Comput Sci Res Dev (2011) 26: 325–337
DOI 10.1007/s00450-011-0165-1

SPECIAL ISSUE PAPER

Experiments with the Fresh Breeze tree-based memory model

Jack B. Dennis · Guang R. Gao · Xiao X. Meng

Published online: 20 April 2011

Abstract The Fresh Breeze memory model and system architecture is proposed as an approach to achieving significant improvements in massively parallel computation by supporting fine-grain management of memory and processing resources and utilizing a global shared name space for all processors and computation tasks. Memory management and the scheduling of tasks are done by hardware realizations, eliminating nearly all operating system execution cycles for data access, task scheduling and security. In particular, the Fresh Breeze memory model uses trees of fixed-size chunks of memory to represent all data objects, which eliminates data consistency issues and simplifies memory management. Low-cost reference-count garbage collection is used to support modular programming in type-safe programming languages.

The main contributions of this paper are: (1) a program exection model for massively parallel computing as the Fresh Breeze application programming interface (API) comprising a radical memory model and a scheme for expressing concurrency; (2) an experimental implementation of the API through simulation using the FAST simulator of the IBM Cyclops 64 many-core chip; (3) simulation results that demonstrate that (a) fine-grain hardware-implemented resource management mechanisms can support massive parallelism and high processor utilization through the latency-hiding properties of multi-tasking; and (b) hardware implementation of a work stealing scheme incorporated in our simulation can effectively distribute tasks over the processors of a many-core parallel computer.

Keywords Memory models · Storage system · Massive parallelism · Concurrency model · System simulation

J.B. Dennis (✉)
Computer Science and Artificial Intelligence Laboratory, MIT, Room 32-G868, 32 Vassar Street, Cambridge, MA 02139, USA
e-mail: dennis@csail.mit.edu

G.R. Gao · X.X. Meng
Department of Electrical and Computer Engineering, University of Delaware, 140 Evans Hall, Newark, DE 19716, USA

G.R. Gao
e-mail: ggao@capsl.udel.edu

X.X. Meng
e-mail: meng@capsl.udel.edu

1 Introduction

The current crisis regarding choice of architecture and programming models in support of massively parallel computation demands new efforts toward understanding the possibilities and advantages of new programming models and hardware structures that support their efficient operation.

The Fresh Breeze memory model and system architecture [1, 2] is proposed in response to this demand and provides a system-wide one-level store supporting fine-grain resource management of processing and memory resources that is compliant with the capability model for implementing privacy and security [3–5].

In the Fresh Breeze vision, the entire memory hierarchy is treated as a unified one-level store, from processor cache memories through the main memory and on to the disk storage units. A single naming scheme is used throughout the hierarchy: a *handle* uniquely identifies a fixed-size *chunk* of program or data. Memory allocation and data transfer is performed entirely by hardware mechanisms so there is zero involvement of operating system software in data access and memory management. The handles of the Fresh Breeze memory model are equivalent to *capabilities* [3–6],

providing a basis for realizing advanced security and privacy properties in a Fresh Breeze system.

The Fresh Breeze vision also includes hardware implementation of activity scheduling, which is simplified by use of a memory model that provides a uniform view of memory throughout all jobs and processors of a massively parallel computer system. The combination of the chunk-based memory model and hardware for fine-grain processor switching provides an ability for modular composition of parallel programs well beyond what is possible with any existing computer system.

The main contributions of this paper are:

1. A program exection model for massively parallel computing as the Fresh Breeze application programming interface (API) comprising a radical memory model and a scheme for expressing concurrency;
2. An experimental implementation of the API through simulation using the FAST simulator of the IBM Cyclops 64 many-core chip;
3. Simulation results that demonstrate that (a) fine-grain hardware-implemented resource management mechanisms can support massive parallelism and high processor utilization through the latency-hiding properties of multi-tasking; and (b) hardware implementation of a work stealing scheme incorporated in our simulation can effectively distribute tasks over the processors of a many-core parallel computer.

Synopsis: In Sect. 2 the Fresh Breeze memory model is presented. Section 3 describes Fresh Breeze support for collections of parallel tasks. A vision of future computer system organization utilizing Fresh Breeze principles is provided in Sect. 4, and its application programming interface (API) discussed. Sections 5 through 7 describe the experimental implementation of the Fresh Breeze API using the Cyclops 64 simulation software. Section 8 presents results and a discussion of their significance. The paper concludes with paragraphs on future plans, related work, conclusions and acknowledgments.

2 The Fresh Breeze memory model

In the Fresh Breeze Memory Model [3, 20] information objects and data structures are represented using fixed size chunks, which are 128 bytes in the present design. Each chunk has a unique 64-bit identifier, its *handle*, equivalent to a capability, that serves to locate the chunk within the storage system, and is a globally valid means of reference to the chunk. Each chunk can hold up to 16 elements that are either 64-bit data values or handles of other chunks. A collection of chunks organized as a directed acyclic graph (DAG) can

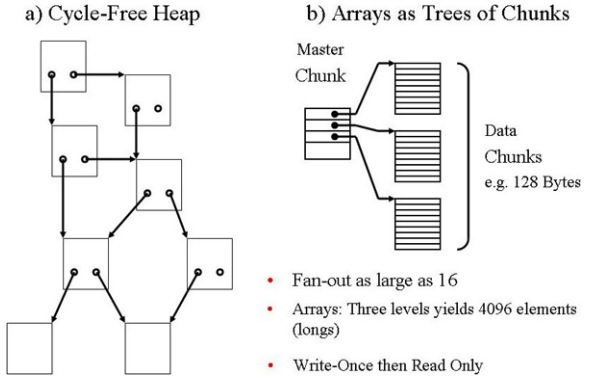


Fig. 1 Data objects as trees of chunks

represent structured information as illustrated in Fig. 1. For example, a three-level tree of chunks can represent an array of 4096 elements. Data objects and data structures may be represented by unbounded trees of chunks.

The Fresh Breeze memory model is a *write-once* model meaning that chunks may be created and written by a user of the memory model, but access to a chunk is not permitted for more than one computing activity (task) until it is rendered read-only. The life-cycle of a chunk may be summarized as follows: (1) A free chunk is acquired by a producer task from the memory system; (2) The chunk is then written and sealed by the producer task; (3) Once sealed, the chunk may be shared with consumer tasks; (4) When usage of the chunk becomes low, it will be evicted from higher levels of the memory hierarchy until it only resides in the lowest level; (5) It is deleted once no references to the chunk exist.

