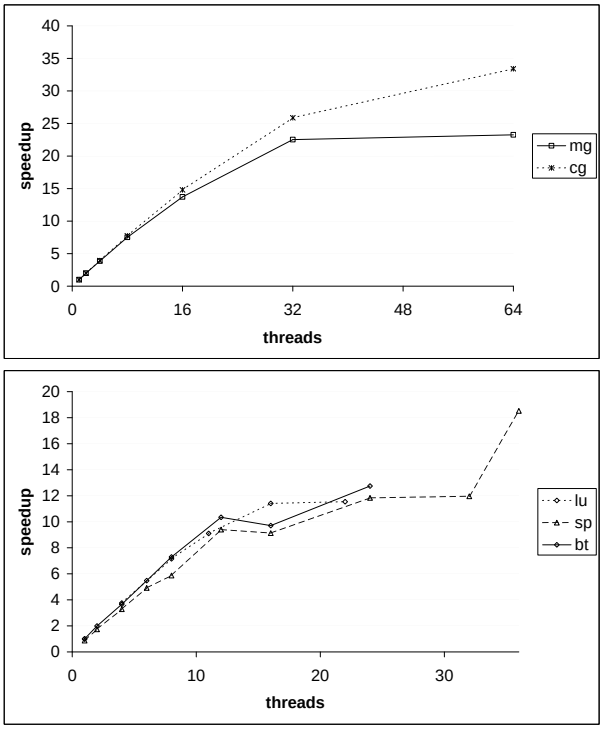


**Figure 4. Initial performance of the NAS Benchmarks (class W)**

accesses to the data caches.

The difference between the executions with 4 and 64 threads comes from the fact that the master thread reduces the amount of work done when going from 4 to 64 threads, but many thread stacks are mapped into cache number 28. As the amount of conflict misses increases, this causes a declining hit rate (as presented in Figure 6 for the MG benchmark), degrading application performance.

## 5. Optimizing the NANOS OpenMP runtime

The first optimization presented is a software optimization that is using padding to map NANOS thread stacks to different caches in order to avoid conflicts. The reason for this optimization is two-fold: i) OpenMP is based on a full shared–memory approach, and thus threads can not take advantage directly of the private stack data mapping; and ii) the default hardware scrambling function of the Cyclops chip maps all the thread stacks in a single cache, as described below.

Cyclops allows all data caches to be accessed as one large data cache of 512 KB. To reduce collisions, the hardware implements a scrambling function that uses the bits of each data address to determine the destination cache as follows: bits 0 to 5 determine the byte inside one line, bits 6

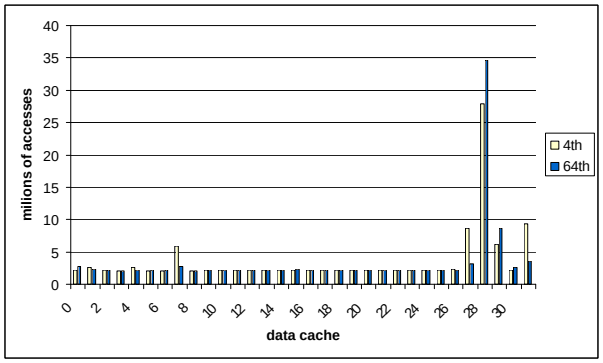


**Figure 5. Comparative of MG (class W) data cache accesses using 4 and 64 threads (baseline implementation)**

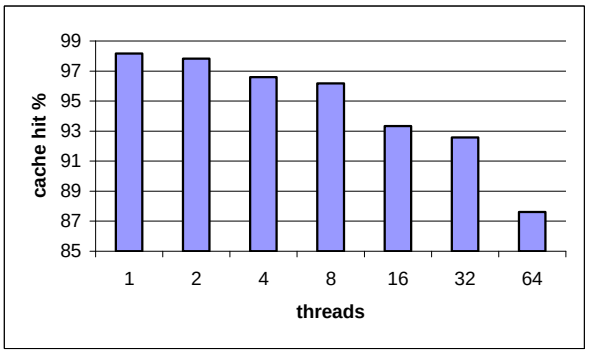


**Figure 6. MG (class W) data cache hits running with different number of threads (baseline implementation)**

to 10 determine the line inside the cache, and bits 11 to 15 to determine the data cache that owns the line. This function maps 2048 consecutive bytes of memory into the same data cache.

