Fan DR, Yuan N, Zhang JC *et al.* Godson-T: An efficient many-core architecture for parallel program executions. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 24(6): 1061–1073 Nov. 2009

Godson-T: An Efficient Many-Core Architecture for Parallel Program Executions

Dong-Rui Fan* (范东睿), *Member*, *CCF*, *IEEE*, Nan Yuan (袁　楠)
Jun-Chao Zhang (张军超), *Member*, *CCF*, *ACM*, Yong-Bin Zhou (周永彬), Wei Lin (林　伟)
Feng-Long Song (宋风龙), Xiao-Chun Ye (叶笑春), He Huang (黄　河), Lei Yu (余　磊)
Guo-Ping Long (龙国平), Hao Zhang (张　浩), and Lei Liu (刘　磊)

Key Laboratory of Computer Systems and Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China

E-mail: {fandr, yuannan, jczhang, ybzhou, linwei, songfenglong, yexiaochun, huangh, yulei, longguoping, zhanghao, lawrenceliu}@ict.ac.cn

Received March 13, 2009; revised September 28, 2009.

Abstract Moore's law will grant computer architects ever more transistors for the foreseeable future, and the challenge is how to use them to deliver efficient performance and flexible programmability. We propose a many-core architecture, Godson-T, to attack this challenge. On the one hand, Godson-T features a region-based cache coherence protocol, asynchronous data transfer agents and hardware-supported synchronization mechanisms, to provide full potential for the high efficiency of the on-chip resource utilization. On the other hand, Godson-T features a highly efficient runtime system, a Pthreads-like programming model, and versatile parallel libraries, which make this many-core design flexibly programmable. This hardware/software cooperating design methodology bridges the high-end computing with mass programmers. Experimental evaluations are conducted on a cycle-accurate simulator of Godson-T. The results show that the proposed architecture has good scalability, fast synchronization, high computational efficiency, and flexible programmability.

Keywords many-core, parallel computing, multithread, data communication, thread synchronization, runtime system

1 Introduction

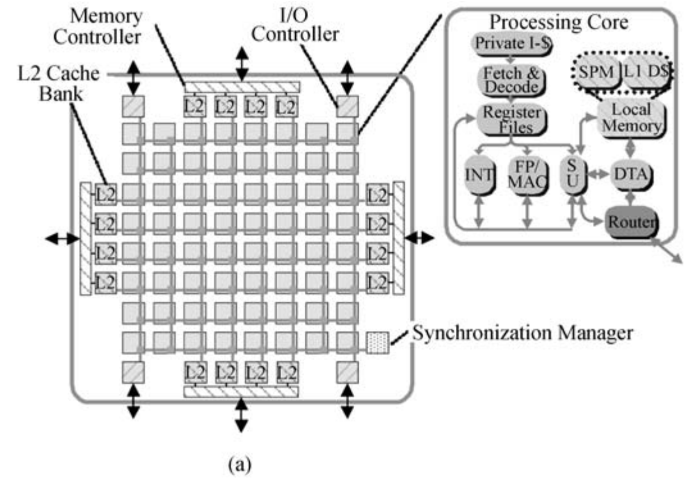
The previous researches of increasing the single thread performance through increasing the clock frequency are now hitting the so-called *Power Wall* and *ILP Wall*[1]. The computer industry has widely consensus that the incremental performance must largely come from increasing the number of processing cores on a die. This has led to the swift changes in computer architectures in recent years: multi-core processors are popular, and many-core processors begin to spread. The Moore's Law suggests that the number of on-chip processing cores doubles every generation. It is anticipated that the future microprocessors will accommodate tens, hundreds, even thousands of processing cores. Although many-core processors provide tremendous computational capability, expressing and exploiting the parallelism correctly and efficiently from such processors remain a grand challenge.

Parallelization always requires much programming effort, although it is not necessary to parallelize all sequential applications. Even though the parallel programs work correctly, performance tuning can be much daunting. Parallel programming brings lots of complex problems that are highly related to the performance issue, for example, managing conflict accesses to shared resource, synchronizing between threads, and so on. In many cases, the big programming effort cannot be transformed into performance gaining. This problem is unlikely resolved by software programming infrastructure alone. Our solution is to provide a widely-used programming paradigm and extremely efficient architectural supports for multithreaded programs. The programmers majorly focus on expressing parallelism, while the processor and runtime system concentrate on efficient parallel executions.

Regular Paper

*Corresponding Author

Supported by the National Basic Research 973 Program of China under Grant No. 2005CB321600, the National High-Tech Research and Development 863 Program of China under Grant No. 2009AA01Z103, the National Natural Science Foundation of China under Grant No. 60736012, the National Science Fund for Distinguished Young Scholars under Grant No. 60925009, and the Beijing Natural Science Foundation under Grant No. 4092044.



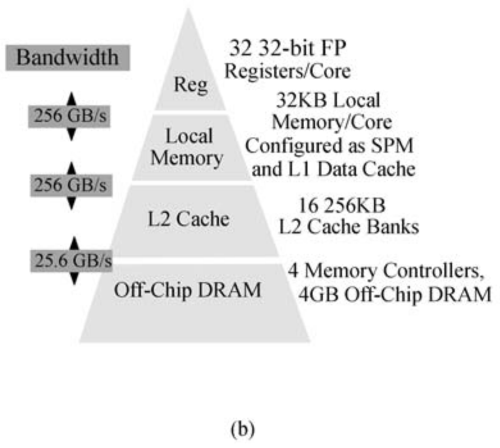

Bandwidth
256 GB/s
256 GB/s
25.6 GB/s
Reg
Local Memory
L2 Cache
Off-Chip DRAM
32 32-bit FP Registers/Core
32KB Local Memory/Core Configured as SPM and L1 Data Cache
16 256KB L2 Cache Banks
4 Memory Controllers, 4GB Off-Chip DRAM
(b)

Fig.1. Overview of Godson-T processor. (a) Microarchitecture of Godson-T. (b) Memory hierarchy and bandwidth of Godson-T.

This paper gives an introduction to Godson-T, a many-core architecture being developed with deep submicron process. Godson-T provides efficient architectural support for two fundamental multithreading operations: data communication and thread synchronization. In the rest of the paper, Section 2 gives an overview of Godson-T processor. Section 3 describes the Godson-T multithreaded programming environment and runtime system. Section 4 highlights the architectural supports for efficient thread communication and synchronization. The experimental results and analysis are shown in Section 5. Related work and conclusion are presented in Section 6 and Section 7, respectively.

2 Overview of Godson-T Processor

As shown in Fig.1, the Godson-T processor has 64 homogeneous, in-order and dual-issue processing cores running at 1 GHz. The 8-pipeline processing core supports MIPS ISA and the synchronization instruction extensions. Two floating-point operations including a multiply-accumulation can be issued to fully-pipelined function units in a cycle. The peak floating-point performance of Godson-T is 192 GFlops. Each processing core has 16 KB 2-way set-associative private instruction cache and 32 KB local memory. The local memory functions as a 32 KB 4-way set-associative private data cache in default. It also can be configured as an explicitly-controlled and global-addressed Scratchpad Memory (SPM), or a hybrid of cache and SPM. A Data Transfer Agent (DTA) is built in each core for fast data communication. When the processing core is doing calculations, DTA can be programmed to manage various patterns of data transfer asynchronously.