A major benefit of a write-once memory model for massively parallel computation is that cache memories may be used without coherence issues: Several computing tasks running in separate parts of a system may access data with no concern that it might be stale. Adopting the write-once property leads to a functional view of memory: A computing step involves accessing existing data values and creating fresh memory chunks to receive results. To work effectively, very efficient mechanisms for allocating memory and collecting chunks that no longer contain accessible data are required. Use of a fixed-size unit of memory allocation and the write-once principle makes this feasible. It also permits use of low-overhead reference counts to identify garbage chunks for reclaiming their memory.

The Fresh Breeze memory model provides a global addressing environment, a virtual one-level store, shared by all user jobs and all processors of a many-core computing system. It can extend to the entirety of online storage, replacing the separate access means for files and databases of conventional systems.

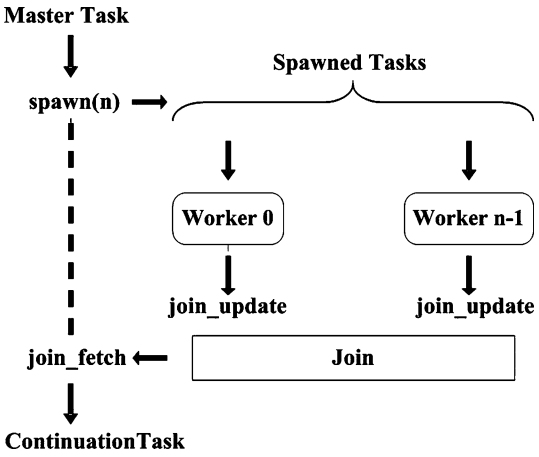


Fig. 2 Fresh Breeze parallelism using Spawn and Join

3 Fresh Breeze tasking

In Fresh Breeze tasking [7], the basic unit of parallelism is the *task*, which is the activity of performing a single execution of a function instantiation, corresponding typically to a single call of a Java method. The organization of multiple tasks is expressed in a way similar to the spawn/join model for parallel programming in Cilk [8].

As shown in Fig. 2, a master task may spawn one or more worker tasks executing independent instances of the same or different functions. Worker tasks may receive data objects (scalar values or handles) as arguments provided by the parent task, and each worker task contributes the results of its activity to a continuation task using a join mechanism [7]. Through repeated use of this scheme, a program can generate an arbitrary hierarchy of concurrent tasks corresponding to available parallelism in the computation being performed. The spawn/join mechanism is implemented by special machine level instructions of the Fresh Breeze API, providing a fine-grain parallel processing capability.

To illustrate, consider the dot product computation which is the focus of the experiments reported in this paper. The complete computation consists of constructing two vectors and then computing their dot product. Straightforward code for this computation may be written as follows:

```
vector BuildVector (long length, long seed) {
    long[] vector = new long[length];
    for (int i = 0; i < length; i++)
        vector [i] = generate (length, seed);
    return vector;
}
long DotProduct (
        long[] vector_a,
        long[] vector_b,
        long length) {
    long sum = 0;
    for (int i = 0; i < length; i++)
        sum += vector_a[i] * vector_b[i];
    return sum;
}
void main () {
    long length = N;
    long[] vector_a = BuildVector (length, seed_a);
    long[] vector_b = BuildVector (length, seed_b);
    long result = DotProduct (
            vector_a, vector_b, length);
}
```

For execution by a Fresh Breeze computer, this code will be compiled into machine code that uses the chunk-based memory model and instructions for spawning and joining tasks. A pseudo-code version of the Fresh Breeze machine code for the DotProduct method of the FunJava program given above follows. The handle data type is used for the capability codes of chunks.

```
long DotProductMain (
      handle vector_a,
      handle vector_b,
      long length) {
   // Calculate tree size
   long tree_size = ... ;
   DotProduct (
         vector_a, vector_b,
         length, tree_size);
   return result;
}
void DotProduct (
      handle vector_a,
      handle vector_b,
      long length),
      long tree_size) {
   chunk chunk_a = chunk_read (vector_a);
   chunk chunk_b = chunk_read (vector_b);
   if (tree_size > CHUNK_SIZE) {
      // Process internal nodes
      chunk join_ticket =
         join_init (count, DotProductDone);
      for (int ix = 0; ix < count; ix++) {
         // Calculate node size and subtree size
         node_size = ... ;
         tree_size = ... ;
         spawn_one (ix, DotProduct (
              chunk_a[ix], chunk_b[ix],
              size, tree_size) );
      }
      exit ();
   } else {
      // Process a leaf node
      long sum = 0;
      for (int ix = 0; ix < count, ix++ ) {
         sum += chunk_a[ix] * chunk_a[ix];
      }
      join_update (sum);
   }
}
```

```
void DotProductDone (int count) {
   handle data = join_fetch ();
   chunk join_data = chunk_read (data);
   long sum = 0;
   for (int ix = 0; ix < count; ix++) {
      sum += join_data [ix];
   }
   join_update (sum);
}
```

The phrases **spawn_init**, **spawn_one**, **join_fetch** and **join_update** are the special Fresh Breeze instructions to support concurrency. The instruction **spawn_init** creates a *join ticket* that holds a *join counter* and the name of a function that defines the task for execution by a worker; **spawn_one** creates a new task for execution with the specified index; **join_ fetch** is used after a join chunk has been filled by worker tasks using the **join_update** instruction. It provides the handle of the (now filled) join data chunk. Execution of a **join_update** causes a worker task to quit, turning the processor over to other tasks.

4 The Fresh Breeze vision and its API

The envisioned organization of a Fresh Breeze computer system is illustrated in Fig. 3. The main components are a multitude of many-core processing chips coupled to a multi-level off-chip storage system. Each many-core processing chip uses processor cores similar to those of the Cyclops 64 chip [10], coupled to the top levels of a memory hierarchy consisting of L1 instruction and data cache memories at each processor, and a shared on-chip L2 cache.

Many-Core Chip. The distinguishing features of the many-core processor chip are:

– The cache memories are organized around chunks instead of typical cache lines to benefit from the locality provided by the chunk-based memory model.
– There is no TLB because capabilities (handles) are held in chunks and in processor registers.
– Processor registers are tagged to flag those holding handles of chunks.
– A new load/store unit will be used to support creating (writing) and reading of memory chunks.
– A hardware task scheduler that implements fast switching among active tasks and a task stealing scheme for load distribution.

Storage System. The Storage System is a hierarchical memory system in which the higher levels (closer to the processors) cache data chunks actively involved in on-going computations [11].

In Fig. 3, two off-chip storage levels are illustrated for simplicity; the architecture may be extended to further levels as demanded by the device technology available and the storage capacity required by a system.

There is no relation of the 64-bit number that encodes the handle of a chunk, to the physical location where the chunk is held in the Storage System. This property permits new data to be stored in proximity to the location in the system where they are generated. Hardware-supported associative search is used to map handles to the physical locations where the designated chunks are to be found.

