Evaluation of a Multithreaded Architecture for Cellular Computing

Călin Caşcaval José G. Castaños Luis Ceze Monty Denneau Manish Gupta
Derek Lieber José E. Moreira Karin Strauss
Henry S. Warren, Jr.
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598-0218
{cascaval,castanos,lceze,denneau,mgupta,lieber,jmoreira,kstrauss,hankw}@us.ibm.com

Abstract

Cyclops is a new architecture for high performance parallel computers being developed at the IBM T. J. Watson Research Center. The basic cell of this architecture is a single-chip SMP system with multiple threads of execution, embedded memory, and integrated communications hardware. Massive intra-chip parallelism is used to tolerate memory and functional unit latencies. Large systems with thousands of chips can be built by replicating this basic cell in a regular pattern. In this paper we describe the Cyclops architecture and evaluate two of its new hardware features: memory hierarchy with flexible cache organization and fast barrier hardware. Our experiments with the STREAM benchmark show that a particular design can achieve a sustainable memory bandwidth of 40 GB/s, equal to the peak hardware bandwidth and similar to the performance of a 128-processor SGI Origin 3800. For small vectors, we have observed in-cache bandwidth above 80 GB/s. We also show that the fast barrier hardware can improve the performance of the Splash-2 FFT kernel by up to 10%. Our results demonstrate that the Cyclops approach of integrating a large number of simple processing elements and multiple memory banks in the same chip is an effective alternative for designing high performance systems.

1. Introduction

With the continuing trends in processor and memory evolution, overall system performance is more and more limited by the performance of the memory subsystem. The traditional solution to this problem, usually referred to as the von Neumann bottleneck, is to design ever larger and deeper memory hierarchies. This results in complex designs where most of the real estate is devoted to storing and moving data, instead of actual data processing. Even within the processing units themselves, significant resources are used to reduce the impact of memory hierarchy and functional unit latencies. Approaches like out-of-order execution, multiple instruction issue, and deep pipelines have been successful in the past. However, the average number of machine cycles to execute an instruction has not improved significantly in the past few years. This indicates that we are reaching a point of diminishing returns as measured by the increase in performance obtained from additional transistors and power.

This paper describes the Cyclops architecture, a new approach to effectively use the transistor and power budget of a piece of silicon. The primary reasoning behind Cyclops is that computer architecture and organization has become too complicated and it is time to simplify. The Cyclops architecture is founded on three main principles: (i) the integration of processing logic and memory in the same piece of silicon; (ii) the use of massive intra-chip parallelism to tolerate latencies; and (iii) a cellular approach to building large systems.

The integration of memory and logic in the same chip results in a simpler memory hierarchy with higher bandwidth and lower latencies. Although this alleviates the memory latency problem, access to data still takes multiple machine cycles. The Cyclops solution is to populate the chip with a large number of thread units. Each thread unit behaves like a simple, single-issue, in-order processor. Expensive resources, like floating-point units and caches, are shared by groups of threads to ensure high utilization. The thread units are independent. If a thread stalls on a memory reference or on the result of an operation, other threads can continue to make progress. The performance of each individual thread is not particularly high, but the aggregate chip performance is much better than a conventional design with an equivalent number of transistors. Large, scalable systems can be built with a cellular approach using the Cyclops chip as a building block [2]. The chip is viewed as a cell that can be replicated as many times as necessary, with the cells interconnected in a regular pattern through communication links provided in each chip.

Cyclops is a research project and the architecture is still evolving. In this paper we describe one particular configuration that we have evaluated. We shall focus on two features that we have demonstrated to improve performance substantially: (i) the memory hierarchy, particularly the novel cache organization, and (ii) the mechanism for fast synchronization. The current design point for Cyclops calls for 128 thread execution units. These thread units are organized in groups of four, called *quads*. Each quad includes one floating-point unit, shared by the four thread units in the quad, and a 16 KB cache unit. Main memory is organized in 16 independent banks of 512 KB each, for a total of 8 MB, and it is shared by all threads in the chip. We want to emphasize that these numbers represent just one of many configurations possible. The total numbers of processing units and memory modules are mainly driven by silicon area, so as to maximize floating-point performance. The degrees of sharing for floating-point and cache units were selected based on instruction mixes observed in current systems [8]. We continue to study these trade-offs in parallel with the hardware design [3], to obtain the best performance.

Designing an effective cache organization for a large SMP like the Cyclops chip is always a challenge. Conventional approaches to cache coherence, such as snooping and directories, have scalability and cost problems (as measured in silicon area). Our approach consists of a software-controlled, non-uniform access, shared cache system. The threads of execution share the multiple cache units in the chip, but each thread is more tightly coupled to one particular unit. Software controls the placement of data in the cache, allowing high-affinity data (*e.g.*, thread stack) to be placed in the cache unit closer to a particular thread. We evaluate the efficacy and efficiency of the Cyclops memory hierarchy through measurements using the STREAM benchmark.

Likewise, designing effective synchronization mechanisms for large SMPs is a challenge. Our initial experiments with memory-based synchronization demonstrated that software barriers for a large number of threads could be very slow, with a degrading performance on important benchmarks [4]. These measurements motivated the development of dedicated hardware barriers in the chip. We evaluate the performance improvements resulting from these fast barriers through measurements using the FFT kernel from the Splash-2 benchmark suite.

The rest of this paper is organized as follows. Section 2 presents a detailed description of the Cyclops architecture. Section 3 describes our simulation environment and presents performance results for the STREAM [11] and Splash-2 FFT [22] benchmarks. Section 4 discusses related work. Finally, Section 5 presents our conclusions and discusses plans for future work.

2. The Cyclops architecture

The architecture of the Cyclops chip is a hierarchical design, shown in Figure 1, in which threads share resources at different levels of the hierarchy. The main idea behind the design is to integrate in one chip as many concurrent threads of execution as possible. Instead of hiding latencies, through out-of-order or speculative execution, Cyclops tolerates latencies through massive parallelism. With this design each thread unit is simpler and expensive resources, such as floating-point units and caches, are shared between different threads.