In addition, there are 16 address-interleaved L2 cache banks (256 KB each) distributed along four sides of the chip. The L2 cache is shared by all cores and can serve up to 64 cache accessing requests in total. Four L2 cache banks in the same side of the chip share a memory controller. The memory hierarchy to Godson-T is shown in Fig.1(b).

A dedicated synchronization manager provides architectural support for efficient mutual exclusion, barrier and signal/wait synchronization. The 8×8, 128-bit-width packet-switching 2-D mesh network connects all on-chip units. The mesh network employs the deterministic X-Y routing policy and provides 2 TB/s on-chip bandwidth among 64 processing cores.

3 Software Programming Environments

Given a many-core processor like Godson-T, the challenge is how to efficiently utilize the large on-chip computational capability. In this section, we give a brief introduction to Godson-T's software programming environments.

Although some former parallel programmers argue that programming with threads is error-prone[2], multithreading is ironically the most popular parallel programming paradigm in the real world. Programming with threads is not so difficult for majority of cases[3−4]. Programming on Godson-T is based on multithreading. Providing a simple and reasonable programming abstraction is undoubtedly critical to programmability. The software stack of Godson-T platform is shown in Fig.2. Conventional C programming and its tool chain are adopted. Besides, we provide a Pthreads-like C library for task management. Therefore, a large amount of parallel program sources written with Pthreads can be ported onto Godson-T conveniently. The library provides a rich set of APIs for task management, including batch thread management which can significantly reduce tasking overhead by grouping identical operations in batches. Programmer is responsible for creating, terminating and synchronizing tasks by inserting appropriate Pthreads-like APIs.

We developed a software runtime system — GodRunner[5], focusing on efficiently abstracting a large number of hardware threading units and dynamic load-balancing. GodRunner task model (in our terminology, task resides in software, while thread resides in hardware) adopts *create-join* method inspired by *Pthreads*. GodRunner task does not support preemptive execution, because frequent context-switch incurs save-restore overhead and cache thrashing. Therefore, a task will keep on executing on a threading unit until it terminates. Non-preemptive execution can make task initialization simple and fast, for example, the stack is statically allocated to each threading unit, which avoids the time-consuming dynamic allocation. GodRunner permits programmers to create more tasks than hardware thread units, and transparently maps them to hardware thread units at runtime. As shown in Fig.3. GodRunner is responsible for swapping the completed

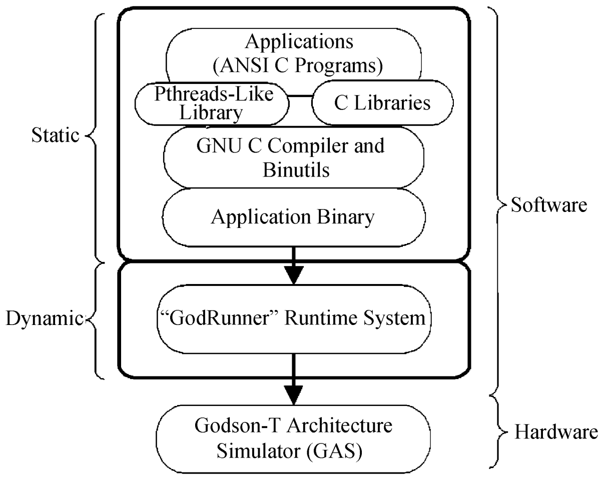

Applications (ANSI C Programs)
Pthreads-Like Library
C Libraries
GNU C Compiler and Binutils
Application Binary
"GodRunner" Runtime System
Godson-T Architecture Simulator (GAS)
Static
Dynamic
Software
Hardware

Fig.2. Software stack of Godson-T platform.

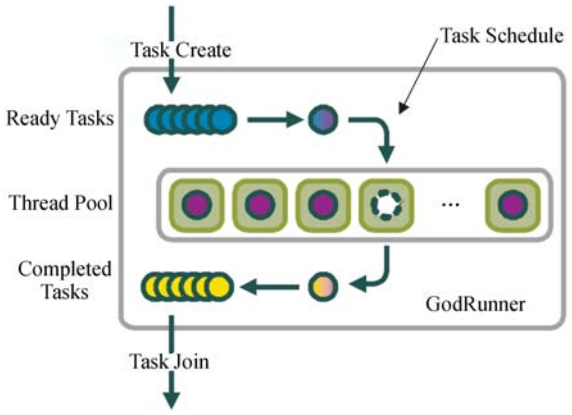


Fig.3. Task scheduling by using GodRunner.

task out and scheduling a new one in a thread unit. Different task scheduling algorithms can be incorporated into GodRunner flexibly. We have implemented two well-known task scheduling algorithms for efficient dynamic load balancing: work-stealing[6] and conditional-spawning[7].

4 Architectural Supports for Multithreading

Unlike sequential programs, threads in parallel programs may be dependent on each other. Hence, a communication mechanism must be provided for managing data transfer among the threads. Since the threads execute asynchronously, additional synchronization operations must be used to realize the data transfer from a producer to a consumer at the correct time. A multithreaded program, no matter how complex it is, is always orchestrated by a collection of computation, communication and synchronization operations. From this point of view, Godson-T provides efficient and ease-to-use architectural supports for data communication and thread synchronization. In this section, firstly we will introduce the novel data communication mechanisms of Godson-T. Then we will introduce the novel thread synchronization mechanisms of Godson-T.

4.1 Thread Communication

4.1.1 Region-Based Cache Coherence Protocol

The conventional directory-based cache coherence protocols for large-scale parallel architecture guarantee that any read and write requests at a memory location perform in partial order. However, this makes the design and verification of the protocol very complex. Another problem of the directory-based protocols is that their hardware overhead would increase quickly when more cores and larger caches are put into a single chip, because the size of a directory is proportional to both the number of cores and the cache size. Furthermore, the directory-based protocols also suffer from unnecessary cache invalidations, like invalidations caused by false sharing. These unproductive invalidations degrade both the performance and the scalability. In Godson-T, these problems are overcome by a novel Region-based Cache Coherence (RCC) protocol.

Maintaining all data (either shared or private) coherent in cache hierarchy is very expensive[8]. So the principle of our proposed cache coherence protocol is that the coherence of data should be lazily guaranteed upon request. The shared access and private access are expected to be distinguished. The cache hierarchy benefits from understanding different types of accesses and takes corresponding consistency operations to minimize the side effect. As a result, RCC is proposed. In our terminology, a region is a code sequence beginning with an open region primitive and ending with a close region primitive. Memory accesses inside the region are regarded as shared accesses and guaranteed coherent, while accesses outside the region are not guaranteed coherent. Ideally, regions only embrace the shared accesses. The flexibility of the RCC allows users to declare the regions in either coarse grain or fine grain. This is convenient for users to ensure the correctness of the program at first and then make incremental performance improvement by reducing the size of regions. It is convenient to port a parallel program to Godson-T correctly by just inserting these primitives at synchronization points, such as mutual exclusive locks.

As mentioned in Section 2, the 32 KB local memory on each core can be configured as a private L1 data cache, and 16 address-interleaved L2 cache banks are shared by all L1 caches. In this two-level cache hierarchy, data communication performs through the shared L2 cache, so that the newest value of shared data should be put to or fetched from the L2 cache. This is carried out by region-based consistency primitives and corresponding consistency operations in L1 cache. The protocol does not involve the hardware directory or those sophisticated protocol state machines.