We have shown in the previous section that the number of accesses to a certain data cache grows rapidly when we increase the number of threads executing the application. When an application executes, the OpenMP slave threads run the same code, with the same calls and stack frames. In all executions the stacks are aligned to a 128 KB boundary, thus bits 11-15 of the stack accesses to thread local data will be the same. Because of it, these threads will compete for a few data cache lines and they will produce a large number of conflicts: threads will not be able to load their memory working set on the data cache.

Our objective is to locate the most used portion of each thread stack into a different data cache. To do this, we have introduced different paddings at the bottom of each thread stack. We have selected the size of the padding to be multiples of 2048 bytes, so that the scrambling function will map the most used portion of each thread stack into different data cache.

Figure 7 shows the scaling for class W of the benchmarks using this modification. All benchmarks scale satisfactorily up to the number of threads that they can feed up (the size of the parallelized dimension of the matrices, shown in Table 2). Comparing these results with the baseline presented in Figure 4, the performance is improved between 10% and 70%, depending on the application.

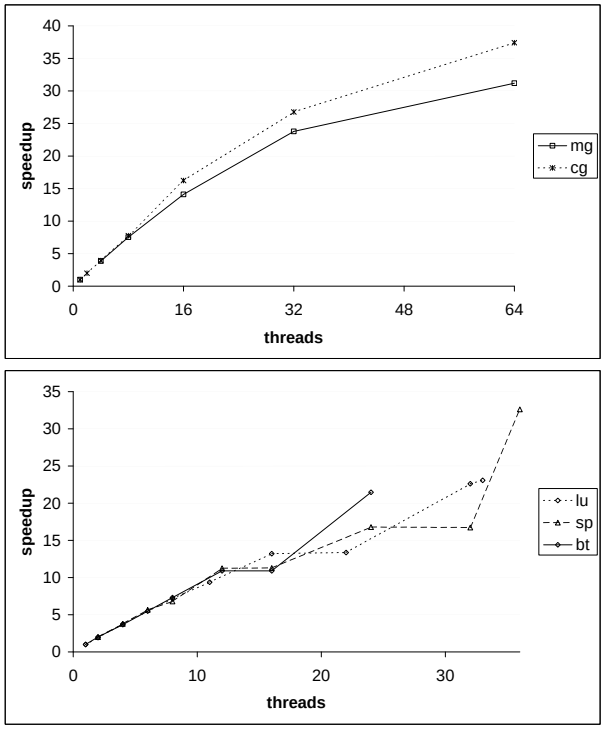


**Figure 7. Performance of the NAS Benchmarks (class W) with the software optimization**

Figure 8 shows that the data accesses to the cache are more evenly distributed, and in particular accesses to stack data located in the slave threads stacks are no longer conflicting. This optimization has a significant larger impact when running with a larger number of threads (64 compared to 4 in the figure). As a result, in contrast with the experiments presented in the previous section, the number of conflict misses is reduced. And thus, the hit rate is kept mostly constant, even when the number of threads is increased, as shown in Figure 9, obtained from the execution of MG.