The architecture itself does not specify the number of components at each level of the hierarchy. In this section we describe a possible implementation of the Cyclops architecture with components determined by silicon area constraints and most common instruction type percentages. We expect these numbers to change as manufacturing technology improves. Also, the balance between different resources might change as a consequence of particular target applications and as our understanding of the different trade-offs improves.

In this paper we consider a 32-bit architecture for Cyclops. The proprietary instruction set architecture (ISA) consists of about 60 instruction types, and follows a 3-operand, load/store RISC design. The decision of using a new simplified ISA bears on the goal of a simpler design. For designing the Cyclops ISA we selected the most widely used instructions in the PowerPC architecture. Instructions were added to enable multithreaded functionality, such as atomic memory operations and synchronization instructions.

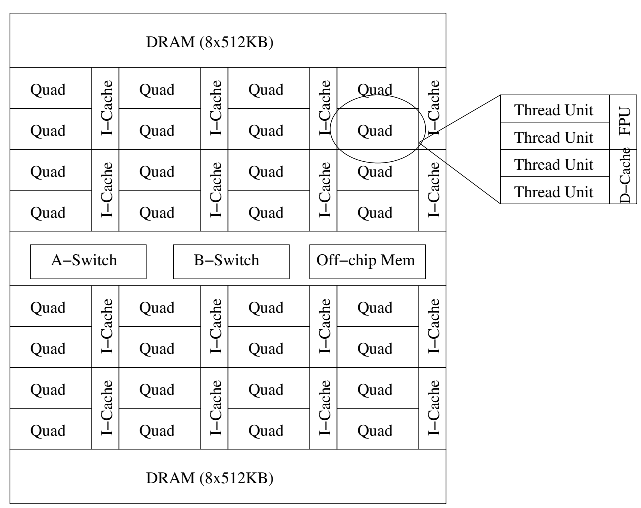


Figure 1. Cyclops chip block diagram.

We evaluate a design with the following characteristics: 128 thread units, each unit containing a register file (64 32-

bit single precision registers, that can be paired for double precision operations), a program counter, a fixed point ALU, and a sequencer. The thread units are very simple computing processors that execute instructions in program order (completion may be out-of-order). Most instructions execute in one cycle. Each thread can issue an instruction at every cycle, if resources are available and there are no dependences with previous instructions. If two threads try to issue instructions using the same shared resource, one thread is selected as winner in a round-robin scheme to prevent starvation. If an instruction cannot be issued, the thread unit stalls until all resources become available, either through the completion of previously issued instructions, or through the release of resources held by other threads.

Groups of four thread units form a *quad*. The threads in a quad share a floating-point unit (FPU) and a data cache. Only the threads within a quad can use that quad's FPU, while any thread can access data stored in any of the data caches. The memory hierarchy has non-uniform access latencies. Thus, threads have faster access to their local data cache than to a remote cache. The floating-point unit consists of three functional units: an adder, a multiplier, and a divide and square root unit. Threads can dispatch a floating point addition and a floating point multiplication at every cycle. The FPU can complete a floating point multiply-add (FMA) every cycle. With a clock cycle of 500 MHz, in $.18\mu m$ CMOS technology, it achieves a peak performance of 1 GFlops per FPU, for a total chip performance of 32 GFlops.

2.1. Memory hierarchy

A large part of the silicon area in the Cyclops design is dedicated to memory. This memory is distributed on two hierarchical levels, main memory and caches for data and instructions, further described in this section.

Our design uses 16 banks of on-chip memory shared between all thread units. Each bank is 512 KB of embedded DRAM, for a total of 8 MB. The banks provide a contiguous address space to the threads. Accesses to the memory banks go through a memory switch, shown in Figure 2, thus the latency to any bank is uniform. Addresses are interleaved to provide higher memory bandwidth. The unit of access is a 32-byte block, and threads accessing two consecutive blocks in the same bank will see a lower latency in burst transfer mode. The physical memory address is 24 bits, giving a maximum addressable memory of 16 MB. The peak bandwidth of the embedded memory is 42 GB/s (64 bytes every 12 cycles, 16 memory banks).

Each of the 16 KB data cache (one per quad) has 64-byte lines and a variable associativity, up to 8-way. The data caches are shared among quads. That is, a thread in a quad can access data in other quads' caches, with lower latency than going to memory. All remote caches have the same access latency, higher than the local cache access latency, since they are accessed through a common switch, as shown in Figure 2. The peak bandwidth out of the caches is 128 GB/s (8 bytes per cycle, 32 caches).

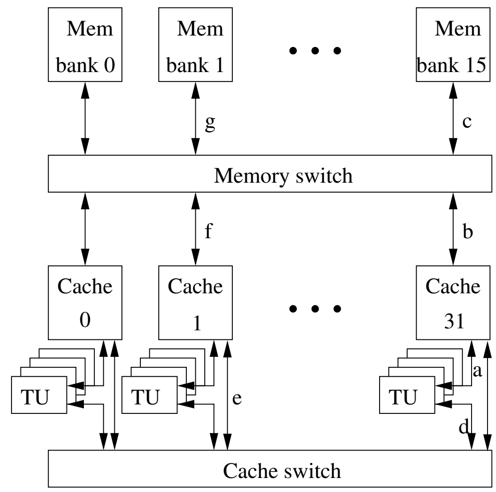


Figure 2. Cyclops memory hierarchy.

In the same figure we have marked the paths that a data item will traverse in different scenarios. For example, if a thread unit (TU) in quad 31 accesses data present in its local cache, the data will come through the path marked *a*. For a local cache miss, the request will go through path *a* to the local cache, will propagate to memory following the path *bg*, and come to the thread following *gba*. A remote cache request for cache 1 will go through the path *de*. A remote cache hit into cache 1 will come through *ed*, while a remote cache miss will follow the path *fcfed*.