Since the cache protocol updates data lazily, all shared data accesses are required to be figured out explicitly for the best performance. The shared and private data accesses should be differentiated on such platforms containing so many cores. The underlying cache hierarchy should perform only necessary expensive consistency operations for good scalability. This objective can be achieved by simply declaring the shared object as "*shared*", as in UPC[9]. Thus, accessing to the shared objects can be automatically identified and guaranteed by RCC protocol.

4.1.2 Orchestrating Data Movement

Feeding the sufficient data to the function units is a great challenge due to the tremendous on-chip computational capability. This problem is relieved on Godson-T by overlapping the communication and the computation. Data Transfer Agent (DTA) is provided on each core to support the fast and asynchronous data transfer. Unlike traditional DMA mechanism, DTA is a more advanced communication-centric coprocessor in Godson-T design. It provides a versatile set of operations for users and cooperates with network-on-chip, enabling less restrictive and more efficient data communication on many-core architecture.

The conventional cache organization permits only *vertical* data communication in memory hierarchy (i.e., between different levels of memory hierarchy). The communication latency might be long and the communication bandwidth might be restricted by the memory hierarchy (e.g., bandwidth degraded due to conflict accesses to the same L2 bank, the network congestion and the L2 cache miss). So that, the *horizontal* data communication (i.e., from the local memory to other cores' local memory directly) is proposed to utilize the low latency of the on-chip communication and the tremendous on-chip bandwidth (2 TB/s on Godson-T) of many-core architecture to compensate the insufficient off-chip bandwidth. DTA supports both the vertical and the horizontal data transfer, as shown in Fig.4(a). It should be pointed out that DTA also support an additional vertical operation which prefetches data blocks from off-chip memory into on-chip L2 cache. This is quite useful for the programs with large data set that exceeds the L2 cache size. The off-chip memory latency can be tolerated by double-buffering on L2 caches and overlapping L2 prefetch operations and on-chip computations. Horizontal data communications are also supported among different SPMs. Compared with vertical operations, the horizontal operations make better utilization of the tremendous on-chip bandwidth. Therefore, keeping data on SPM as much as possible and communicating through horizontal DTA operations are encouraged.

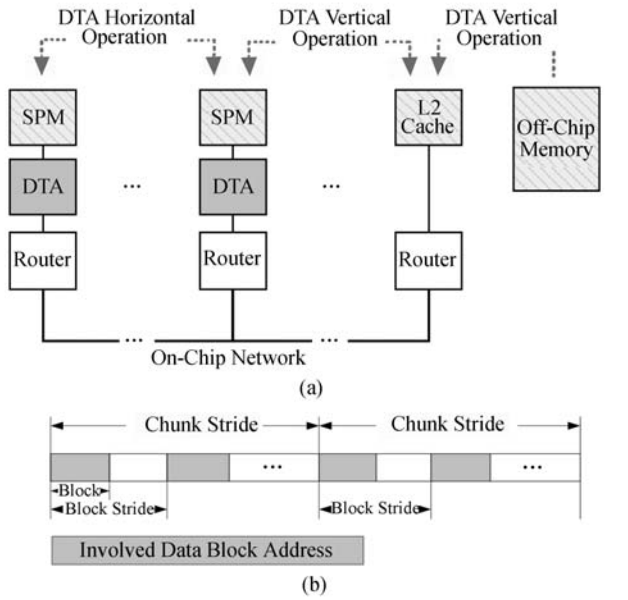


Fig.4. Data communications supported by DTA operations. (a) Vertical and horizontal DTA operations. (b) 2D strided DTA operations.

Besides accessing the continuous data blocks, DTA also supports data accessing with 2-D stride: block stride and chunk stride, as shown in Fig.4(b). Moreover, DTA operations are not restricted to one-to-one data communication. It can recognize the generated addresses of different destinations automatically. It can communicate with different memory units in an operation, and can just move data blocks inside the local SPM. With the 2-D stride operations of DTA, the programmers would be much free from the restriction of data layout and the communication patterns in algorithms. For example, it is convenient to retrieve a matrix, and transpose it on-the-fly through a DTA operation. The flexibility is also enhanced by supporting up to 4 outstanding operations in DTA.

Another important feature of DTA is its automatic network-load aware function. DTA sends a "ping" request to a destination after sending a bunch of data requests to the destination. When the "echo" response is returned, DTA can adjust the injection bandwidth to the network according to the round-trip latency automatically. It is unnecessary to send special "ping" request in DTA load operations since the round-trip latency can be estimated by the load request and the data response. This function is critical to preserve the delivered bandwidth of the network. If DTAs send out their packets into network continuously and blindly, the on-chip network will be congested soon. This results in poor utilization of the bandwidth.

Programming SPM and DTA on Godson-T is much like programming a conventional DMA engine. Currently, it is done manually on Godson-T. Orchestrating data movement explicitly could be painful. However, we believe more advanced programming models and compiler techniques will alleviate this burden. There are already a lot of promising works in this field. For example, the Partitioned Global Address Space (PGAS) languages[9−11], developed for clustered computers, are naturally fit for Godson-T. The PGAS languages, assuming a hardware environment similar to Godson-T, can program DMA automatically. Other examples include Sequoia[12] and Hierarchical Tiled Array[13]. These languages focus on organizing data reuse and movement automatically between different levels of memory hierarchy, where DMA is also automatically used.

4.2 Thread Synchronization

4.2.1 Synchronization Without Memory

Synchronization in the conventional shared-memory programming is based on memory communication mechanism (e.g., cache coherence protocol). We propose a synchronization scheme including several synchronization primitives and a novel synchronization manager to handle mutual exclusion, barrier and producer/consumer synchronization effectively. The main difference between our scheme and conventional ones is that our scheme cooperates with memory communication but does not build on it. This separation makes our synchronization scheme extra-efficient.

Table 1 listed dedicated synchronization primitives. These primitives are not associated with the memory locations. Instead, they use *id* to identify different synchronizations. It offers users another space to conveniently express the synchronization. An example is the mutable accessing to a shared array: the address of element can be used as the *id* of the lock to access the shared element, so that the program could be easily orchestrated with fine-grained locks without the array of lock variables in memory. Compared with the conventional synchronization, the proposed method does not have to emulate synchronization with a sequence of memory accesses. The dedicated synchronization primitives are also more analyzable for compiler and hardware, enabling more opportunities for optimization.

Table 1. Architecturally Supported Synchronization Primitives

Primitive	Function
read_lock *id*	Acquire a read lock denoted by *id*. The requested core is sleeping until the acknowledgement from SM is received. Note that the acknowledgement may be received before lock since try_lock is used.
write_lock *id*	Acquire a write lock denoted by *id*. The requested core is sleeping until the acknowledgement from SM is received. Note that the acknowledgement may be received before lock since try_lock is used.
unlock *id*	Release a lock denoted by *id*.
signal *id*, *count*	Produce a semaphore *id* that has *count* consumers.
wait *id*	Consume a semaphore *id*.
barrier *id*, *count*	Perform the barrier synchronization denoted by *id* and involved *count* core number. The requested core is sleeping until the acknowledgement from SM is received.