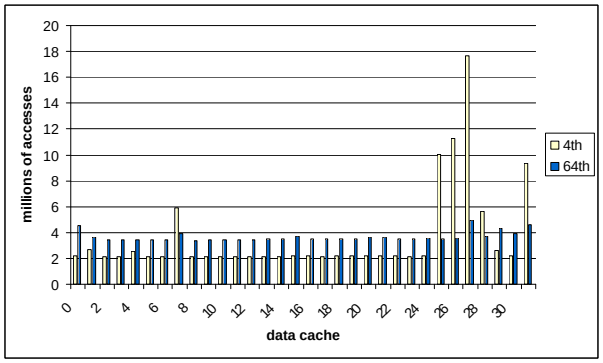


**Figure 8. Comparative of MG (class W) data cache accesses using 4 and 64 threads (software optimization)**

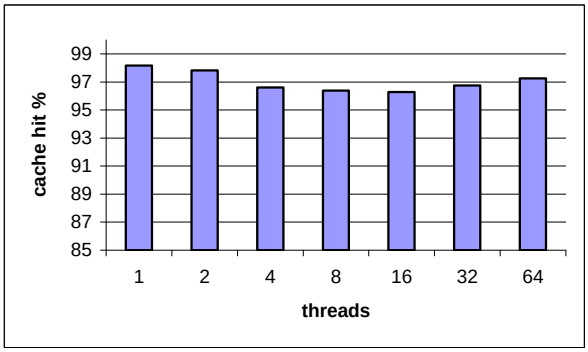


**Figure 9. MG (class W) data cache hits running with different number of threads (software optimization)**

## 6. Optimizing the shared cache behavior

The second optimization that we explore is changing the hardware scrambling function, such that software threads can map stacks to local caches. Compared with the previous optimization, this technique has the following advantages: i) reduces the wasting of memory because padding is no longer necessary — which in the case of Cyclops' limited on-chip memory is even more critical than for other architectures; and ii) threads access local stack data into local caches, thus giving better performance.

To implement this technique, we change the bits used by the scrambling function to decide into which data cache to map a specific data address. Now, bits 17 to 21 are used, giving a total of 128 KB of consecutive memory mapped onto the same data cache. We also add a new register to describe how to map the thread stacks into each data cache. The NthLib runtime was modified to take advantage of these hardware modifications and to map each stack into the thread unit's quad data cache and provide 128KB of stack to the slave threads.

Figure 10 presents the performance results using this technique. As shown, the scaling is similar similar to the software padding solution, however, the cost is much lower.

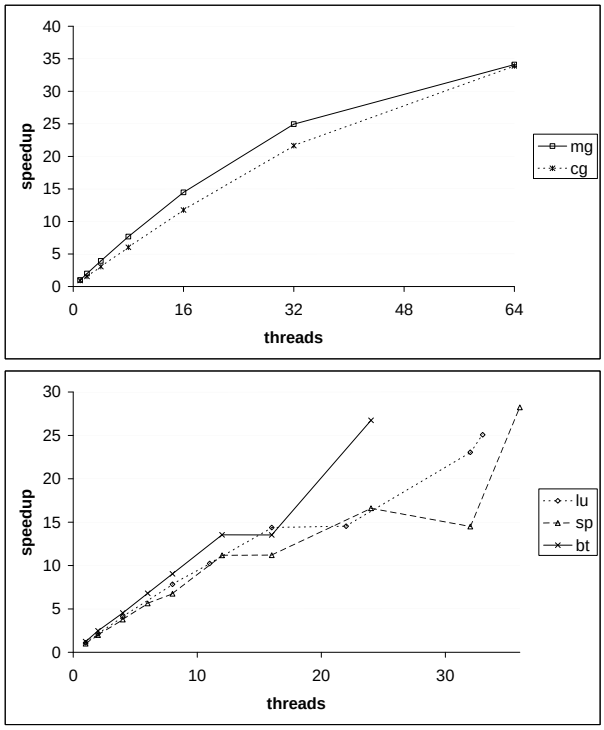


**Figure 10. Performance of the NAS Benchmarks with the hardware optimization**

Results improve the baseline implementation for two reasons. First, because the threads stacks are mapped locally into the thread quad and the local data cache latency is smaller than the remote ones (6 cycles compared to 17 for a hit and 24 to 36 for a miss, as presented in Table 1). And second, because the new scrambling function changes the cache behavior and stacks usually aligned are placed on different data cache lines, avoiding cache conflicts among them. The benefits in this case depend more on the type of the application. For instance it has no effects on CG, while it improves the other NAS benchmarks between 40% and 100%.

In addition, as the padding sizes are reduced, this hardware modification increases significantly the amount of memory available for the application. We estimate that we can save about 8 MB of memory by avoiding the padding.

Figure 11 shows that in MG the data accesses are also evenly distributed across the data caches, as in the software

optimization case. This is also more noticeable when the number of threads is increased, which shows the scalability of the approach.