The hardware does not implement any cache coherence mechanism to deal with multiple copies of a memory line in different caches. However, the architecture supports an entire spectrum of access schemes, from no coherence at all to coherent caches shared at different levels. The levels range from sharing across the entire chip down to sharing within each quad. Any memory location can be placed in any cache under software control. The same physical address can be mapped to different caches depending on the logical address. Since a physical address is limited to 24 bits, we use the upper 8 bits of the 32-bit effective address to encode cache placement information. The encoding, presented in Table 1, allows a thread to specify in which cache the data accessed is mapped. We call this *interest group* encoding, and it works as follows: the q bits in the first column in Table 1 specify a number that defines one set of caches, shown in the second column of the table. If the set contains more than one member, the hardware will select one of the caches in the set, utilizing a scrambling function

Table 1. Interest group encoding

Encoding	Selected Caches	Comments
0b000 0 0 0 0 0		thread's own
0b00$q_4q_3q_2q_1q_0$1	{0}, {1}, ... {31}	exactly one
0b00$q_3q_2q_1q_0$1 0	{0,1}, {2,3}, ... {30,31}	one of a pair
0b00$q_2q_1q_0$1 0 0	{0,1,2,3}, ... {28,29,30,31}	one of four
0b00$q_1q_0$1 0 0 0	{0, ... 8}, ... {24, ... 31}	one of eight
0b00$q_0$1 0 0 0 0	{0,1, ... 15}, {16,17, ... 31}	one of sixteen
0b001 0 0 0 0 0	{0,1, ... 31}	one of all

so that all the caches are uniformly utilized. The function is completely deterministic and relies only on the address such that references to the same effective address get mapped to the same cache.

If all references use the interest group 0b00100000, the caches behave as a single 512 KB coherent unit shared across the chip. This is the default used by our system software. Each piece of data is mapped only to a single cache. A drawback of this scheme is that given a uniform distribution of accesses, only one out of 32 accesses will be in the local cache. The other non-zero interest groups partition the caches in different ways, from two units of 256 KB each to 32 units of 16 KB each. In each of those cases, one effective address identifies one, and only one, cache location. Hence, the cache coherence problem does not arise when using those interest groups. For example, an interest group of 0b00010001, indicates that the data should be cached in cache number 8, while an interest group of 0b00010010 indicates either cache 7 or cache 8. When using the interest group zero (0b00000000), each thread accessing that data will bring it into its own cache, resulting in a potentially non-coherent system. The cache selected depends on the accessing thread. This means that the same memory location can be mapped to multiple caches. Without coherence support in hardware, it is up to user level code to guarantee that this potential replication is done correctly.

An important use of this flexible cache organization is to exploit locality and shared read-only data. For example, data frequently accessed by a thread, such as stack data or constants, can be cached in the local cache by using the appropriate interest group. The same constant could be cached in different caches by threads in separate quads by using interest group zero and a physical address that points to the same memory location.

A data cache can also be partitioned with a granularity of 2 KB (one set) so that a portion of it can be used as an addressable fast memory, for streaming data or temporary work areas. The threads sharing a data cache have to agree on a certain organization for a particular application. This feature can potentially result in higher performance for applications that are coded to use this fast memory directly, instead of relying on the dynamic, and often hard to control, cache behavior.

Instruction caches are 32 KB, 8-way set-associative with 64-byte line size. One instruction cache is shared by 2 quads. Unlike the data caches, the instruction caches are private to the quad pair. In addition, to improve instruction fetching, each thread has a Prefetch Instruction Buffer (PIB) that can hold up to 16 instructions.

Some applications require more memory than is available on the Cyclops chip. To support these applications, the design includes optional off-chip memory ranging in size from 128 MB to 2 GB. In the current design the off-chip memory is not directly addressable. Blocks of data, 1 KB in size, are transferred between the external memory and the embedded memory much like disk operations.

2.2. Communication interface

The Cyclops chip provides six input and six output links. These links allow a chip to be directly connected in a three dimensional topology (mesh or torus). The links are 16-bit wide and operate at 500 MHz, giving a maximum I/O bandwidth of 12 GB/s. In addition, a seventh link can be used to connect to a host computer. These links can be used to build larger systems without additional hardware. However, this is not focus of this paper.

2.3. Synchronization primitives

An additional feature of the Cyclops chip is the fast interthread hardware barrier, provided through a special purpose register (SPR). It is actually implemented as a wired OR for all the threads on the chip. Each thread writes its SPR independently, and it reads the ORed value of all the threads' SPRs. The register has 8 bits and we use 2 bits per barrier, thus providing 4 distinct barriers. One of the bits holds the state of the *current* barrier cycle while the other holds the state of the *next* barrier cycle. In one cycle, all threads participating in the barrier initially set their *current* barrier cycle bit to 1. The threads not participating in the barrier leave both bits set to 0. When a thread is ready to enter a barrier, it atomically writes a 0 to the *current* bit, thereby removing its contribution to the current barrier cycle, and a 1 to the *next* bit, thereby initializing the next barrier cycle. Each thread then reads back its register and spins, waiting for the value of the *current* bit to become 0. This will happen when all threads have written a 0 to that bit position in their special purpose registers. Roles are interchanged after each use of the barrier. Because each thread spin-waits on its own register, there is no contention for other chip resources and all threads run at full speed. Performance data for the fast barrier operations are presented in Section 3.3.

3. Experimental results

In this section we describe our simulation environment for executing Cyclops code and evaluating the performance of the Cyclops chip. We also report experiments with two different benchmarks: the STREAM benchmark is used to assess the sustained memory bandwidth and overall performance of the Cyclops chip when executing vector kernels; the FFT kernel from the Splash-2 benchmark suite measures the performance impact of Cyclops' fast barrier mechanism.