An on-chip Synchronization Manager (SM) is proposed to handle the primitives listed above efficiently. SM is a 128-entry and 4-way set-associative table in which each entry records a synchronization request. The overflow of the table will trigger a software replacement process. SM handles mutual exclusion by using queue-based MCS algorithm[14]. In a similar way, the SM collects barrier requests with the same barrier *id*. When the last barrier request arrives, SM sends out ACK messages to all involved processing cores. The ACK message with longer routing distance has higher priority to be sent out first. Hence, all involved cores restart at approximately the same time, which is helpful to reduce the overhead of load imbalance. SM can also control producer-consumer synchronization by handling the signal/wait pair.

4.2.2 Full/Empty Bit Synchronization

Godson-T supports the fine-grain synchronization by associating the full/empty bits with the local SPMs. The full/empty bit tagged on memory cell indicates the presence of data on the memory location, e.g., "1" for "full", and "0" for "empty". Compared with the previous works about full/empty bits[15–16], our scheme has three advantages. 1) It is built with on-chip memory, not off-chip memory. Hence, the synchronization is extremely fast. This also avoids the sophisticated cache hierarchy design when the full/empty bit mechanism is associated with off-chip memory. 2) 8-bit tag of each data cache line changes to the full/empty bits when the local memory is configured as SPM, so no more hardware budget is required. 3) Besides synchronized load/store instructions, the data-driven synchronization mechanism is naturally incorporated into the DTA operations. The synchronized DTA operations are expected to improve the utilization of the function units in memory-intensive applications. Fig.5 is the illustration of the performance improvement by using the synchronized DTA operations.

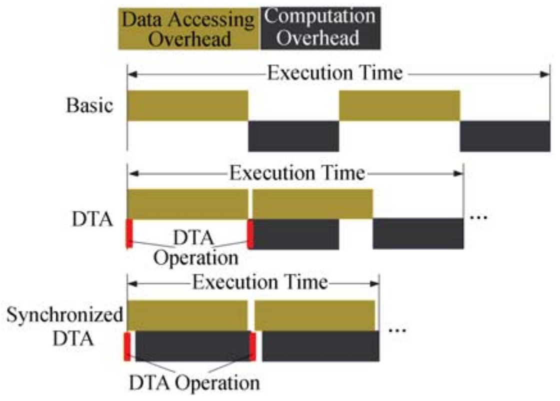


Fig.5. Synchronized DTA operations.

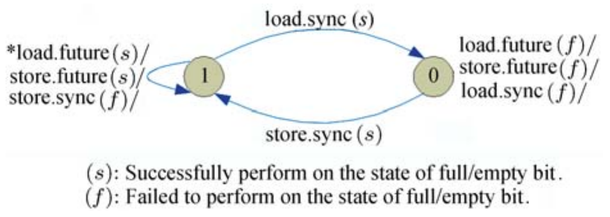


Fig.6. State machine of full/empty bit.

There are two types of fine-grain synchronization instructions on Godson-T: *sync* type and *future* type. The former is used for producer-consumer style synchronization, whereas the latter is used for future data object protection. Fig.6 illustrates the state machine of full/empty bit. An instruction will fail and retry when it operates on an improper state of full/empty bit, and the state of full/empty bit will not be changed. Various synchronization patterns can be constructed based on these two basic synchronization types. For example, the single-writer-multiple-reader synchronization can be implemented with the cooperation of *store.sync* and *load.future*.

5 Experimental Results and Analysis

In this section, experimental platform and benchmarks are introduced at first. The design features of Godson-T are examined through experimental evaluation and analysis in the following subsections.

5.1 Experimental Platform and Benchmarks

Experiments are conducted on a cycle-accurate simulator of Godson-T architecture. Major architectural parameters used in the experiments are listed in Table 2. To compare the synchronization efficiency, we also setup an SMP machine. This machine has eight 2.4 GHz AMD Opteron processors and each processor has dual cores.

5.1.1 Evaluation of Data Communication

Four kernels of SPLASH-2[17] and two bioinformatics applications (pFind[18] and iBLASTP[19]) are ported to Godson-T to examine the efficiency of the proposed cache organization. For executing correctly on the Region-based coherence protocol, the program sources are slightly modified for correctness, e.g., to identify and tag the regions. pFind is a search engine system for the automated peptide and protein identification from the tandem mass spectra. The subset-seed based iBLASTP is a protein banks comparison algorithm derived from the well-known BLAST. Two important kernels including SGEMM (Single-precision General Matrix Multiplication) and FFT are composed to exploit the potential of high-performance computing of Godson-T. Intensive optimizations are manually applied on the kernels, such as using the SPM and DTA of Godson-T. These kernels are frequently used in the high-performance computing community, representing the dense algebra and spectral method in 13 "motifs"[1].

5.1.2 Evaluation of Thread Synchronization

The microbenchmarks and methodology described in [20] are adopted to evaluate the mutual exclusion and barrier synchronization mechanism of Godson-T. These microbenchmarks perform stress testing of synchronization primitives. We use two micro-benchmarks to evaluate the efficiency of the producer-consumer synchronization supported by the full/empty-bit mechanism of Godson-T: the 2-D wavefront, which is common in scientific codes with abundant single-writer-multiple-reader synchronizations; and the Loop 6 from the Livermore Loop benchmarks[21], which computes linear recurrence equations with abundant single-writer-multiple-reader synchronizations.

Table 2. Simulation Parameters of Godson-T Micro-Architecture

Processing Core	
Processing core	64 cores, in-order, dual-issue, each running at 1 GHz
Load-to-use latency	3 cycles
FMAC unit latency	4 cycles
Memory Subsystem	
L1 I-cache	16 KB, 2-way set associative, 32 B/cacheline
Local memory	Configured to 32 KB SPM, 16 64-bit-width SRAM sub-banks with 2 memory ports each (1 for read, 1 for write)
L2 cache	16 banks, total 4 MB, 8-way set-associative, 64 B/cacheline
Memory controller & off-chip DRAM	4 memory controllers, running at the same clock rate as the processing core, 64-bit FSB. (Each memory controller controls a 1 GB DDR2-800 DRAM. DRAM clock is 400 MHz, $tCAS = 5$, $tRCD = 5$, $tRP = 5$, $tRAS = 15$, tRC=24 measured at memory clock.)
Contentionless Latency	
L1 I-cache hit	1 cycle
SPM load/store hit	1 cycle
Mesh network	2 cycles per hop
L2 hit	12∼40 cycles according to routing distance
Off-chip memory	62∼120 cycles according to routing distance and DRAM access pattern (e.g., whether the access is on the same row as previous ones)
Synchronization	Each synchronization request to SM and acknowledgement from SM spend 6∼66 cycles. Lock or barrier request consume additional cycles until the synchronization dependence is resolved, e.g., a barrier request should wait until all barrier requests from all involved processing cores are collected in SM.

5.2 Main Observations

Observation 1 (See Subsection 5.3 for Details). Our experimental results show that the proposed thread communication mechanisms of Godson-T have the following advantages.