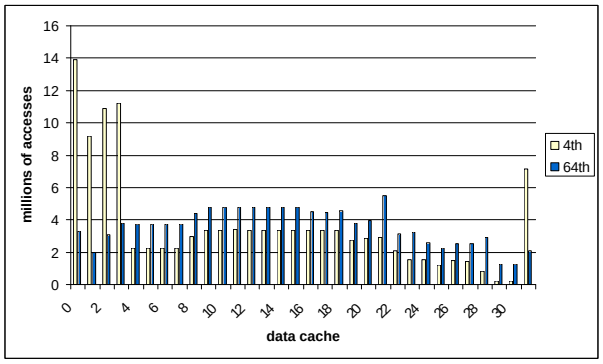


**Figure 11. Comparative of MG data cache accesses using 4 and 64 threads (hardware optimization)**

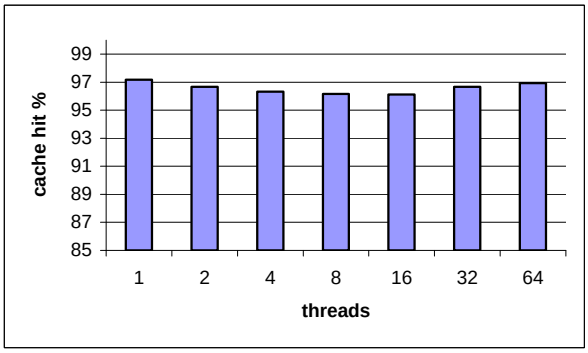


**Figure 12. MG data cache hits running with different number of threads (hardware optimization)**

Figure 12 shows that the hit rate of this solution slightly decreases compared to the hit rate obtained from the software solution. This is because although the new data scrambling function is more beneficial for the local data, it causes an increment of data cache conflict misses in global data areas. The reason for this is that any large data structure of more than 16 KB will not fit in one quad data cache and it will generate many conflict misses. This is reflected in Figure 11, shown as an increment on the number of accesses to data caches 9-20 and the corresponding reduction in data caches 21-31.

## 7. Related Work

Simultaneous multithreading exploits both instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle. It was shown to be a more effective approach to improve resource utilization than superscalar execution. Results presented in [6, 23] support our work by showing that there is not enough instruction-level parallelism in a single thread of execution, therefore it is more efficient to execute multiple threads concurrently.

The Tera MTA [19, 20] is another example of a modern architecture that tolerates latencies through massive parallelism. In the case of Tera, 128 thread contexts share the execution hardware. This contrasts with BG/C, in which each thread has its own execution hardware. Both architectures can tolerate long latencies.

Several studies have been done to determine the cache effects of multithreading. Yamamoto, et al. [25, 24] and Gulati and Bagherzadeh [9] both found that the cache miss rates in simultaneous multithreading processors increased when more threads were used. Thekkath and Eggers [22] found that cache interference between threads varied depending on the benchmark. With our proposals, we achieve a better distribution of data cache accesses onto the available cache memory, thus solving these problems.

As far as we know, this is the first attempt to port efficiently an OpenMP runtime system to a massive parallel multithreaded system on–chip. The porting is based on the experience gained over the years on implementing such an environment on top of other execution environments, including small SMPs and large cc-NUMA. Vendors also provide fine-tuned implementations for their target machines, such as SGI IRIX MP[21] library or the IBM run-time library for AIX. For example, the SGI MP library provides a complete execution environment for each application, supporting thread creation, management, synchronization and NUMA features, such as memory placement. The library is aware of the machine load, trying to adjust the parallelism which is exploiting to the available resources. A number of projects also try to extend the use of OpenMP to clusters with DSM (Distributed Shared Memory) support. The long latencies experienced when accessing remote data and the memory granularity at the page level impose new constraints in these implementations [13, 18].

The NANOS execution environment, the layer used to implement OpenMP on BG/C, focuses on adaptability at different levels: the effective exploitation of nested parallelism and the specification of precedence relations among computations that form pipelines. All these aspects form a set of extensions to OpenMP whose impact is being investigated on BG/C.

