**ISC manuscript No.**
(will be inserted by the editor)

# Experiments with the Fresh Breeze Tree-Based Memory Model

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

Received: date / Accepted: date

**Abstract** The Fresh Breeze memory model and system architecture is proposed as an approach to achieving significant improvements by supporting fine-grain management of memory and processing resources and utilizing a global shared name space for all processors and computation tasks. Scheduling of tasks and storage allocation 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.

The main contribution of this paper includes: (1) a programming API for the Fresh Breeze memory model on a modern many-core architecture; (2) an experimental implementation of the API through simulation by using the FAST simulator for the 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 · Massively parallelism · Concurrency model · System simulation

Jack B. Dennis
Stata Center, MIT, Room 32-G868, 32 Vassar Street, Cambridge, MA 02139
Tel.: +1-(617)253-6856
Fax: +1-(617)253-6652
E-mail: dennis@csail.mit.edu

Guang R. Gao
140 Evans Hall, University of Delaware, Newark, DE 19716
Tel.: +1-(302)831-8218
Fax: +1-(302)831-4316
E-mail: ggao@capsl.udel.edu

Xiao X. Meng
140 Evans Hall, University of Delaware, Newark, DE 19716
Tel.: +1-(302)831-6534
Fax: +1-(302)831-4316
E-mail: meng@capsl.udel.edu

## 1 Introduction

The Fresh Breeze memory model and system architecture [1, 2] is proposed to provide 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, Figure 1, 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 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 will provide an ability for modular

composition of parallel programs well beyond what is possible with any existing computer system.

The main contribution of this paper includes:

1. A programming API for the Fresh Breeze memory model on a modern many-core architecture.
2. An experimental implementation of the API through simulation by using the FAST simulator for the Cyclops 64 many-core chip. It is a real chip in production use since 2008, and the performance of the FAST simulator has been calibrated against the real chip through various measurements under a collaboration between the chip design team at IBM and the system software team at ETI.
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 Section 2 the Fresh Breeze memory model is presented. Section 3 describes the programming API of the Fresh Breeze memory model. A vision of future computer system organization utilizing Fresh Breeze principles is provided in Section 4 and 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.

## 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, a capability, that serves to locate the chunk within the storage system, and is a globally valid reference or handle to the chunk. Each chunk can then hold up to 16 elements that are either a 64-bit data or handle. A collection of chunks organized as a directed acyclic graph (DAG) can represent structured information as illustrated in Figure 1. For example, a three-level tree of chunks could represent an array of 16 * 16 * 16 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

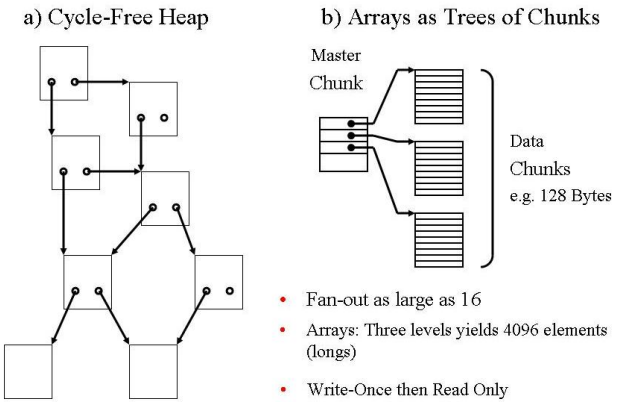


Fig. 1: Data objects as trees of chunks.

chunk may be summarized as follows: (1) A 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 is 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.

One benefit of a write-once memory model is that cache memories may be used without coherency 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 to files and databases of conventional systems.

## 3 API of the Fresh Breeze Memory Model

To map and demonstrate the fresh breeze memory model on a modern many-core chip architecture and systems, we have proposed and designed an instruction level application programming interface (API). It expresses the program execution concurrency [7] in a way similar to the spawn/join model of Cilk [8] parallel programming.

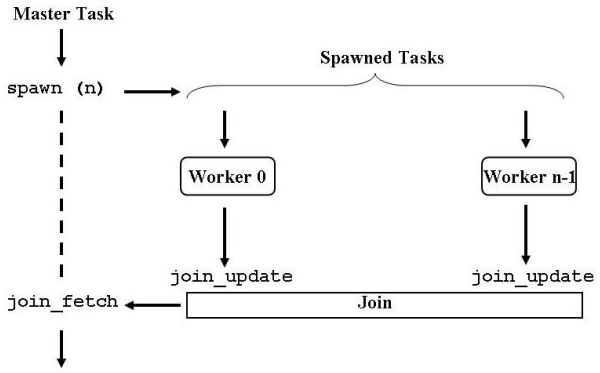


Fig. 2: Fresh Breeze parallelism using Spawn and Join.