3.1. The Cyclops simulation environment

Cyclops executables (kernel, libraries, applications) are currently being generated with a cross-compiler based on the GNU toolkit, re-targeted for the Cyclops instruction set architecture. This cross-compiler supports C, C++, and FORTRAN 77. An architecturally accurate simulator executes instructions from the Cyclops instruction set, modeling resource contention between instructions, and thus estimating the number of cycles each instruction executes. The simulator is parametrized such that different architectural features can be specified when the program runs.

The performance parameters for the simulated architecture are shown in Table 2. The upper section shows instruction latencies, in cycles. The execution column is the number of cycles the functional unit is busy executing the instruction. The additional cycles after which the result becomes available are shown in the latency column. The lower section summarizes the hardware parameters used in simulations.

Table 2. Simulation Parameters.

Instruction type	Execution	Latency
Branches	2	0
Integer multiplication	1	5
Integer divide	33	0
Floating point add, mult. and conv.	1	5
Floating point divide (double prec.)	30	0
Floating point square root (double prec.)	56	0
Floating point multiply-and-add	1	9
Memory operation (local cache hit)	1	6
Memory operation (local cache miss)	1	24
Memory operation (remote cache hit)	1	17
Memory operation (remote cache miss)	1	36
All other operations	1	0

Component	# of units	Params/unit
Threads	128	single issue, in-order, 500 MHz
FPUs	32	1 add, 1 multiply, 1 divide/square root
D-cache	32	16 KB, up to 8-way assoc., 64-byte lines
I-cache	16	32 KB, 8-way assoc., 32-byte lines
Memory	16	512 KB

Each chip runs a resident system kernel, which executes with supervisor privileges. The kernel supports single user, single program, multithreaded applications within each chip. These applications execute in user mode. The kernel exposes a single-address space shared by all threads. Due to the small address space and large number of hardware threads available, no resource virtualization is performed in software: virtual addresses map directly to physical addresses (no paging) and software threads map directly to hardware threads. The kernel does not support preemption (except in debugging mode), scheduling or prioritization. Every software thread is preallocated with a fixed size stack per thread (selected at boot time), resulting in fast thread creation and reuse.

Before going into the detailed analysis of the memory bandwidth and hardware barriers, we present parallel speedups obtained on a subset of the Splash-2 benchmarks (see Figure 3). While not optimized specifically for Cyclops, most of these benchmarks reach appropriate levels of scalability, comparable to those reported in [22].

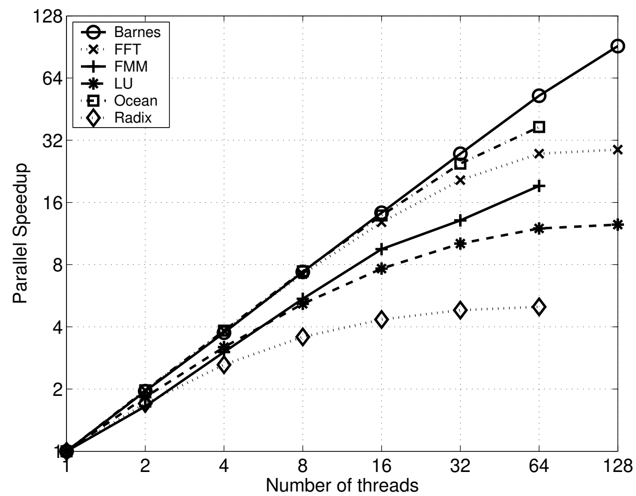
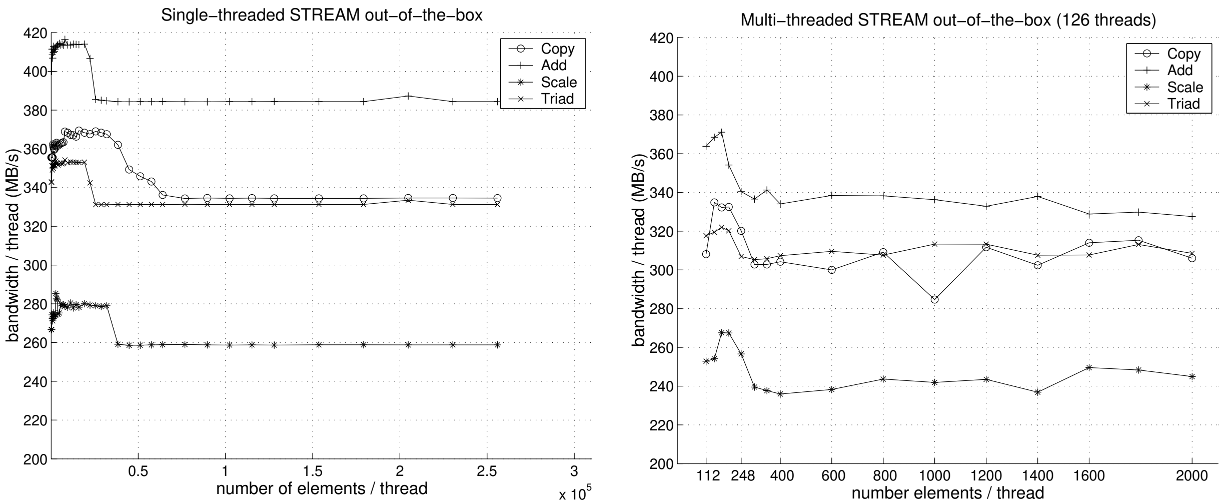


Figure 3. SPLASH2 parallel speedups.