Another function performed by the Storage System is to supply codes of free handles to the processing chips for assignment to newly created chunks. A data structure is maintained, that keeps a record of available codes. Handle codes are assigned from the free pool and returned to the pool when a reference count shows they are no longer in use.

The principal components at each level of the Storage System are multiple storage devices to hold data chunks, and an associative directory for mapping chunk handles to locations where chunks reside. At the lowest level, the set of storage devices is sufficient to hold all data in the computer system, and is partitioned according to a division of the set of possible handle codes. At each level each directory must

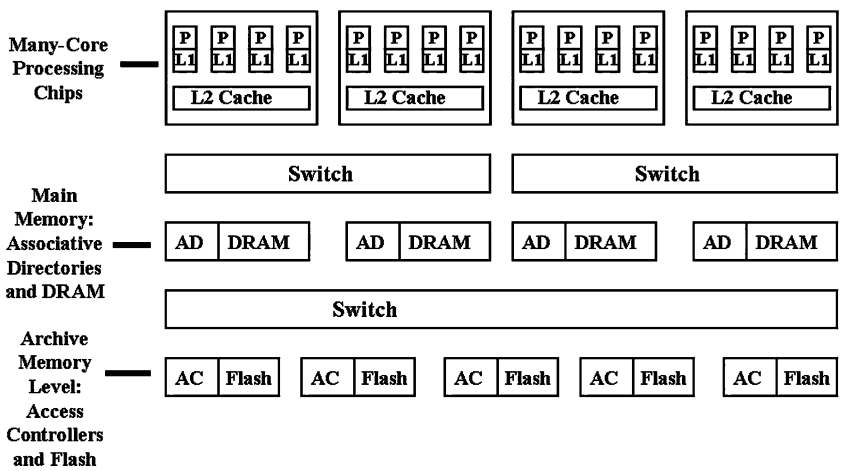


Fig. 3 Vision of a massively parallel Fresh Breeze system

map handles to a sufficiently large physical space to accommodate all data in its part, and its implementation must be able to handle the anticipated traffic.

4.1 The Fresh Breeze API

The Fresh Breeze API is the interface through which users express computations for execution by a Fresh Breeze system. In the Fresh Breeze vision, the API is a set of machine instructions that invoke hardware to perform various actions on behalf of a user program. This paper is concerned with two sets of instructions: those that express memory actions and those concerned with concurrent tasks. The instructions of interest in the present paper are:

Memory instructions

chunk_write (private task memory location);
Obtain a fresh chunk and write into it the data at the specified location.
chunk_read (handle);
Read into task private memory the chunk specified by handle.

Tasking instructions

join_init (count, function identifier);
Return a join ticket with the specified count using the specified function as the continuation task.
spawn_one (idx, function)
Spawn a new task with the specified index to execute the specified function.
exit ();
Terminate the task.

join_update (value);
Store the result value of a worker task in the join data chunk.
join_fetch ();
Get the handle of the join data chunk for the continuation task.

For simplicity, instructions for ensuring correct operation of reference-count garbage collection have been omitted. Also, the intention is to include additional instructions to support further functions such as implementing independent user jobs, secure sharing of objects, resource quotas and exception events.

For modeling a simplified form of the envisioned system, the above API has been implemented as a set of runtime routines for the Cyclops multi-core chip.

5 Simulation

For evaluating the proposed system structure and API we have chosen to employ a unique facility available at the University of Delaware. This is the FAST simulation tool developed built by a collaboration of IBM and E.T. International, for testing and evaluating the IBM Cyclops 64 many-core chip [10]. This chip contains 80 processing assemblies, each consisting of two independent Thread Units (TUs) sharing a 64-bit floating point unit. Each TU has an associated 30 KB block of SRAM. There are several instruction cache memories, each serving a group of ten TUs. The chip incorporates a cross-bar switching network that interconnects all 160 TUs, allowing each TU to access the SRAM of any other TU. The TUs have access to 1 Gb of off-chip DRAM memory through four additional ports of the X-bar network.

In our Fresh Breeze simulation, 40 thread units (labeled EUs and TSUs in Fig. 4) can execute 40 concurrent application tasks and simulate the L1 Cache units; most of the remaining 120 thread units are used to implement a simulation of the Shared Memory. Runtime software has been written to schedule user tasks on the EUs and to implement the Storage System simulation. Test programs are written in C and compiled by the Cyclops C compiler.

In the future, we expect Fresh Breeze programs to be written in a high level language related to Java and compiled to the Fresh Breeze API.

To utilize this simulation tool, a scheme was adopted for modeling each component of the system to be modeled (Fig. 3) using resources available in the Cyclops chip. Keep in mind that many Cyclops processing cores (Thread Units) are used to model memory components of the modeled system and do not correspond to the envisioned Fresh Breeze processor cores of the modeled system.

6 Scheduling and work stealing

The Fresh Breeze simulation models a hardware scheduling mechanism in each of the EUs. The elements of this mechanism are the Active Task List (ATL) and the Pending Task Queue (PTQ). The ATL contains an entry for each of several tasks that the EU switches among when a task in execution becomes blocked (usually due to a **chunk_read** instruction). An entry in the ATL holds the complete processor state for resuming the task when the reason for being blocked is resolved. (A blocked task is never resumed on another processor; it runs on its assigned processor until the work of the task is complete.)

The PTQ is a queue of tasks generated by Spawn instructions, that are available for execution. An entry in the PTQ just contains: (1) the address of the function to be applied by the new task, (2) the handle of an argument structure (chunk) containing argument values for use by the new task, and (3) the handle of the **join_ticket** used by the new task to record its result. The PTQ entry does not include any processor register contents because a new task is assumed to start

fresh and not depend on any register contents; The program counter is implicitly set to zero (indicating the first instruction of the method for the spawned task). Any application processor can perform any pending task just by loading the contents of a PTQ entry, a consequence of the global validity of handles and their power to provide access to arbitrarily large data objects.

In the experiments (Sect. 8), the ATL in each EU has five entries and the PTQ has 64 entries. The chip area required for the ATL and PTQ would be a small fraction of the silicon area of a processor.

Actions performed by an EU to simulate a Fresh Breeze Processor are:

1. Execute a task from the ATL.
2. Perform a storage system **chunk_read** or **chunk_write** instruction issued by a task.
3. On a **join_init** instruction, initialize a **join_ ticket** chunk.
4. On a **spawn_one** instruction, add an entry to the PTQ and continue task execution.
5. On a **task_exit** instruction, delete the task from the ATL and select a task from the PTQ to make active.