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. As shown in Figure 2 a task may spawn one or more worker tasks executing independent instances of the same or different functions. Worker tasks may read data objects (scalar values or capabilities) from their parent task, and each worker task contributes the results of its activity to the parent 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.

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 Fun-Java 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, count);
        for (int idx = 0; idx < count; idx++) {
            // Calculate node size and subtree size
            node_size = ... ;
            tree_size = ... ;
            spawn_one (idx, DotProduct (
                    chunk_a[idx], chunk_b[idx], size, tree_size) );
        }       exit ();
    } else {
        // Process a leaf node
        long sum = 0;
        for (int idx = 0; idx < count, idx++ ) {
            sum += chunk_a[idx] * chunk_a[idx];
        }
        join_update (sum);
    }
}
void DotProductDone (int count) {
    handle data = join_fetch ();
    chunk join_data = chunk_read [idx];
```

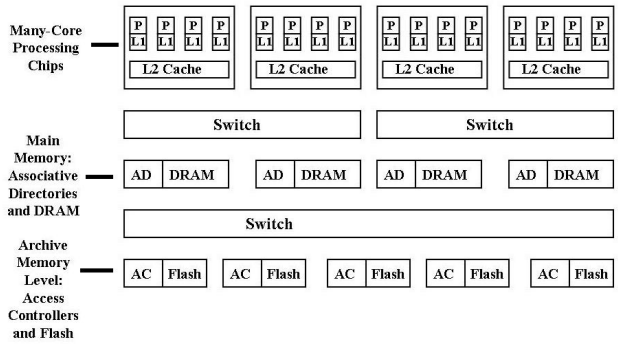


Fig. 3: Vision of a massively parallel Fresh Breeze system.

```
    long sum = 0;
    for (int idx = 0; idx < count; idx++) {
        sum += join_data [idx];
    }
    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 to other tasks.

## 4 Computer System Structure and the Memory Hierarchy

The envisioned organization of a Fresh Breeze computer system is illustrated in Figure 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 [9], 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 will be tagged to flag those holding capabilities.
- A new load/store unit will be used to provide creating (writing) and reading execute support for memory chunks.

**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 [10].

In Figure 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 is the capability code of a chunk, and the physical location where it 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 free capability codes to the processing chips for assignment to newly created chunks. A data structure is maintained, that keeps a record of available codes. Capability codes are assigned from the free pool and returned to the pool when the reference count shows they are no longer needed.

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 identifiers (global pointers) to the 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 capability codes. Accordingly, each directory must map to a sufficiently large physical space to accommodate all data in its part, and its implementation must be able to handle the anticipated traffic, although a relatively long search time may be acceptable to reduce cost.

## 5 Simulation Facility

A simulation model of a two-level Fresh Breeze memory system has been implemented. It uses an existing simulation system [11], built by a collaboration of IBM and E.T. International, for testing and evaluating the IBM Cyclops 64 many-core chip [9]. The chip contains 80 processing assemblies, each consisting of two independent Thread Units (TUs) sharing a 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 1Gb of off- chip DRAM memory through four additional ports of the X-bar network.

In our Fresh Breeze simulation, 40 thread units serve as E-processors that execute application tasks; most of the remaining 120 are S-processors used to implement a simulation of the Fresh Breeze Storage System, using SRAM for associative directories of a top storage level and the DRAM for a shared main storage level. Runtime software has been written to schedule user tasks on the E-processors and to implement the Storage System simulation. Test programs are written in C and compiled by the Cyclops C compiler.

## 6 Scheduling and Work Stealing

The Fresh Breeze simulation models a hardware scheduling mechanism in each of the E-processors. 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 E-processor 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 it quits.)

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 (Section 8), the ATL for each E-processor 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 the simulated E-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, write the result value (scalar or handle) into the **join_data** chunk, update the join count, and terminate the worker task.
2. On a **join_fetch** instruction, return the handle of the **join_data** chunk to the master application 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. 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).

The simulation uses two tables located in memory globally accessible by all processors of a domain or cluster of processors in a large system. This approach can be extended hierarchically as needed. These tables are managed by a reserved Steal Daemon processor 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 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 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 our 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 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 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 System 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, so we use the "main level" access time as a principal parameter 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.