3.2. Experiments with the STREAM Benchmark

The STREAM benchmark [11] is a simple synthetic benchmark program that measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernels. It is intended to characterize the behavior of a system for applications that are limited in performance by the memory bandwidth of the system, rather than by the computational performance of the CPU. The STREAM benchmark consists of four vector kernels: Copy ($c_i = b_i$), Add ($c_i = a_i + b_i$), Scale ($b_i = s \times c_i$), and Triad ($a_i = b_i + s \times c_i$), which operate on vectors a, b, and c of double-precision floating point elements. To investigate the behavior of the memory subsystem, we run the benchmark for different values of n, the vector length. We report the measured bandwidth following the STREAM convention:



(a) (b)

Figure 4. Single- and multi-threaded (126 threads) STREAM out-of-the-box performance

Copy and Scale move $2n$ 64-bit double words per run, whereas Add and Triad move $3n$ double words per run. We perform ten runs per experiment and report the highest performance result. STREAM specifies that each vector size should be at least four times the size of the last level cache or 1 million elements, whatever is larger. Unfortunately, since the main memory of a Cyclops chip is 8 MB we cannot satisfy the million-element requirement. The largest vector size we use is 252,000 elements, or approximately 2 MB. This is four times the size of the combined data caches.

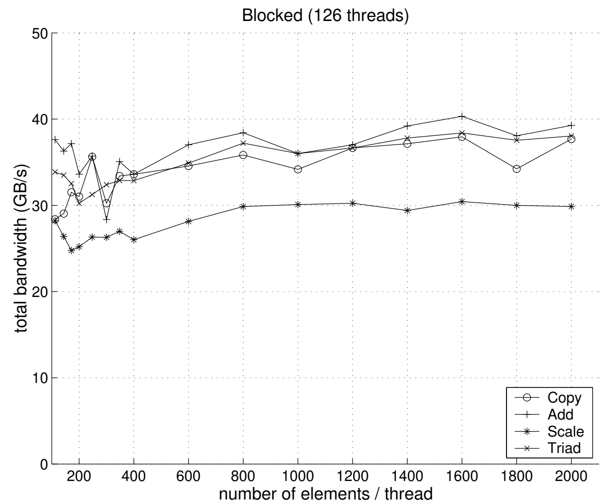
3.2.1. STREAM out-of-the-box. We started by running the STREAM benchmark directly out-of-the-box as a single-threaded computation. Results as a function of vector size are reported in Figure 4(a). This figure shows the transition between in-cache and out-of-cache modes of operation, as the vector size increases. The transition for Add and Triad happens for smaller vector sizes since those operations use three vectors. Copy and Scale use two vectors each. We also run 126 copies of the benchmark as a multithreaded computation, where each thread performing its own benchmark independently. Although the total number of threads in the chip is 128, only 126 could be used for the benchmark because two of them are reserved for the system. Results, in terms of average memory bandwidth sustained by each thread, as a function of vector size *per thread* are reported in Figure 4(b). Although the curves are not as smooth, we can still observe a transition between in-cache and out-of-cache modes at 200-300 elements per thread.

The thread in the single thread run achieves a higher performance compared to each individual thread from the multithreaded run, as can be seen in Figure 4. This happens because in the multithreaded run, the threads are contending for shared bandwidth. The aggregate bandwidth achieved by the multithreaded version corresponds to the sum of the bandwidths observed for all 126 threads. For large vectors, that bandwidth is from 112 (for Add) to 120 (for Triad) times larger than for the single-threaded case.

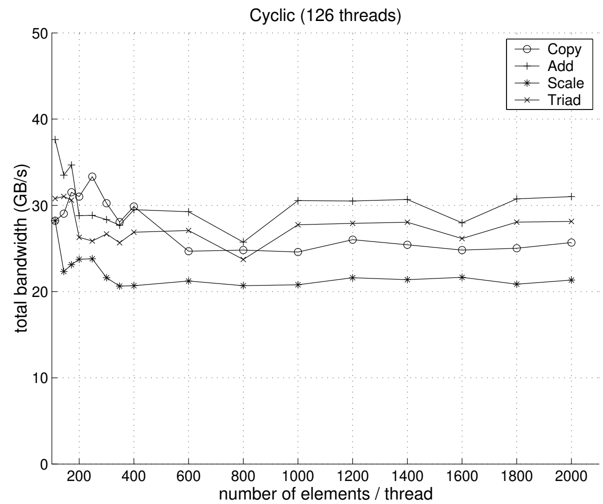
3.2.2. Multithreaded STREAM. We then proceeded to evaluate the parallel execution of a single STREAM benchmark. The code was parallelized by hand using *pthreads*. We perform experiments to measure the impact of loop partitioning, use of local caches, thread allocation policies, and code optimization. We also compare multithreaded STREAM execution in Cyclops to execution on a commercial large-scale shared memory system.

Loop partitioning: Using the Cyclops cache system as a single 512 KB cache, we studied both block and cyclic partitioning of loop iterations among the threads. We note that block and cyclic partitioning of iterations correspond to block and cyclic access patterns for each thread. To avoid all threads hitting the same region of memory at the same time, in the cyclic mode threads were combined in groups of eight, and each group started execution from a different region of the iteration space. By combining threads in groups of eight we allow for reuse of cache lines, which contain eight double-precision elements.

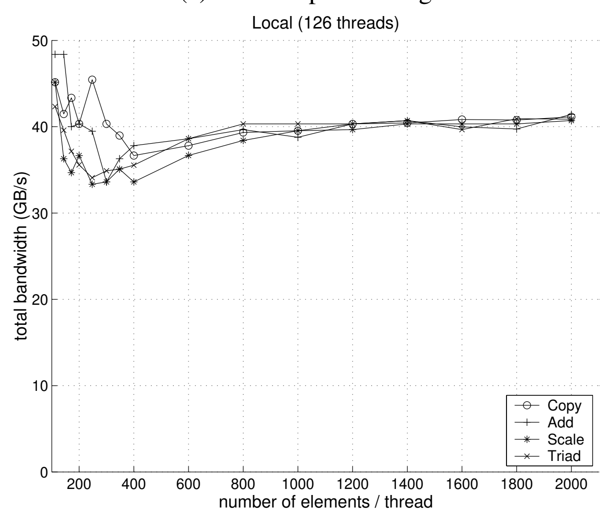
Results for the blocked and cyclic partitioning are shown in Figure 5(a) and (b). For the same vector size, the performance achieved in blocked mode is better than that achieved in cyclic mode. In blocked mode, each thread loads one cache line from main memory to cache and uses the other seven elements later. In this case, each cache line is used by only one thread. In cyclic mode, each cache line is accessed