Additional actions are used for implementing the join mechanism:

1. On a **join_update** instruction executed by a worker task, write the result value (scalar or handle) into the **join_data** chunk, update the join count, and terminate the task.
2. On a **join_fetch** instruction executed by a continuation task, return the handle of the **join_data** chunk to the continuation task and mark the **join_ticket** chunk as garbage.

The scheduling mechanism described above does not provide for distributing spawned tasks over the large number of processors of a massively parallel system. The current Fresh Breeze simulation includes a work stealing scheme that is a variation on work stealing in Cilk [15]. It is designed to model a low-cost hardware mechanism.

Task stealing is used by a processor to maintain the number of entries in its PTQ between two limits; if the number of entries is less than the lower limit, this processor is not willing to give away any of its tasks; if the upper limit is exceeded, the processor will not try to fetch a task from the global deferred task queue (see below).

Task stealing uses two tables located in memory globally accessible to all processors of a domain (the 40 EUs in the present simulations). The mechanism to be described may be extended hierarchically as needed in a massively multicore system. The two tables are managed by a reserved Steal Daemon processor (thread unit SU) in the simulation. The work of the Steal Daemon is sufficiently simple that it could readily be implemented in hardware in the envisioned Fresh Breeze system.

One table, the *Steal List*, contains an entry for each processor of its domain/cluster. The entry specifies the identity (processor number) of some processor of the domain that has tasks available for stealing. The entry is undefined if the Steal Daemon judges that stealing has no benefit for the task processor at this time. A processor accesses its entry in the table using a read/replace memory operation that sets the entry to undefined and provides the identity of a processor with available tasks in its PTQ; the processor then removes the task from the target processor's PTQ. If stealing fails, the requesting processor will do other work and make a new request after a preset time interval.

The second table, the *Load Table*, is provided so the Steal Deamon can know the load status of each processor of the domain. It contains simply a boolean value maintained by each processor to indicate whether or not the processor's PTQ has more entries than its lower limit. The steal Daemon maintains the Steal Table continuously based on its knowledge of the load on each processor. The rule is: initialize all entries of the Steal Table to undefined; then, for each processor, if its entry is undefined, set it to the identifier of some processor with more than the lower limit of entries in its PTQ.

An additional problem arises when so many tasks are generated that there is insufficient room in the PTQs of all processors. The scheduler must somehow retain records of them so they may be scheduled at a future time when the overload condition has subsided. This is done in the present simulations by means of a global deferred task queue held in the memory system.

7 System modeling with simulation

In this section the relation between the system being modeled and the simulation of it is discussed. First, the system studied by our modeling experiments is described. It is limited to a two-level memory hierarchy by the design of the present simulation capability. Extension to a more extensive memory hierarchy is planned. Then the issues in relating actions in the modeled system to simulation events are discussed, together with the solution adopted to obtain accurate modeling of the timing of actions in the modeled system

7.1 The system modeled

Figure 4 shows the system modeled in our simulations which has two memory levels. We take the upper level as modeling an L1 cache unit which is private to each processor. The lower memory level is a shared memory, we will call the Shared Memory, that may be regarded either as a shared L2 cache accessible to all processors, or as a main memory level. The two choices differ in their access times,

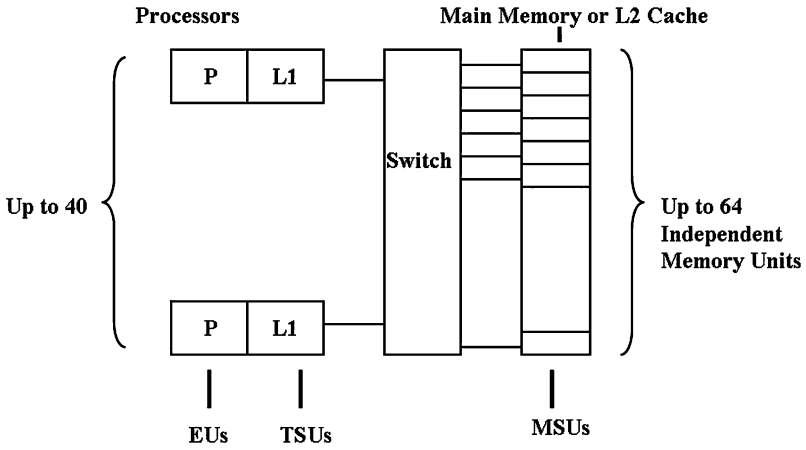


Fig. 4 Fresh Breeze system model with two memory levels

so we use the Shared Memory access time as the principal parameter varied in our experiments. In both levels, memory is allocated in units of one chunk. Reference count garbage collection is used to reclaim memory chunks no longer accessible. The hardware implementation of garbage collection is not expected to have a significant impact on the performance results reported below.

For the present experiments, only data objects are held as trees of chunks. The program instructions are held just as code is held for normal Cyclops 64 simulation. This should not affect our experiments other than by Cyclops instruction cache misses as discussed in Sect. 8.

We assume the upper memory level (L1) may be accessed in two clock cycles and that reading one chunk of data into processor registers takes 16 clocks. Since this combination always occurs together in the Dot Product test program, we treat the pair as a single action. This permits use of less padding to equalize the times per clock of all actions and provides a more practical duration of simulation runs.

The upper memory level is operated as a fully associative cache where the cache tag is the handle of the referenced chunk. Each L1 cache holds 128 chunks or 16 K bytes of data.

For specificity we chose the system clock rate to be 500 MHz, a common choice for many-core chips such as the Cyclops 64.

7.2 Events in simulation versus actions in an implementation

The simulation code consists of routines that model various actions in the modeled system. Unfortunately, there is a large disparity among the numbers of Cyclops chip cycles required for the various actions. Table 1 shows the several actions exercised by the Dot Product test program. For each action the table shows the clock cycles assumed needed in the modeled system action and the simulation cycles used by the corresponding simulation routine. For our experiments, we made the simulation time exactly proportional to the modeled system time by choosing a ratio of simulation cycles to system cycles and adding "padding" cycle to each simulation action routine to provide a uniform ratio of 160. In this way, cycle-accurate modeling of the subject system is achieved. The padding cycles and total simulation cycles for each action are shown in columns four and five of Table 1.

Table 1 Cycle-accurate modeling of the system

Action	System cycles	Simulation cycles	Padding	Total
Task startup	4	262	378	640
Task compute	32	376	4744	5120
Task switch save/restore	16	262	2298	2560
L1 cache access	32	3047	2073	5120

All of the actions listed in Table 1 are simulated by routines that run on the EUs of the simulated Cyclops 64 chip. The simulation also includes actions that model the handling of access requests to Shared Memory (instructions **chunk_read** and **chunk_write**) that get queued for services at the MSUs.