1) The experiment shows that our low-cost cache coherence mechanism on Godson-T assures good scalability for traditional multithreaded programs. (See Subsection 5.3.1)

2) DTA significantly boosts the performance of both vertical and horizontal data communication, and feeds ample data to all processing cores to reach the performance as high as the peak performance of Godson-T. (See Subsection 5.3.2)

Observation 2 (See Subsection 5.4 for Details). Our experiments demonstrate that the features of our proposed thread synchronization mechanisms of Godson-T have the following advantages.

1) The architectural support of synchronization, including mutual exclusion and barrier synchronization, makes the synchronization even thousands of times faster than Pthreads on traditional SMP machine, and also outperforms atomic-instruction based synchronization on Godson-T. (See Subsection 5.4.1)

2) On-chip memory-based full/empty bit synchronization of Godson-T significantly reduces the synchronization overhead. The DTA operations combined with fine-grain synchronization can further reduce the communication overhead. (See Subsection 5.4.2)

5.3 Experimental Results and Analysis of Thread Communication

5.3.1 *Evaluating the Scalability of Cache Coherence Protocol*

Fig.7 illustrates the speedup of two bioinformatics applications and four kernels of SPLASH-2. Both bioinformatics applications achieve excellent speedup, because they are embarrassingly parallel and computation-intensive. The speedup of the four SPLASH-2 kernels with default input data-set size is comparable or superior to [17], even though [17] uses ideal memory. The sub-linear speedup of the kernels is mainly due to the on-chip interconnection congestion (FFT), the capacity miss of 4 MB L2 cache (RADIX, CHOLESKY), and the load imbalanced algorithm (LU, CHOLESKY). The performance of these kernels can be improved after adopting load-balanced algorithm and

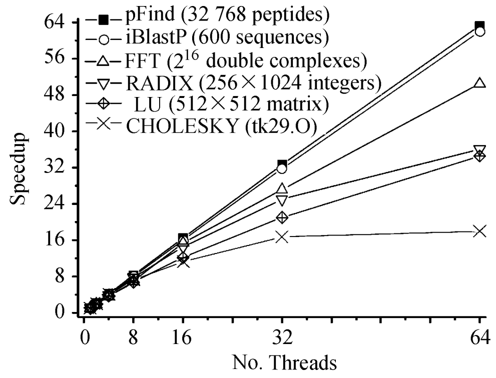


Fig.7. Speedup of the bioinformatics applications and SPLASH-2 benchmarks on Godson-T. (Characters in parentheses means the input data set.)

optimizing for Godson-T. The optimized FFT kernel on Godson-T is 10.5 times faster than that in SPLASH-2 with the same algorithm. Nevertheless, the efficiency and scalability of the proposed region-based cache coherence protocol has been solidly proved for conventional multithreaded programs.

5.3.2 Evaluating On-Chip DTA

The SGEMM and 1-D FFT are optimized on Godson-T by directly utilizing architectural support such as SPM and DTA.

SGEMM. In the 1280×1280 SGEMM kernel, tiling and double buffering are used for each level of memory hierarchy. We choose 40×40 matrix multiplication for each core to fill SPM storage, so that the whole processor calculates 320×320 matrix multiplication in Cannon's algorithm. Each of the iteration in Cannon's algorithm is synchronized with the barrier primitive. When each core does calculation, DTA gets data from neighboring cores through horizontal DTA operation and prefetch another 320×320 matrix block from off-chip memory to L2 cache for the next-turn multiplication through vertical DTA operation. Both on-chip and off-chip bandwidths are sufficient in this case. Almost all the data communication overhead is hidden by computation. The detail algorithm can be found in [22].

1-D FFT. We use the well-known six-step 1-D FFT algorithm described in SPLASH-2 with a 64 K single-precision floating-point complex array. The complex array is treated as a 2-D matrix, and needs to be transposed in steps 1, 4 and 6. These transposes can definitely benefit from the strided DTA operations. In step 1, columns of the matrix are transferred from the off-chip memory to each SPM by vertical DTA operations. The FFT calculations in steps 2, 3 can be overlapped with step 1. In step 4, the transpose is done through exchanging columns of matrix among SPMs by DTA horizontal operations. This can be overlapped with step 5. Step 6 is a vertical DTA operation to store the results from SPM to the L2 cache.

Fig.8 demonstrates the efficiency of using SPM and DTA. All the kernels running on different configurations use the same algorithm described above. The data communication in cache configuration can only be realized through shared L2 cache vertically. SPM configuration uses load/store instructions to perform horizontal communication. The DTA configuration uses DTA horizontal operations. The theoretical performance is estimated by assuming that data always exist on SPM when they are required. Thus the difference between theoretical and actual performance mainly represents the overhead of data communication. The performance on cache organization slightly outperforms that on SPM, even though SPM enables the horizontal communication. The reason is that a cache miss causes a whole cache line refilled — this is a kind of data prefetching that would benefit regular data accessing in these kernels. But this prefetching will not happen in basic SPM configuration. Performance has been dramatically accelerated by DTA operations since much data communication latency has been removed from the critical path of the execution. But there still exists a gap between actual performance and theoretical performance for FFT, since a fraction of data accessing latency cannot be hidden in computation for this algorithm.

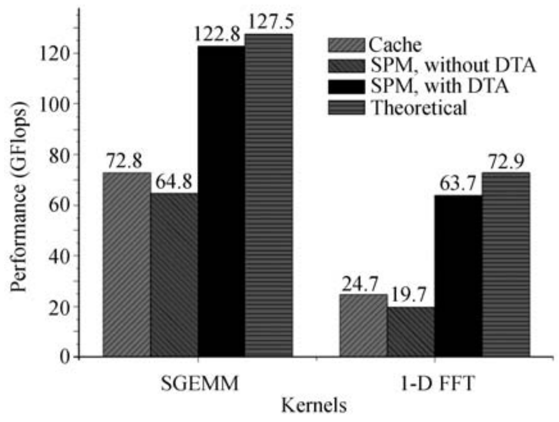


Fig.8. Performance comparisons of SGEMM and 1-D FFT.

To illustrate the benefit of the flow-control function of DTA, we take SGEMM as an example. When 64 threads start to run, each of the threads needs to vertically retrieve three 40×40 matrices from L2 caches to its local SPM. During the retrieving, the utilized bandwidth of our proposed DTA is 5.6 times larger than the DTA without network flow-control.

Table 3 concludes the performance and efficiency of

Table 3. Performance (GFlops) and Computational Efficiency (%) Comparisons

Processor	Benchmark			
	SGEMM		1-D FFT	
	Efficiency	Performance	Efficiency	Performance
Godson-T	95.9% *	122.81	33.2%	63.72
IBM Cell	99.9%**	204.70	20.4%	41.80
Cyclops-64	43.4%	13.90	25.8%	20.70
GTX8800	60.0%	206.00	29.9%	155.00

*The SGEMM kernel contains only multiply-and-add operation, so that the ideal peak performance is measured by the multiply-and-add function unit, which is 128 GFlops.

**Efficiency of SGEMM on Cell is slightly better than that on Godson-T, because 256 KB SPM for each SPE on Cell makes better utilization of data locality.

SGEMM and 1-D FFT kernels on Godson-T. The best performance reported publicly on several other parallel processors is also listed in [23–27]. As we can see, the efficiency of Godson-T is comparable or superior to other processors. Compared with other processors, Godson-T provides explicit but simple memory hierarchy abstraction with fast and versatile data communication mechanisms (e.g., horizontal DTA operation), which is very important to obtain extremely high computational efficiency.