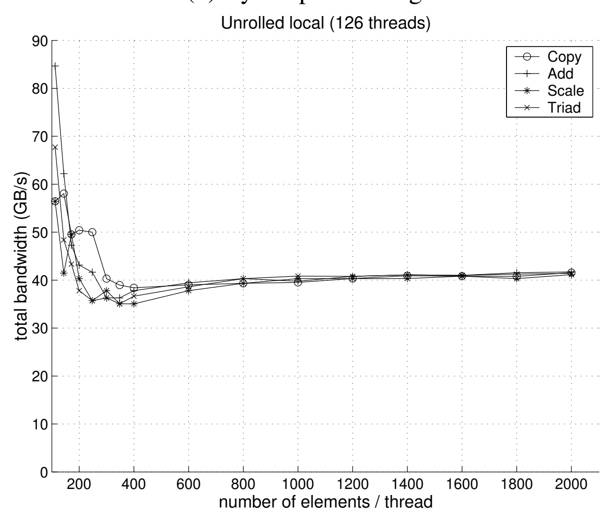
(a) Blocked partitioning



(b) Cyclic partitioning



(c) Blocked partitioning with local caches



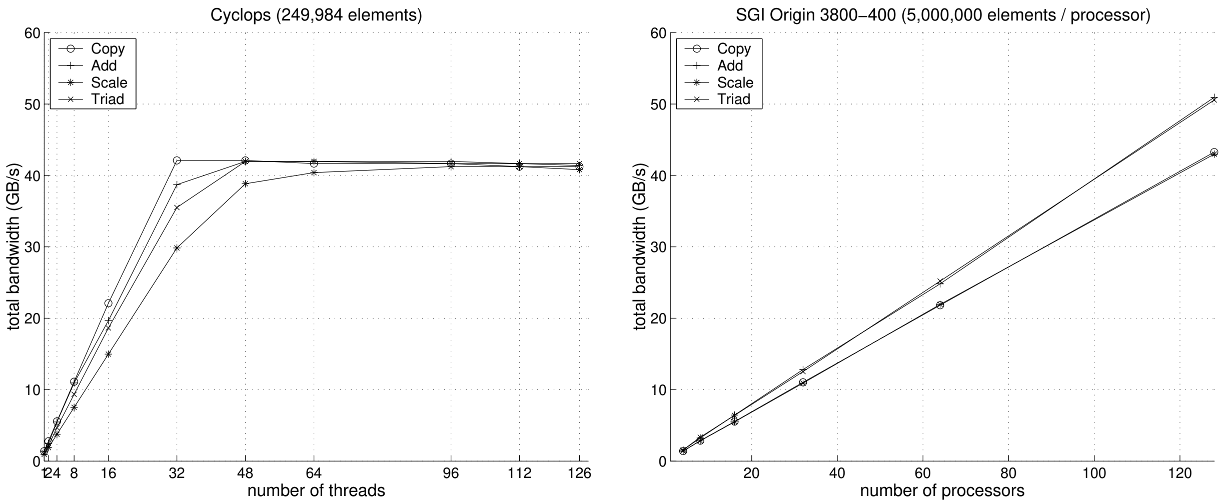
(d) Unrolled loops with block partitioning and local caches

Figure 5. Memory bandwidth vs. vector size for 126 threads in different modes.

by the eight threads in a group. The threads access the same cache line at approximately the same time, while the cache line is still being retrieved from main memory. Because of that, each thread will have to wait longer to get the data it needs. Therefore, the average waiting period for the data is greater in cyclic mode.

Taking advantage of local caches: In the measurements reported above, data accessed by one thread is always spread over all quads. As a result, most of the accesses (on the order of $\frac{31}{32}$) are remote cache references. The access time to a local cache is three times faster than the access time to a remote cache (6 cycles *vs.* 17 cycles). To improve performance, we use the interest group feature of Cyclops to force all vector elements accessed by a thread to map into its local cache. False sharing was avoided by making the block sizes multiples of cache lines and aligning the blocks to cache line boundaries. For the same vector size, performance with vector blocks mapped to the local cache are better than with distributed caches (Figure 5(c)). For small vectors we observe improvements of up to 60% in total bandwidth. Although the improvements are smaller for large vectors, as performance is limited by the main memory bandwidth, we still see a 30% improvement for Scale.

Thread allocation policies: By default threads are sequentially allocated. That is, threads 0 through 3 are allocated in quad 0, threads 4 through 7 are allocated in quad 1 and so on. We can also use a balanced thread allocation policy. With that policy, threads are allocated cyclically on the quads: threads 0, 32, 64, and 96 in quad 0, threads 1, 33, 65, and 97 in quad 1, and so on. We measured the im-



(a) (b)

Figure 6. Comparing performance of (a) Cyclops with unrolled loops, local caches, balanced thread allocation policy and block partitioning (vector size is 249,984 elements) *vs.* that of (b) SGI Origin 3800-400 (vector size is 5,000,000 elements/processor).

pact of the balanced policy when used with the local cache mode. The balanced policy improves results for local access mode when less than all threads are used. In general, with the balanced policy there is less pressure from the threads to each cache. In the case of Copy, the bandwidth with the balanced policy can be up to 20% higher than with the unbalanced policy. When all threads are being used, the allocation makes no difference because all quads have four threads active.

Code optimization (unrolling): When the original STREAM benchmark code is compiled, the resulting instruction sequence inside the loops is: load the operands, execute the operation and store the result. Since there are dependences between the instructions, a thread has to stall until its load/store operations are completed. Issuing other independent instructions while the load or store instructions execute is desirable. That can be achieved by unrolling the code. We perform a four-way unrolling of the code by hand, because the GNU compiler does not handle it satisfactorily.