For each unit of the modelled Shared Memory the simulation routine run on the corresponding MSU maintains a queue of access requests. In the modeled system a Shared Memory access request must traverse the switch, with arbitration, and then wait at the memory unit until it can be served. Then the data transfer is performed in 16 cycles. The switch, arbitration, and queuing delays make up the Access Time, which is a parameter of the simulation runs. Instead of padding each simulation routine to model the delay, time stamps are used in the operation of each request queue so that many requests may be entered while each requested data transfer is not performed until the specified number of cycles

have elapsed. The actions simulated by the MSUs are sufficiently simple that we are assured they will be completed without slowing the overall simulation.

Simulation experiments were conducted for two scenarios: In the first scenario, the Shared Memory models a shared L2 cache memory. For this case, access times are relatively short and performing **chunk_read** operations without blocking the processor is the preferred mode of operation. For these tests the actions of Task Save and Task Restore do not apply. In the second scenario, the Shared Memory System models a main memory with longer access times. For the modeled Fresh Breeze architecture, task switching times are sufficiently short that it is beneficial to use a *blocking read* wherein the processor is switched to a different task while a **chunk_read** operation is performed. For these experiments the Task Save and Task Restore actions model the retention of processor register state across read operations.

8 Experiments

In our simulation runs, the Dot Product computation was run for several vector lengths and various values of parameters of the modeled system.

To begin, Table 2 shows the numbers of task executions needed for processing leaf chunks and non-leaf chunks of tree representations of the vectors. Since 16 multiplies and 15 adds are performed in processing a leaf chunk and 15 adds are performed for each non-leaf chunk, the totals of adds and multiplies are readily calculated.

First presented are basic performance measures where performance is presented as the average number of cycles per task over all tasks executed in a simulation run. The charts show the performance for three vector lengths and various Shared Memory access times for the two cases of interest. In Fig. 5 reads are non-blocking, modeling behavior of an L2 shared cache; In Fig. 6 reads are blocking, with suspension of the task and swapping processor state to run an alternative task. This models a main memory where the fine-grain task management of the Fresh Breeze architecture serves to provide help with latency tolerance, even for typical main memory access times. In all of these runs a system having 40 processors and 64 independent shared memory units was simulated.

Table 2 Number of task executions and operations

Vector length	Leaf tasks	Non-leaf tasks	Total tasks	Adds or multiplies
16^3	256	17	273	4096
16^4	4096	273	4369	65536
16^5	65536	4369	69895	1048576

The best performance shown in these runs achieves an average around 250 cycles per task for the non-blocking case and 350 cycles in the blocking case. Using the numbers of leaf and non-leaf tasks from Table 2 and the corresponding counts of adds and multiplies, the number of operations per task for vectors of length 16^5 is 30.0. For a processor operating at a 500 MHz clock rate, this corresponds to a performance of $(30*500)/250 = 60$ million operations per second per processor or 2400 MOPS for the set of 40 processors.

In addition to average performance, the simulations have demonstrated the effectiveness of hardware-supported work stealing. Figure 7 shows how well the task processing load is distributed over the 40 processors for the Main Memory model for 100 cycles access time. Similar results were obtained for the L2 Cache simulations, although load distribution was slightly less effective for vectors of length 16^4.

8.1 Discussion

For the processor characteristics chosen for this study the maximum possible performance for the Dot Product computation for a 16-element vector is determined by the 32 cycles to execute 32 pipelined arithmetic operations and 32 cycles to access vector elements from top-level cache or $(32*500)/64 = 250$ MOPS. The experiments show that the Fresh Breeze architecture is able to achieve about 25 percent of this maximum. This is satisfying as memory and storage management functions are both performed entirely by the system, with no involvement of application programmer or a compiler. Without the degradation of simulation performance due to instruction cache misses, we believe at least twice the observed performance would be achieved. Given that the processor utilization data show processors are idle for very little time with long vectors, they must be performing the actions listed in Table 1. Future work using event-driven simulation is expected to provide more reliable results.

8.2 Work stealing

A high processor utilization requires that the tree of parallel tasks be distributed over the available processors as quickly as possible. Under the modeled system structure shown in Fig. 4, all of the shared storage units are equally accessible to all the processors. It makes no difference which processor gets to run any particular task. Under these conditions, the goal of scheduling and task distribution is to ensure that if there is a free processor and work to be done, the processor gets some work assigned. The mechanism employed in the system modeled in these experiments has been shown to be effective at this job.

However, for much larger systems it becomes important to recognize the non-uniform access characteristics of prac-