A number of studies have been published in which differ-

ent compiler optimizations are evaluated for multithreaded architectures. For example, [12] relaxes and modifies some of the requirements on code scheduling and data access used by current compilers.

As stated before, the BG/C architecture is focused on the execution of a single multithreaded application within each chip. Other architecture proposals such as $\alpha$–Coral [26] provides a mostly hardware managed simultaneous multiprogramming and multithreading environment.

## 8. Conclusions

The work presented in this paper shows how to improve the performance of multithreaded systems by taking advantage of our experience on multiprocessor systems. The paper focuses on one of the critical architectural differences, cache organization, and shows how we can overcome the differences by using simple techniques. In multiprocessor systems each physical thread (processor) has its own cache and thus local stack accesses are cheap, and the overhead occurs when accessing shared data (either through the cache coherence protocol, or explicit cache management).

By contrast, in multithreaded systems a few threads are more closely integrated, and thus competing for the same resources (caches included). The goal is to share as much information as possible and also to avoid conflict misses caused by different threads. This will offer higher performance on many multithreaded processors, including multicore chips, where the L2 cache is shared among all cores. In these architectures, it is desirable to take advantage of the tight coupling, and translate it into better scalability in the execution of parallel applications.

The two different solutions presented to improve the cache management achieve both goals: they increase the amount of information shared in the cache and they avoid conflict misses. Both solutions have similar efficiency impact but different costs.

The first solution consists of introducing different padding sizes into different thread stacks, so that the conflict misses between threads are reduced significantly. In this case, the performance is improved between 10% and 70%, depending on the application.

The second solution consists of changing the hardware function used to map main memory addresses into data caches. This technique can also be used in hyperthreaded and multicore processors. With this solution, the data accessed by the threads is located in its own cache, and accesses are faster. And also, it reduces the number of conflict misses. The benefits in this case depend more on the type of the application, and it has no effects on CG, while it improves the other NAS benchmarks between 40% and 100%.

The results presented in this paper show how with small changes in either hardware or software, we can achieve very good scalability in parallel applications. Our results also show that standard execution environments developed for multiprocessor architectures can be easily adapted to exploit multithreaded processors.

## 9. Future Work

The padding optimization shown in this paper does not consider stack usage; usually, there will be some stack addresses more often accessed than others, as shown in the data cache usage figures. These addresses should be remapped into the thread's quad data cache by setting a more accurate padding.

Cyclops provides a feature that makes possible to have private variables at each quad and ensure that we will have a fast access to that space. It should be necessary to identify non-shared data of each thread and make it private to its data cache; the expected results should be better than the two solutions presented here.

When it is not possible to identify non-shared data to be placed in the threads stacks, we should use a more sophisticated scrambling function than the one presented in this paper. An alternative solution would be to provides an address translation mechanism, which can be used to do the logical to physical translation and map all stack pages into the local data caches. The cost benefits of this solutions may be explored in the future.

Another possibility is to change or adapt the loop scheduling at application level in order to provide the fastest data access into the local quad. When a thread causes a cache miss, a new line is loaded from memory, but the same line may be needed at next iteration. As we use more threads per quad, there exists less space for each thread's lines and may be more difficult to load the complete working-set of many threads into the same cache. The idea is to cause a miss into a line once, and share the line with other threads. In this same way, we will explore multilevel OpenMP, creating groups into the same quads in order to share the same working-set.

Finally, we expect that all measurements and investigations will help us to develop an OpenMP environment capable to get advantage of both shared memory multiprocessor systems and distributed systems based on MPI, where multithreaded processors are used.

## Acknowledgments

This research has been supported by the Ministry of Science and Technology of Spain under contract TIN2004-07739-C02-01 and the European Union under contract IST-2001-33071.

## References


[1] G. Almasi, E. Ayguade, C. Caşcaval, J. G. Castaños, J. Labarta, F. Mart´ınez, X. Martorell, and J. Moreira. Evaluation of openmp for the cyclops multithreaded architecture. In *OpenMP Shared Memory Parallel Programming, International Workshop on OpenMP, Applications and Tools, WOMPAT 2003, Lecture Notes on Computer Science 2716*, pages 64–84, June 2003.