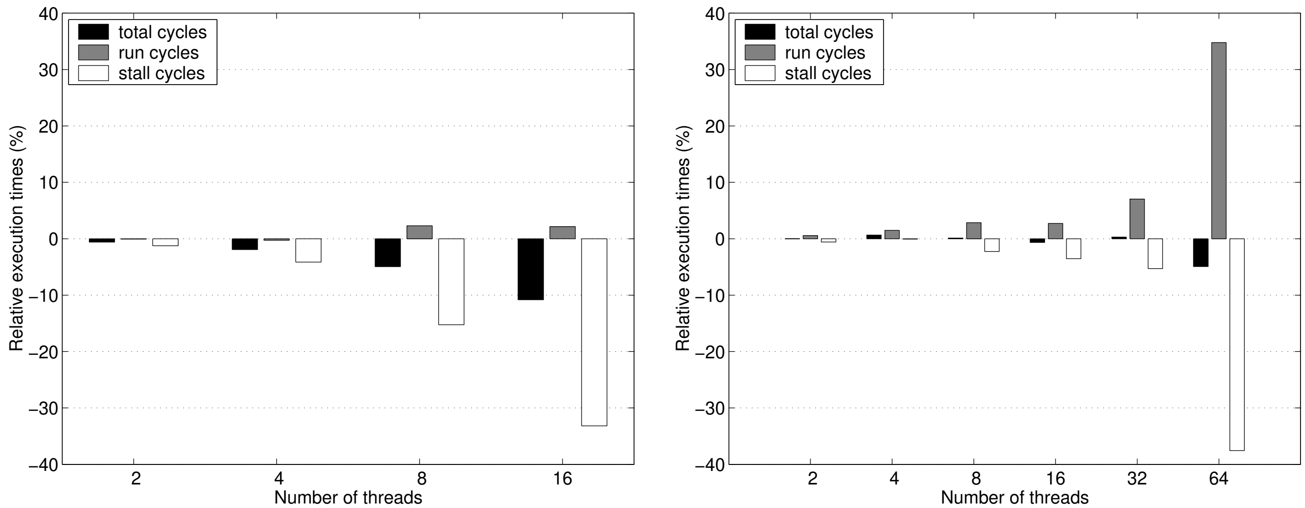
Figure 5(d) shows that loop unrolling, combined with blocked partitioning and the use of local caches, improves the overall performance for small vectors, since other useful instructions are being issued while the load/store operations complete. In the case of long vectors, overall performance is constrained by main memory bandwidth, and unrolling does not make a difference.

Comparing with a commercial machine: The best STREAM results for Cyclops, for different numbers of threads, are compared with the published results for the SGI Origin 3800/400 in Figure 6. We used a large fixed vector length for Cyclops, which forces out-of-cache operation. The published results for the SGI Origin used vector lengths that were a function of the number of processors. We note that the vector sizes for the Origin are much larger, and that Cyclops does not have enough memory to run that size. Nevertheless, it is remarkable that a single Cyclops chip can achieve sustainable memory bandwidth similar to that of a top-of-the-line commercial machine.

3.2.3. Conclusions from the STREAM benchmark tests. It is clear from the measurements that the best performance in STREAM for Cyclops is achieved with a combination of block partitioning of the data, use of local caches, and code optimization. The balanced thread allocation policy also improves performance, but only in combination with a local cache policy and only when not all threads are being used. Unrolling the code helps to improve performance because it increases the number of useful instructions between load/store dependences and thus reduces the number of stalls by a thread. However, for large problem sizes the memory bandwidth becomes the real limiting factor and unrolling loses its impact.

3.3. Hardware barriers validation

We compared the performance of the hardware barrier feature of Cyclops against a software implementation, using the FFT kernel from the Splash-2 [22] benchmark suite. The



(a) 256-point FFT

(b) 64K-point FFT

Figure 7. Hardware *vs.* software barriers in SPLASH2 FFT.

software barriers are a tree based scheme: on entering a barrier a thread first notifies its parent and then spins on a memory location that is written by the thread's parent when all threads have completed the barrier.

Figure 7 shows the performance improvement of the hardware barriers over the software implemented barriers for two input data sizes: a 256-point and a 64K-point FFT. The benchmark requires the number of points per processor to be greater than or equal to the square root of the total number of points, and the number of processors to be a power of two. Because of the first constraint, in the 256-point version the maximum number of threads is 16. Due to the second constraint and the fact that some threads in the Cyclops chip are reserved by the system, the maximum number of threads in the 64K-point version is 64. In the figure, we show the relative improvement in performance as percentage bars. For each benchmark we present three bars: total number of cycles, run cycles – in which the threads were busy computing, and stall cycles – in which threads were stalled for resources. Negative bars represent a reduction in the number of cycles, and therefore an improvement in performance. We note that in general the number of run cycles increases for the hardware barrier implementation, while the number of stalls decreases significantly. This is in line with the expectations. The hardware barrier implementation executes more, cheaper instructions, that do not contend for shared memory. The performance gain is about 10% for the 256-point FFT with 16 threads, and about 5% for the 64K-point FFT with 64 threads.

4. Related work

Our design for Cyclops is ambitious, but within the realm of current or near-future silicon technology. Combined logic-memory microelectronics processes will soon deliver chips with hundreds of millions of transistors. Several research groups have advanced processor-in-memory designs that rely on that technology. We discuss some of the projects that are related to Cyclops.

The MIT RAW architecture [1, 21] consists of a highly parallel VLSI design that fully exposes all hardware details to the compiler. The chip consists of a set of interconnected tiles, each tile implementing a block of memory, functional units, and switch for interconnect. The interconnect network has dynamic message routing and a programmable switch. The RAW architecture does not implement a fixed instruction set architecture (ISA). Instead, it relies on compiler technology to map applications to hardware in a manner that optimizes the allocation of resources.