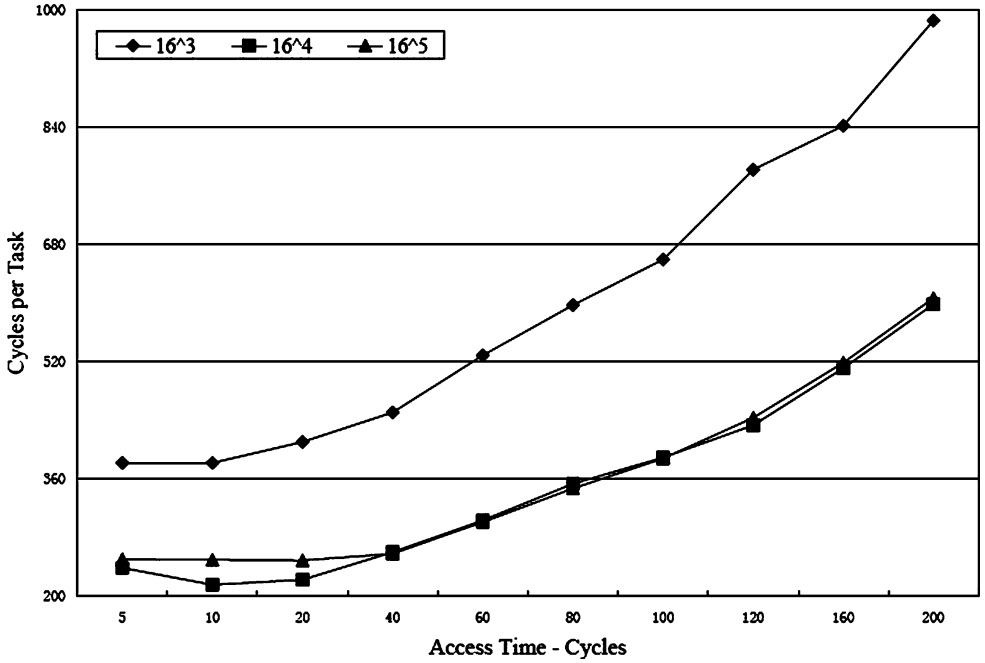


Fig. 5 Non-blocking read scenario: system cycles per task

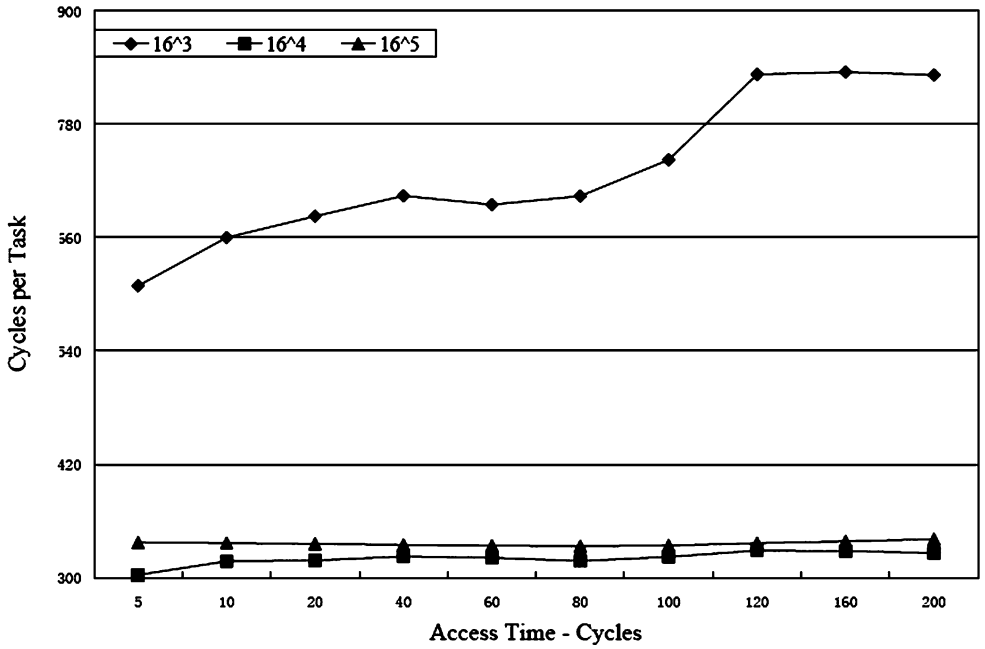


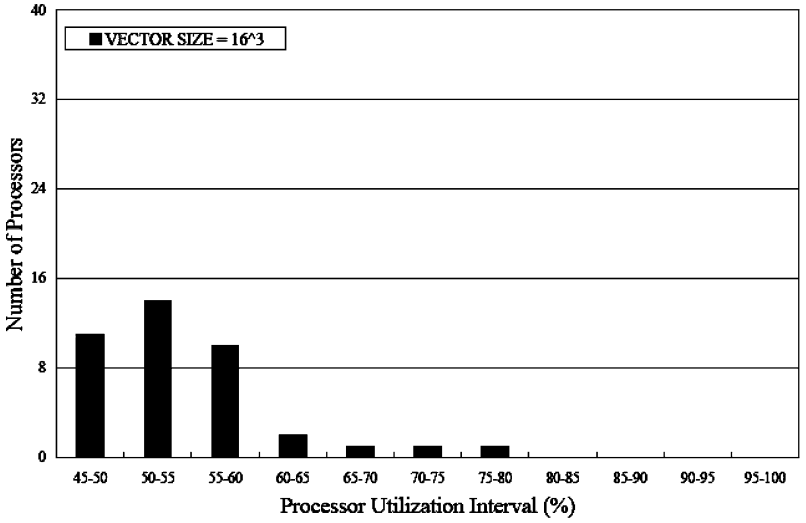
Fig. 6 Blocking read scenario: system cycles per task

tical scalable architectures for memory hierarchies. It follows that the locations of data structures must be considered in the design of any scalable, general task distribution scheme. This is expected to be a challenge for our continuing research.

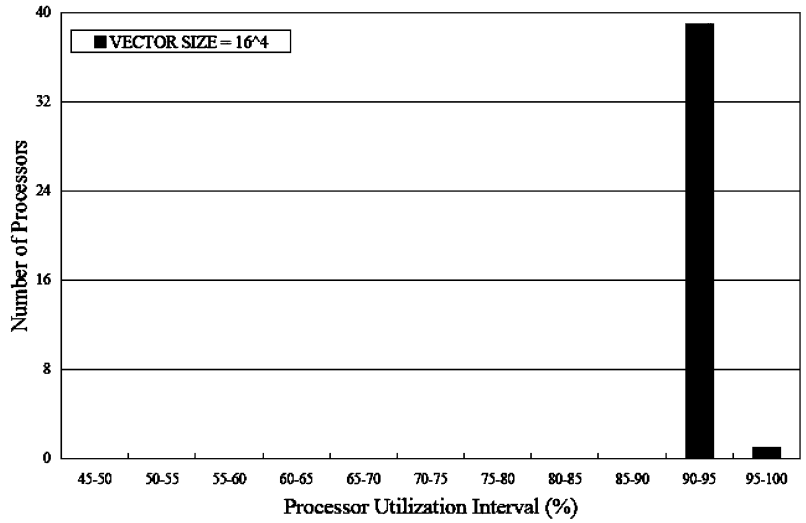
8.3 Caching

The traditional role of cache memories in computer systems has been to reduce the idle time of processors by exploiting temporal and spatial locality. However, the size of the L1 cache played no role in these simulation results. Essentially all memory references in runs of Dot Product resulted in data transfers from the Shared Memory. This did not result in a big performance problem because of the inherent spatial locality of data residing in one memory chunk: a cache miss on a **chunk_read** instruction causes transfer of the entire chunk and furthr accesses proceed at the L1 cache rate.

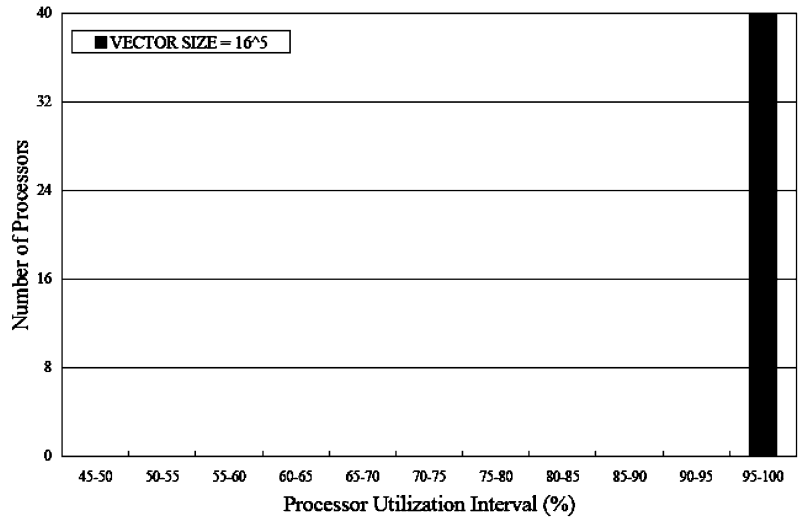
Further system design study may exploit another locality benefit of the tree-structured data model: if a node is accessed, it is likely that its children will also be accessed. This



(a)



(b)



(c)

Fig. 7 Load distribution performance of work stealing for Main Memory

suggests an implementation in which the system automatically fetches the child chunks of a node to some memory level when a request is received at that level for access to the node.