[2] G. Almasi, C. Caşcaval, J. G. Castaños, M. Denneau, D. Lieber, J. E. Moreira, and J. Henry S. Warren. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. In *MEDEA Workshop on On-Chip Multiprocessor: Processor Architecture and Memory Hierarchy related Issues*, September 2002.

[3] G. Almasi, C. Caşcaval, J. G. Castaños, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren, Jr. Performance evaluation of the Cyclops architecture family. Technical Report RC22243, IBM T. J. Watson Research Center, November 2001.

[4] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The nas parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[5] J. Clabes, J. Friedrich, M. Sweet, J. D. a nd Sam Chu, D. Plass, J. Dawson, P. Muench, L. P. an d Michael Floyd, B. Sinharoy, M. Lee, M. Goulet, J. W. agoner, N. Schwartz, S. Runyon, G. Gorman, P. R. a nd Ronald Kalla, J. McGill, and S. Dodson. Design and implementation of the power5 microprocessor. In *DAC 04: Proceedings of the 41st annual conference on Design automa tion*, pages 670–672. ACM Press, 2004.

[6] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, pages 12–18, September/October 1997.

[7] F. A. et al. Blue gene: A vision for protein science using a petafbp supercomputer. *IBM Systems Journal*, 40(2):310–328, 2001.

[8] M. Gonzalez, E. Ayguade, X. Martorell, J. Labarta, N. Navarro, and J. Oliver. NanosCompiler: Supporting flexible multilevel parallelism in OpenMP. *Concurrency: Practice and Experience*, 12(9), August 2000.

[9] M. Gulati and N. Bagherzadeh. Performance study of multithreaded superscalar microprocessors. In *2nd International Symposium on High Performance Computer Architecture*, pages 291–301, February 1996.

[10] Intel. Intel's t's deliver new platform enhancements beyond gigahertz, Dec. 2004, Jan. 2005. http://www.intel.com/update/departments/initech/it12041.pdf.

[11] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of the NAS parallel benchmarks and its performance. Technical Report Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[12] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *International Symposium on Microarchitecture*, pages 114–124, 1997.

[13] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on network of workstations. In *Proc. of Supercomputing'98*, 1998.

[14] X. Martorell, E. Ayguade, J. Navarro, J. Corbalan, M. Gonzalez, and J. Labarta. Thread fork/join techniques for multi–level parallelism exploitation in NUMA multiprocessors. In *Proceedings of the 13th Int. Conference on Supercomputing ICS'99*, June 1999.

[15] X. Martorell, J. Labarta, J. Navarro, and E. Ayguade. A library implementation of the nano-threads programming model. In *Proceedings of Euro-Par'96*, August 1996.

[16] O. Organization. OpenMP Fortran application interface, v. 2.0. *www.openmp.org*, June 2000.

[17] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors. In *1989 International Conference on Parallel Processing*, volume II, pages 39–48, St. Charles, Ill., 1989.

[18] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP compiler for an smp cluster, 1999.

[19] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchel, J. Feo, and B. Koblenz. Multiprocessor performance on the Tera MTA. In *Proceedings Supercomputing '98*, Orlando, Florida, Nov. 7-13 1998.

[20] A. Snavely, G. Johnson, and J. Genetti. Data intensive volume visualization on the Tera MTA and Cray T3E. In *Proceedings of the High Performance Computing Symposium - HPC '99*, pages 59–64, 1999.

[21] S. G. C. Systems. Origin2000 and Onyx2 performance tuning and optimization guide. Technical Report Doc. num. 007-3430-002, 1998.

[22] R. Thekkath and S. J. Eggers. The effectiveness of multiple hardware contexts. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 328–337, October 1994.

[23] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[24] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 95)*, pages 49–58, June 1995.

[25] W. Yamamoto, M. Serrano, A. Talcott, R. Wood, and M. Nemirovsky. Performance estimation of multistreamed, superscalar processors. In *27th Hawaii International Conference on System Sciences*, pages I:195–204, January 1994.

[26] M. Yankelevsky and C. D. Polychronopoulos. $\alpha$–Coral: A multigrain, multithreading processor architecture. In *Proceedings of International Conference on Supercomputing'01*, 2001.