5.4 Experimental Results and Analysis of Thread Synchronization

5.4.1 Evaluating Synchronization Without Memory

A series of micro-benchmarks and methodologies described in [20] are adopted to stress testing the synchronization overhead of Godson-T. Fig.9 depicts the overhead of the mutual exclusion synchronization. Fig.9(a) shows that the overhead of FAA-based lock (implemented by fetch-and-add atomic memory operations) and SM-based lock (primitives handled by synchronization manager) on Godson-T is approximately 10 times lower than that on the SMP machine when there is no lock contention. Fig.9(b) shows the transferring overhead of locks (the period between a thread releasing a lock and the lock acquired by another thread) when all active threads contending for the same lock. The SM-based lock overhead remains constant as the number of threads increases, while the cost of other types of locks increases. Besides, the transfer time of SM-based lock costs much lower than the others. Fig.9(c) shows the average load latency of the load operations issued in critical sections when there are multiple threads contending for the same lock. The test evaluates the side effect of lock contention on network-on-chip. In this figure, these locks do not have obvious influence on the network. In conclusion, SM-based lock greatly outperforms other locks, and the results also reveal its good scalability even though there is a heavy lock contention. Note that the performance of FAA-based lock on Godson-T, which represents lock mechanism based on atomic operations, is also worse than SM-based lock even though they share the same on-chip latency and bandwidth.

Fig.10 shows the overhead of the barrier synchronization. Fig.10(a) shows that the raw overhead of barrier without workload between barriers. It can be seen that the overhead on Godson-T is significantly less than the SMP machine. As the number of threads increases, the overhead of barrier synchronization slightly grows

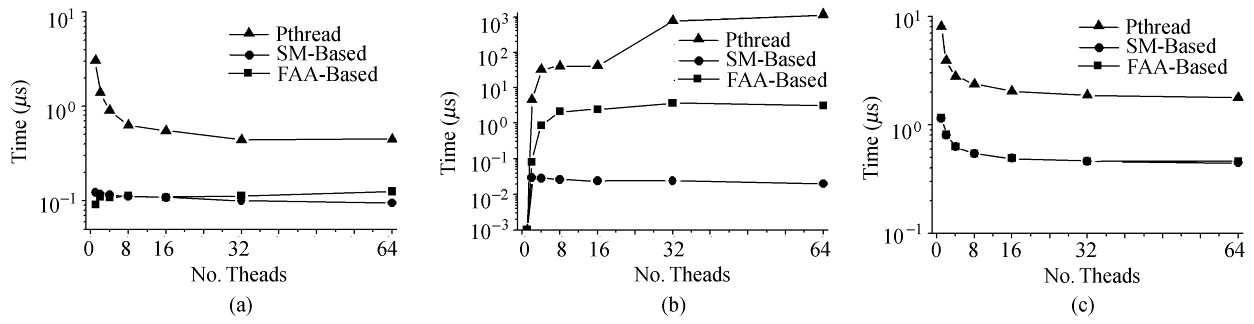


Fig.9. Overhead of mutual exclusion synchronization. (a) Lock overhead without lock contention. (b) Lock transferring overhead. (c) Average time for each load.

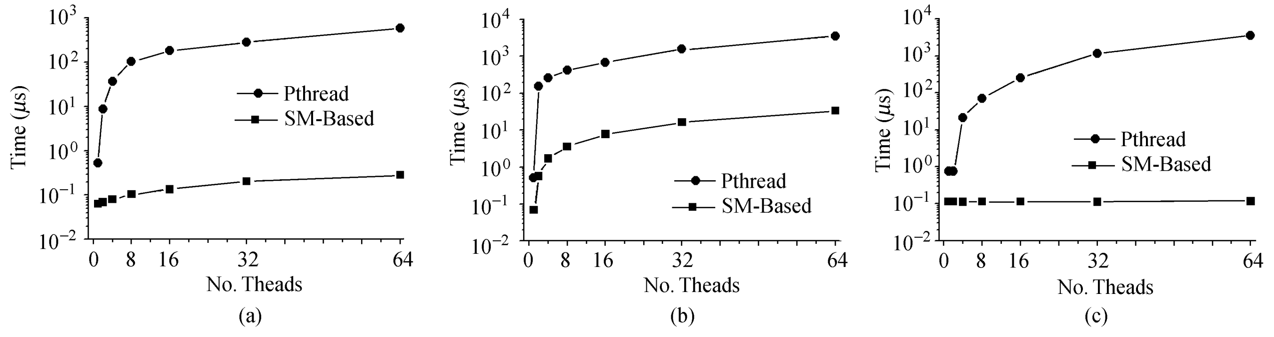


Fig.10. Overhead of barrier synchronization. (a) Barrier overhead without workload. (b) Barrier overhead with load imbalancing. (c) Average time of each load.

on Godson-T. But the overhead grows by several hundreds or even thousands of times on the SMP machine. The result proves the good scalability of our approach on thread synchronization. Fig.10(b) shows the overhead of the barrier synchronization of load-imbalancing threads. The average overhead on the SMP machine is larger than Godson-T by hundreds of times. Fig.10(c) shows the negative effect on network when there are multiple threads performing the barrier operations. The Pthreads barrier makes more and more congestion as the number of threads increases, while this phenomenon does not appear on Godson-T.

5.4.2 *Evaluating Full/Empty Bit Synchronization*

We use two micro-benchmarks to evaluate the efficiency of the producer-consumer synchronization supported by the full/empty-bit mechanism of Godson-T: the 2-D Wavefront and Livermore Loop 6. In 2-D Wavefront, each element of a 2-D matrix $\boldsymbol{A}$ is determined by three adjacent elements: $\boldsymbol{A}[i][j] = \boldsymbol{A}[i-1][j-1]+\boldsymbol{A}[i-1][j] + \boldsymbol{A}[i][j-1]$. The first row and the first column of matrix $\boldsymbol{A}$ are initialized before execution. $\boldsymbol{A}$ is a 512×512 double-precision matrix in our implementation. To parallel the program, rows of $\boldsymbol{A}$ are distributed on SPMs and assigned to threads in round-robin. The dependence of elements is guaranteed by ldc1.sync and sdc1.sync operating with full/empty bit. Performance is compared with a serial version of the program. As illustrated in Fig.11, the parallelized program shows an optimal speedup over the serial one.

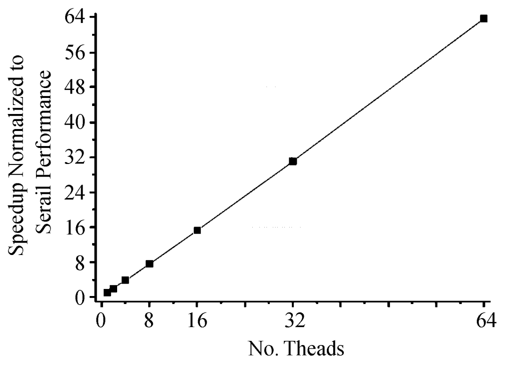


Fig.11. Speedup of 2-D Wavefront.