The Dot Product test computation involved zero reuse of data. This is not characteristic of most computations, for example, matrix multiplication and other examples which are being studied for the Fresh Breeze architecture. In general, the cache mechanism will likely be important to overall performance in future Fresh Breeze designs.

8.4 Excess parallelism

The fine-grained hardware-implemented scheduling of tasks permits a user computation to offer much more parallelism than can be actually exploited at any time by the system. This generates a need for either "throttling" the generation of tasks or providing a way for the system to remember the tasks that need to be taken up when resources (processors) become free.

In our experimental simulation, we chose to have each processor maintain a 64-position queue of pending tasks. With this choice the computation of the dot product for vectors of length 16^5 generated 209,716 tasks for the main memory model with 100 cycles of access latency (which is a typical value for the current main memory technology). All but 412 of these excess tasks were never sent to the deferred pool but were either taken from the pending list by the local processor, or were stolen from the pending list of another processor. Thus it seems that managing deferred tasks is manageable with suitable fine-grain hardware scheduling support.

9 Future plans

To further explore and demonstrate Fresh Breeze principles two lines of work will be followed.

First, our simulation facility must be augmented to encompass a more complete model of a realistic Fresh Breeze computer system (as illustrated by Fig. 3). In particular, a memory hierarchy of at least three levels is needed to show the power of the memory model to replace conventional file storage media. For this work, it is expected that event-driven simulation will be used to achieve greater efficiency in accurate modeling of a complex asynchronous system such as the Fresh Breeze vision. Later, it is hoped that a demonstration system including a large capacity flash memory level can be built using FPGA technology.

The second direction is to expand testing and evaluation to representative computations from a variety of application areas. It is expected that this will include stream processing and transaction-oriented applications as well as more ambitious codes for scientific computation. Codes for matrix multiplication, the FFT, and solution of linear equation systems are under development. Applicability of the trees-of-chunks model to computations on large graphs will be tested. In support of code development for testing, and eventual complete application codes, completion of usable compilers for FunJava [9] and Habanero Java [28] is anticipated.

10 Related work

The idea of building a computer system with unique handles for all data objects is central to the capability concept. It is the logical extension of virtual memory ideas embodied in Multics [13], and the successful commercial implementation in IBM AS/400 systems [14]. A software implementation of capabilities is available [6] and its successor Coyotos is under development. However, the software implementations do not have the tight security features of hardware-based capabilities. Use of the write-once storage concept in a parallel programming model was proposed by Dennis in [2].

During the past two decades, techniques for dynamic load balancing have been studied extensively in the context of several multithreading implementations. These include Cilk [8, 15], EARTH [16, 17] and the scheduling of parcels in HTMT [18]. The Rice University proposal for the HPC language Habanero Java includes the idea of *place tree hierarchies* as a means to offer programmers a range of options from fully specifying the mapping of parallel tasks to processors, to granting the system the responsibility of making assignments. This work is a revision of the X10 programming language, which uses the asynch/finish concurrency control primitives [19–22]. Related work appears in the HPC language Cascade [23].

In contrast to these software approaches, the Japanese Sigma 1 data flow computer included an interprocessor network that automatically routes remote function invocations to lightly loaded processors [24]. The work stealing technique proposed here for implementing the Fresh Breeze vision may be regarded as an implementation of Cilk ideas using principles similar to those of the Sigma 1.

Tools for conducting system evaluation through simulation and emulation is an area of active work [25, 26]. The RAMP project [27] at Berkeley is a FPGA-based many core emulation platform. In employs Xilinx Vertex-II Pro FPGAs on 16-21 BEE2 boards to implement a many core system composed of 1000 plus processor cores. The purpose of the RAMP project is to explore the architecture design space for future many-core computer architecture and enable early software development and debugging. It is intended to define and create the next generation tools

for computer architecture and computer system research. In contrast, the FAST simulation tool used in this paper is an industrial-strength system that can simulate the entire logic of the IBM Cyclops-64 chip with its 160 cores [12]. An implementation of a system emulation facility equivalent to the FAST simulator has been constructed using FPGA devices and is used for validation of both architecture and system software implementation for the Cyclops 64 device.

11 Conclusion

The work reported here has suggested the merits of a new memory model using trees of fixed size memory chunks to represent all data objects. Furthermore, the advantages of hardware implementation of scheduling and load distribution functions have been demonstrated, albeit in a limited scenario. Further work is needed to extend the system model and to study its performance for a variety of applications.

The Fresh Breeze architecture is an attractive basis for building future multiuser computer system with excellent security and protection properties by virtue of the equivalence of handles of objects with capabilities.

Further exploration of novel approaches to the architecture of highly parallel systems seems eminently justified.

Acknowledgements The authors thank the National Science foundation for funding this work under Grant CCF-0937907. This research was also partially supported by NFS Grants (CCF-0925863, OCI-0904534, CCF-0833122, CNS-0720531), and other government sponsors. In addition, we acknowledge our appreciation of the collaboration of IBM, University of Delaware and ET International that led to development of the FAST simulation software used in our experiments.

We thank the members of CAPSL group at University of Delaware for providing a stimulating research environment and inspiration. In particular, we also thank Elkin Garcia for careful proof reading and editing the entire paper and to make sure all figures and tables are included correctly, and Joshua Suetterlein for providing a number useful comments on the presentation as well as corrections.