Architectures that integrate processors and memories on the same chip are called Processor-In-Memory (PIM) or Intelligent Memory architectures. They have been spurred by technological advances that enable the integration of compute logic and memory on a single chip. These architectures deliver higher performance by reducing the latency and increasing the bandwidth of processor-memory communication. Examples of such architectures are IRAM [14], Shamrock [10], Imagine [15], FlexRAM [9, 18], DIVA [7], Active Pages [13], Gilgamesh [23], MAJC [19], and Piranha [5]. In some cases, the PIM chip is used as a coprocessor (Imagine, FlexRAM), while in other cases it is used as the main engine in the machine (IRAM, Shamrock, MAJC, Piranha). Cyclops uses the second approach. Another difference is that some architectures include many (32-64), relatively-simple processors on the chip (Imagine, FlexRAM) and others include only a handful (4-8) of processors (IRAM, Shamrock, MAJC, Piranha). Cyclops belongs to the first class.

Simultaneous multithreading [6, 20] 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. Their results 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 [16, 17] 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 Cyclops, in which each thread has its own execution hardware. The Tera approach, however, can tolerate longer latencies to memory and supports the design of a machine without caches.

5. Conclusions

In this paper we have described the Cyclops architecture, a highly parallel processor-and-memory system on a chip. The Cyclops chip is intended to be used as a building block for scalable machines. Cyclops minimizes the impact of memory and functional unit latencies not through complicated architectural features, but through the use of massive parallelism. We have measured the performance attained from two of Cyclops' distinguishing features: its memory subsystem organization and its hardware support for fast interthread barriers. We have demonstrated that a single Cyclops chip can achieve sustainable memory bandwidth on the order of 40 GB/s, similar to a top-of-the-line commercial machine. We have also shown that the fast barrier hardware can improve the performance of an FFT kernel by up to 10%.

As future work, we plan to investigate the performance of Cyclops in more detail [3]. In particular, we want to study the impact of its fault tolerance features. Although we did not discuss this aspect of the Cyclops architecture here, the chip is expected to function even with broken components. For example, if a memory bank fails, the hardware will set a special register to specify the maximum amount of memory available on the chip and will re-map all the addresses so that the address space is contiguous. If thread units fail, there is enough parallelism in the chip so that useful work can still be accomplished. If an FPU breaks, an entire quad will be disabled, but there are 31 other quads available for computation. From a system perspective, in which multiple chips are connected together, an application with knowledge of the machine status can adapt its communication patterns based on chip availability. We have only started to explore the system software components necessary to take advantage of these architectural features. It will be important to characterize the impact of faults on overall chip and system performance.

Finally, we need to discuss two important limitations of the Cyclops architecture. First, combined logic and memory processes have a negative impact: the logic is not as fast as in a pure logic process and the memory is not as dense as in a pure memory process. For Cyclops to be successful we need to demonstrate that the benefits of this single-chip integration, such as improved memory bandwidth, outweigh the disadvantages. Second, due to its single-chip nature, Cyclops is a small-memory system. The external DRAM is not directly addressable and the bandwidth to it is much lower. We can expect future generations of Cyclops to include larger memory, but the current ratio of 250 bytes of storage to MFlop of compute power (compared to approximately 1MB/1MFlop in conventional machines) will tend to decrease.

The result is that Cyclops systems are not single purpose machines such as MD-Grape [12] but are not truly general purpose computers either. Our architecture targets problems that exhibit two important characteristics. First, they should be able to exploit massive amounts of parallelism, on the order of a million processors in very large systems. Second, they should be compute intensive. Examples of applications that match these requirements are molecular dynamics [4], raytracing, and linear algebra.

Finally, we should stress that the results presented in this paper were obtained through simulation. Although we are confident of the general trends demonstrated, the results need to be validated through real measurements in hardware. As we proceed to complete the design of Cyclops and build prototypes, we will have the capability to perform those measurements.

References

[1] A. Agarwal. Raw computation. *Scientific American*, August 1999.

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

[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] G. S. Almasi, C. Caşcaval, J. G. Castaños, M. Denneau, W. Donath, M. Eleftheriou, M. Giampapa, H. Ho, D. Lieber,

J. E. Moreira, D. Newns, M. Snir, and H. S. Warren, Jr. Demonstrating the scalability of a molecular dynamics application on a Petaflop computer. In *Proceedings of the 2001 International Conference on Supercomputing*, pages 393–406, June 2001.

[5] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[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] M. W. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCross, J. Brockman, W. Athas, A. Srivasava, V. Freech, J. Shin, , and J. Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of SC99*, November 1999.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Apporach*. Morgan Kaufmann, second edition edition, 1996.

[9] Y. Kang, M. Huang, S.-M. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *International Conference on Computer Design (ICCD)*, October 1999.

[10] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Frontiers of Massively Parallel Computation Symposium*, 1996.

[11] J. D. McCalpin. Sustainable memory bandwidth in current high performance computers, 1995. http://home.austin.rr.com/mccalpin/papers/bandwidth/.

[12] MD Grape project. http://www.research.ibm.com/grape.

[13] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A computation model for intelligent memory. In *International Symposium on Computer Architecture*, pages 192–203, 1998.

[14] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM: IRAM. In *Proceedings of IEEE Micro*, April 1997.

[15] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. Owens. A bandwidth-efficient architecture for media processing. In *31st International Symposium on Microarchitecture*, November 1998.

[16] 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.

[17] 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.

[18] J. Torrellas, L. Yang, and A.-T. Nguyen. Toward a cost-effective DSM organization that exploits processor-memory integration. In *Sixth International Symposium on High-Performance Computer Architecture*, January 2000.

[19] M. Tremblay. MAJC: Microprocessor architecture for Java computing. Presentation at Hot Chips, August 1999.

[20] 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.

[21] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, pages 86–93, September 1997.

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture (ISCA)*, June 1995.

[23] H. P. Zima and T. Sterling. The Gilgamesh processor-in-memory architecture and its execution model. In *Workshop on Compilers for Parallel Computers*, Edinburgh, Scotland, UK, June 2001.