Livermore Loop 6 is described as follows:

```
for ($i = 1$; $i < n$; $i++$)
    for ($k = 0$; $k < i$; $k++$)
        $\boldsymbol{W}[i] += B[k][i] * \boldsymbol{W}[(i-k)-1]$;
```

In this loop, $\boldsymbol{W}[i]$ depends on $\boldsymbol{W}[i-1] \sim \boldsymbol{W}[0]$ recursively. The single-writer-multiple-reader synchronization is easily guaranteed by ldc1.future and swc1.sync. Besides, before calculating $\boldsymbol{W}[i]$, we issue a synchronized DTA operation to transfer the required part of $\boldsymbol{B}$ to the corresponding SPM and set the full/empty bits. Then we use an ldc1.sync to load $\boldsymbol{B}[k][i]$ from the SPM when it is required.

The performance of the parallelized program described in [21] with coarse-grained synchronization is compared with our programs using full/empty bits, with or without DTA support. The result in Fig.12 shows that compared with the coarse-grained synchronization version, the fine-grain synchronization gains 70% speedup of the performance, and the synchronized DTA operation further increases 56.8% performance.

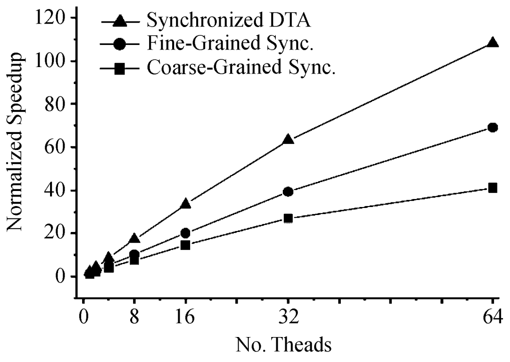


Fig.12. Speedup of Livermore loop 6.

6 Related Work

There has been a considerable amount of previous researches on multi-core and many-core architectures. It is well beyond the scope of this paper to cover all such related works. We will summarize the works that most closely resemble the techniques proposed in our paper.

Location consistency cache protocol[28] is also a region-based cache coherence protocol, in which the region contains only one memory access. The difference between CRF[29] and our work is that our cache performs consistency operation guided by the region, while the operation is guided by cache instructions in CRF. Our cache protocol tallies with Scope Consistency[30] memory model proposed in DSM, but implemented in hardware approach. Besides, the producer region and the consumer region are differentiated in our cache coherence protocol to avoid unnecessary and expensive coherence operations. BulkSC[31] also supports a region-based sequential consistency (SC) memory model. A region in BulkSC is called a chunk. BulkSC dynamically groups sets of consecutive instructions into chunks that appear to execute sequentially. BulkSC has hardware support for chunk checkpoint and rollback to ensure SC. It is fundamentally different from RCC. RCC is much weaker than SC and a region in RCC is statically decided. A motivation of RCC is to minimize the unnecessary synchronizations. The idea of RCC is to distinguish different memory accesses. This is a major step over traditional memory consistency models which consider all memory accesses with the same way.

Several parallel architectures adopt explicitly programmable on-chip memory rather than coherent data cache, such as IBM Cell[32], Cyclops-64[33], nVidia GeForce[34]. Godson-T provides SPM enabling both vertical and horizontal data communication. Moreover, the efficient data communication and fine-grained synchronization mechanisms are provided with Godson-T's SPM. Cell also supports both horizontal and vertical DMA operations, but Cell's DMA operations are function-limited and not aware of network congestion. The main difference between our full/empty bit synchronization scheme and the previous works[15–16] is that our design is based on fast on-chip local memory, rather than off-chip memory. Moreover, our scheme can cooperate with DTA to facilitate on-chip data communication.

There are many works about the hardware supported mutual exclusion synchronization[16,35] and barrier synchronization[36–38]. These schemes are based on memory communication or network infrastructure. However, our proposed synchronization scheme is totally independent of memory hierarchy and network infrastructure.

7 Conclusion

Performance and programmability are often considered as two contrary sides of a parallel architecture. Godson-T shows a solution to achieving performance without losing programmability. The programmability of Godson-T is compatible with the popular conventional multithreading programming on cache hierarchy. Godson-T also provides the accessibility of on-chip resource and reasonable abstraction of micro-architecture for programmers to easily obtain high performance. We believe the design methodology used in Godson-T will contribute to the future many-core architecture designs.

References

[1] Asanovic K *et al.* The landscape of parallel computing research: A view from Berkeley. Technical Report No.UCB/EECS-2006-183, University of California, Berkeley, December 18, 2006.

[2] Lee E A. The problem with threads. *Computer*, 2006, 39(5): 33–42.

[3] Cantrill B, Bonwick J. Real-world concurrency. *ACM Queue*, 2008, 6(5): 16–25.

[4] Adve S V, Adve V S *et al.* Parallel computing research at Illinois: The UPCRC agenda. Technical Report, University of Illinois at Urbana-Champaign, November 2008.

[5] Yuan N, Yu L, Fan D. An efficient and flexible task management for many-core architectures. In *Proc. Workshop on Software and Hardware Challenges of Manycore Platforms*, in *Conjunction with the 35th International Symposium on Computer Architecture* (*ISCA-35*), Beijing, China, June 22–26, 2008, pp.1–17.

[6] Blumofe R D, Leiserson C E. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 1999, 46(5): 720–748.

[7] Palatin P, Lhuillier Y, Temam O. CAPSULE: Hardware-assisted parallel execution of component-based programs. In *Proc. the 39th Annual IEEE/ACM International Symposium on Micro-Architecture*, Washington, DC, USA: IEEE Computer Society, Dec. 9–13, 2006, pp.247–258.

[8] Villa O, Palermo G, Silvano C. Efficiency and scalability of barrier synchronization on NoC based many-core architecture. In *Proc. CASES 2008*, Atlanta, USA, Oct. 19–24, 2008, pp.81–90.

[9] Carlson W W, Draper J M *et al.* Introduction to UPC and language specification. Technical Report No. CCS-TR-99-157, University of California, Berkeley, 1999.

[10] Numrich R W, Reid J. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 1998, 17(2): 1–31.

[11] Yelick K, Semenzato L *et al.* Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 1998, 10(11-13): 825–836.

[12] Fatahalian K, Horn D R *et al.* Sequoia: Programming the memory hierarchy. In *Proc. the 2006 ACM/IEEE Conference on Supercomputing,* Tampa, Florida, Nov. 11–17, 2006, pp.83–95.

[13] Bikshandi G, Guo J *et al.* Programming for parallelism and locality with hierarchically tiled arrays. In *Proc. the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, USA, March 29–31, 2006, pp.48–57.

[14] Mellor-Crummey J M, Scott M L. Synchronization without contention. In *Proc. Architectural Support for Programming Languages and Operating Systems*, Santa Clara, USA, April 8–11, 1991, pp.269–278.

[15] Alverson R, Callahan D *et al.* The Tera computer system. In *Proc. the 4th Int. Conf. Supercomputing*, Amsterdam, The Netherlands, June 11–15, 1990, pp.1–6.

[16] Zhu W, Sreedhar V C *et al.* Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures. In *Proc. the 34th Annual International Symposium on Computer Architecture*, San Diego, USA, June 9–13, 2007, pp.35–45.