References

1. Dennis JB (1997) A parallel program execution model supporting modular software construction. In: Massively parallel programming models. IEEE Comput Soc, Los Alamitos, pp 50–60
2. Dennis JB (2003) Fresh breeze: a multiprocessor chip architecture guided by modular programming principles. SIGARCH Comput Archit News 31(1):7–15
3. Dennis JB, Horn ECV (1966) Programming semantics for multiprogrammed computations. Commun ACM, 9, Feb 1966
4. Levy H (1984) Capability-based computer systems. Butterworth-Heinemann, Stoneham-London
5. Wilkes MV (1979) The Cambridge CAP computer and its operating system (Operating and programming systems series). Operating and programming systems series. North-Holland, Amsterdam
6. Shapiro JS, Smith JM, Farber DJ (1999) Eros: a fast capability system. In: Proceedings of the seventeenth ACM symposium on operating systems principles, SOSP'99. ACM, New York, pp 170–185
7. Dennis JB (2006) The Fresh Breeze model of thread execution. In: Workshop on programming models for ubiquitous parallelism. IEEE Comput Soc, Los Alamitos. Published with PACT-2006
8. Frigo M, Leiserson CE, Randall KH (1998) The implementation of the Cilk-5 multithreaded language. ACM SIGPLAN Not 33:212–223
9. Ginzburg I (2007) Compiling array computations for the Fresh Breeze parallel processor. Thesis for the Master of Engineering degree, MIT Department of Electrical Engineering and Computer Science, May 2007
10. del Cuvillo J, Zhu W, Hu Z, Gao GR (2005) Tiny threads: a thread virtual machine for the Cyclops 64 cellular architecture. In: International parallel and distributed processing symposium. IEEE Comput Soc, Los Alamitos, p 265
11. Schmidt B (2008) A shared memory system for Fresh Breeze. Master's thesis, MIT Department of Electrical Engineering and Computer Science, May 2008
12. del Cuvillo J, Zhu W, Hu Z, Gao GR (2005) FAST: a functionally accurate simulation toolset for the Cyclops 64 cellular architecture
13. Bensoussan A, Clingen CT, Daley RC (1969) The Multics virtual memory. In: Proceedings of the second symposium on operating systems principles. ACM, New York, pp 30–42
14. Soltis FG (1996) Inside the AS/400. Duke Press, Loveland
15. Vee V-Y, Hsu W-J (1999) Applying Cilk in provably efficient task scheduling. Comput J 42:699–712
16. Theobald KB (1999) EARTH: an efficient architecture for running threads. PhD thesis, University of Delaware, May 1999
17. Hum HHJ, Maquelin O, Theobald KB, Tian X, Tang X, Gao GR (1995) A design study of the EARTH multiprocessor. In: Conference on parallel architectures and compilation techniques, PACT. IEEE Comput Soc, Los Alamitos, pp 59–68
18. Theobald KB, Gao GR, Sterling TL (1999) Superconducting processors for HTMT: Issues and challenges. In: ACM'87: the 7th symp on the frontiers of massively parallel computation: today and tomorrow. ACM, New York, pp 260–267
19. Charles P, Grotho C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, von Praun C, Sarkar V (2005) X10: an object-oriented approach to non-uniform cluster computing. In: 2005 conference on objectoriented programming. ACM, New York, pp 519–538
20. Sarkar V, Hennessy J (1986) Compile-time partitioning and scheduling of parallel programs. In: 86 symposium on compiler construction, SIGPLAN. ACM, New York, pp 17–26
21. Shirako J, Peixotto D, Sarkar V, Scherer W (2008) Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In: Twenty-second international conference on supercomputing. IEEE Comput Soc, Los Alamitos
22. Guo Y, Barik R, Raman R, Sarkar V (2009) Work-first and help-first scheduling policies for async-finish task parallelism. In: International parallel and distributed processing symposium, IPDPS. IEEE Comput Soc, Los Alamitos
23. Callahan D, Chamberlain BL, Zima HP (2004) The Cascade high productivity language. In: Ninth international workshop on high-level parallel programming models and supportive environments
24. Yuba T, Hiraki K, Shimada T, Sekiguchi S, Nishida K (1987) The Sigma-1 dataflow computer. In: ACM'87: proceedings of the 1987 fall joint computer conference on exploring technology: today and tomorrow. IEEE Comput Soc, Los Alamitos, pp 578–585

25. Darringer J, Davidson E, Hathaway D, Koenemann B, Lavin M, Morrell J, Rahmat K, Roesner W, Schanzenbach E, Tellez G, Trevillyan L (2000) EDA in IBM: past present, and future. IEEE Trans Comput-Aided Des Integr Circuits Syst 19:1476–1497
26. Dubois M, Jeong J, Song Y, Moga A (1998) Rapid hardware prototyping on RPM-2. IEEE Des Test Comput, pp 112–118
27. Wawrzynek J, Patterson D, Oskin M, Lu S-L, Kozyrakis C, Hoe J, Chiou D, Asanovic K (2007) RAMP: research accelerator for multiple processors. IEEE MICRO 27:46–57
28. Cavé V, Budimlić Z, Sarkar V (2010) Comparing the usability of library vs. language approaches to task parallelism. ACM PLATEAU'10, evaluation and usability of programming languages and tools, pp 9.1–9.6

Jack B. Dennis is Professor of Computer Science and Engineering, Emeritus, at MIT. Prof. Dennis received his education at MIT, completing the doctorate in Electrical Engineering in 1958. He joined the MIT faculty in the Department of Electrical Engineering and was appointed full professor in 1969. As leader of the Computation Structures Group, MIT Laboratory for Computer Science, from 1963 through 1985, Professor Dennis developed aspects of computer architecture and programming languages based on dataflow models of program representation and execution. These developments led to dataflow projects at many universities and research institutes around the world, and won the ACM/IEEE Eckert-Mauchly Award for Professor Dennis in 1984. In 1994 he was inducted as a Fellow of the ACM and elected member of the NAE in 2008. Since 1987 he has been working as an independent consultant and research scientist on projects in parallel computer hardware and software.

Guang R. Gao is Endowed Distinguished Professor in the Department of Electrical and Computer Engineering at the University of Delaware. Dr. Gao received both his Masters and Ph.D. degrees in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology, in 1982 and 1986, respectively. He is the first person from China to receive computer science graduate degrees from MIT. At University of Delaware, Prof. Gao founded and directed the Computer Architecture and Parallel Systems Lab (CAPSL). Dr. Gao is a Senior Member of the IEEE and has been elected Fellow of both ACM and IEEE. Dr. Gao's main research interest is in high-performance computer systems: architecture, programming models, compilers, system software, and applications. Gao is a founder and Chairman of ET International Inc., a spin off from University of Delaware in 2000.

Xiao X. Meng is a post-doc researcher in the *Department of Electrical and Computer Engineering* at *University of Delaware*, beginning November, 2009. His research interests include parallel computing architecture, large-scale network storage systems, file systems and storage virtualization. Dr. Meng earned the Ph.D. degree in computer engineering from the *Institute of Computing Technology*, *Chinese Academy of Sciences* (*ICT*, *CAS*) in 2009, and received his bachelor degree in computer science from *Nanjing University* in 2002.