[17] Woo S C, Ohara M *et al.* The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. the 22nd Annual International Symposium on Computer Architecture*, Santa Margnerita Ligure, Italy, June 22–24, 1995, pp.24–36.

[18] Fu Y, Yang Q *et al.* Exploiting the kernel trick to correlate fragment ions for peptide identification via tandem mass spectrometry. *Bioinformatics*, 2004, 20(1): 1948–1954.

[19] Altschul S, Madden T, Schaffer A *et al.* Gapped Blast and Psi-Blast: A new generation of protein database search programs. *Nucleic Acids Research*, 1997, 25(17): 3389–3402.

[20] Kumar S, Jiang D *et al.* Evaluating synchronization on shared address space multiprocessors: Methodology and performance. *ACM SIGMETRICS Performance Evaluation Review* (*SIGMETRICS 1999*), 1999, 27(1): 23–34.

[21] Feo J. An analysis of the computational and parallel complexity of the Livermore loops. *Parallel Computing*, 1988, 7(2): 163–185.

[22] Yuan N, Zhou Y *et al.* High performance matrix multiplication on many cores. In *Proc. European Conference on Parallel and Distributed Computing* (*Euro-Par*), Delft, The Netherlands, Aug. 25–28, 2009, pp.948–959.

[23] Volkov V, Demmel J W. Benchmarking GPUs to tune dense linear algebra. In *Proc. 2008 ACM/IEEE Conf. Supercomputing* (*SC 2008*), Austin, USA, Now. 15–21, IEEE Press, 2008, pp.1–11.

[24] Chen L, Hu Z *et al.* Optimizing fast Fourier transform on a multi-core architecture. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, Long Beach, USA, March 26–30, 2007, pp.1–8.

[25] Hu Z, Cuvillo J *et al.* Optimization of dense matrix multiplication on IBM Cyclops-64: Challenges and experiences. In *Proc. Euro-Par* 2006, Dresden, Germany, August 28–September 1, pp.134–144.

[26] Govindaraju N K *et al.* High performance discrete Fourier transforms on graphics processors. In *Proc. the 2008 ACM/IEEE Conference on Supercomputing* (*SC2008*), Austin, USA, Nov. 15–21, 2008, pp.13–24.

[27] Williams S, Shalf J *et al.* The potential of the cell processor for scientific computing. In *Proc. CF'06*, Ischia, Italy, May 3–5, 2006, pp.9–20.

[28] Gao G R, Sarkar V. Location consistency — A new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 2000, 49(8): 798–813.

[29] Shen X *et al.* Commit-reconcile & fences (CRF): A new memory model for architects and compiler writers. In *Proc. the 26th Annual International Symposium on Computer Architecture*, Atlanta, USA, May 2–4, 1999, pp.150–161.

[30] Lftode L *et al.* Scope consistency: A bridge between release consistency and entry consistency. In *Proc. the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, June 24–26, 1996, pp.277–287.

[31] Ceze L, Tuck J *et al.* BulkSC: Bulk enforcement of sequential consistency. In *Proc. the 34th Annual International Symposium on Computer Architecture*, San Diego, USA, June 9–13, 2007, pp.278–289.

[32] Hofstee P. Power efficient architecture and the cell processor. In *Proc. HPCA-11*, San Francisco, USA, February 12–16, 2005, pp.258–262.

[33] Almasi G, Cascaval C *et al.* Dissecting cyclops: A detailed analysis of a multithreaded architecture. *ACM SIGARCH Computer Architecture News*, 2003, 31(1): 26–38.

[34] Lindholm E *et al.* NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 2008, 28(2): 39–55.

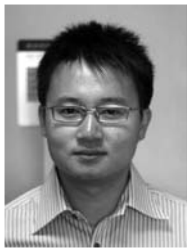
[35] Mellor-Crummey, J M, Scott M L. Synchronization without contention. In *Proc. Architectural Support for Programming Languages and Operating Systems*, Santa Clara, USA, April 8–11, 1991, pp.269–278.

[36] Keckler S W *et al.* Exploiting fine-grain thread level parallelism on the MIT multi-alu processor. In *Proc. the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, June 27–July 1, 1998, pp.306–317.

[37] Sampson J, Gonzalez R. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *Proc. the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Orlando, USA, Dec. 9–13, 2006, pp.235–246.

[38] Villa O *et al.* Efficiency and scalability of barrier synchronization on NoC based many-core architecture. In *Proc. CASES 2008*, Atlanta, USA, October 19–24, 2008, pp.81–90.

Dong-Rui Fan graduated from the Department of Mathematical Science at Beijing Jiaotong University with a Bachelor's degree in 2000, and he received the Ph.D. degree from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS) in 2005. Now, he is an associate researcher at ICT, a member of CCF and IEEE. He worked together with members of AMS (Advanced Micro-System) research group and designed the new processing models — Godson-X and Godson-T. Currently, His research interest focuses on many-core system, including the design of microarchitecture, parallel processing, and runtime system.



Nan Yuan graduated from the Department of Computer Science and Technology at Beijing University of Posts and Telecommunication with a Bachelor's degree in 2004, and he is currently a Ph.D. candidate of ICT, CAS. His current research interests include parallel architecture design and runtime system design.

Jun-Chao Zhang is currently an engineer at ICT, CAS. He received his Ph.D. degree in computer science from ICT, CAS in 2005 and his B.Eng. degree from Xi'an Jiaotong University in 1999. His research interests include computer architecture, parallel computing, compiler and parallel languages. He is an ACM member and CCF member.

Yong-Bin Zhou received his B.Eng. degree from University of Science and Technology of China (USTC). Currently, he is a Ph.D. candidate in computer science at ICT, CAS. His recent research topics include computer architecture and parallel computing.

Wei Lin received his B.Sc. degree from Tianjin University. Currently, he is a Ph.D. candidate in computer science at ICT, CAS. His research interests include computer architecture, parallel computing, and operating system.

Feng-Long Song graduated from the Department of Management and Economics at Shandong Normal University and received Master's degree in 2006. He is a Ph.D. candidate of ICT, CAS. His research interests focus on high performance computer architecture, on-chip memory hierarchy, and parallel computing.

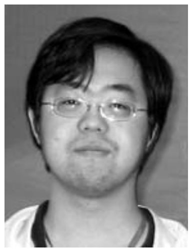
Xiao-Chun Ye received his B.Sc. degree from Beijing Normal University in 2004. Currently, he is a Ph.D. candidate in computer science at ICT, CAS. His recent research topics include computer architecture, parallel computing, and bioinformatics.

He Huang is a Ph.D. candidate at ICT, CAS. His research interests include processor micro-architecture, operating system and VLSI backend design.

Lei Yu is currently a Ph.D. candidate at ICT, CAS. His current research interests include computer architecture and parallel computing.

Guo-Ping Long is currently a Ph.D. candidate at ICT, CAS. His research interests include parallel programming, performance modeling and evaluation.

Hao Zhang is an assistant researcher at ICT, CAS. Zhang received the Ph.D. degree in computer science from ICT in 2008. His research interests include design, analysis, implementation, and benchmarking of processor architectures; switching and routing of on chip networks; and high throughput memory system.



Lei Liu received his B.Sc. degree from Peking University in 2004. Currently he is a Ph.D. candidate at ICT, CAS. His research topic is power management of many-core architecture.