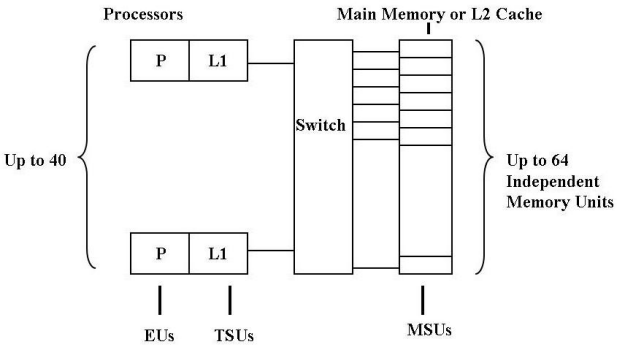


Fig. 4: Fresh Breeze system for modeling with two memory levels.

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

We assume the upper memory level (L1) may be accessed in two clock cycles and that read one chunk of data into processor register 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 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 16K 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 action and they depart significantly from a uniform multiple of the clock cycles needed in the modeled system. The following table 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 and the simulation cycles used by the corresponding simulation routine. For our experiments, we made the simulation time exactly proportional to the modeled system

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 | 4844 | 5120 |
| Task (Save/Restore) | 16 | 262 | 4095 | 2560 |
| Shared Mem. Data Transfer | 16 | 3047 | 0 | 2560 |

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

The simulation experiments are conducted for two scenarios: In the first scenario, the Shared Memory System 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 action 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 Fresh Breeze architecture, task switching times are sufficiently short that it may be beneficial to use a *blocking read* wherein the processor is switched to a different task while a **chunk_read** operation is performed. For these tests the Task Save/Restore action model the retention of processor register state across read operations.

The Shared Memory System is modeled by simulation routines running on each Cyclops processor used to simulate the Shared Memory. Each routine maintains a queue of access requests for each separate memory unit. 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 and the chunk location determined. 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 to operate 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.

## 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.

Table 2: Number of task executions and operations

| vector length | leaf tasks | non-leaf tasks | total tasks | adds | multiplies |
|---|---|---|---|---|---|
| $16^3$ | 256 | 17 | 273 | 4095 | 4096 |
| $16^4$ | 4096 | 273 | 4369 | 65535 | 65536 |
| $16^5$ | 65536 | 4369 | 69895 | 1048575 | 1048576 |

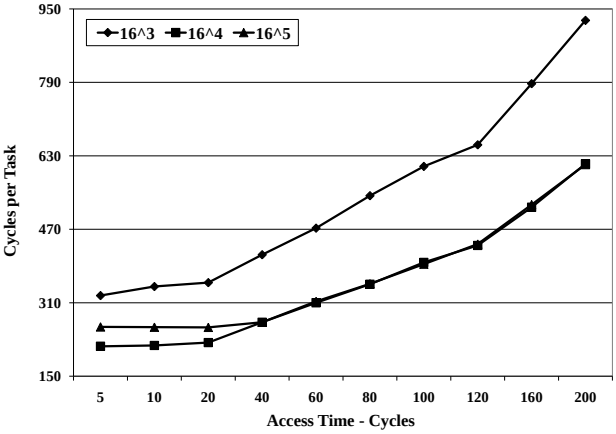


Fig. 5: Non-blocking read scenario: system cycles per task.

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 the simulation run. The charts show the performance for three vector lengths and various shared memory access times for the two cases of interest. In Figure 5 reads are non-blocking, modeling behaivoir of an L2 shared cache; In Figure 6 reads are blocking, with suspension of the task and swapping procesor 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.

The best performance shown in these runs achieves an average of 200 cycles per task. Using the numbers of leaf and non-leaf tasks from the table 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 $(30*500)/200 = 75$ million

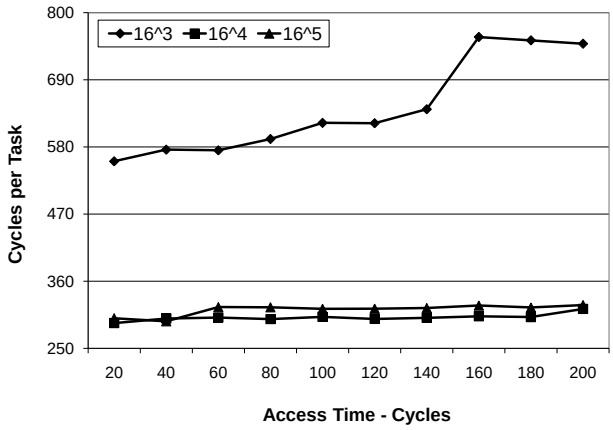


Fig. 6: Blocking read scenario: system cycles per task.

operations per second per processor or 3000 MOPS for the set of 40 processors.

In addition to average performance, the simulations have demonstrated the effectiveness of hardware-supported work stealing. Figures 7 show how well the task processing load is distributed over the 40 processors for the Main Memory model. 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 30 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 compiler.

### 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 Figure 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

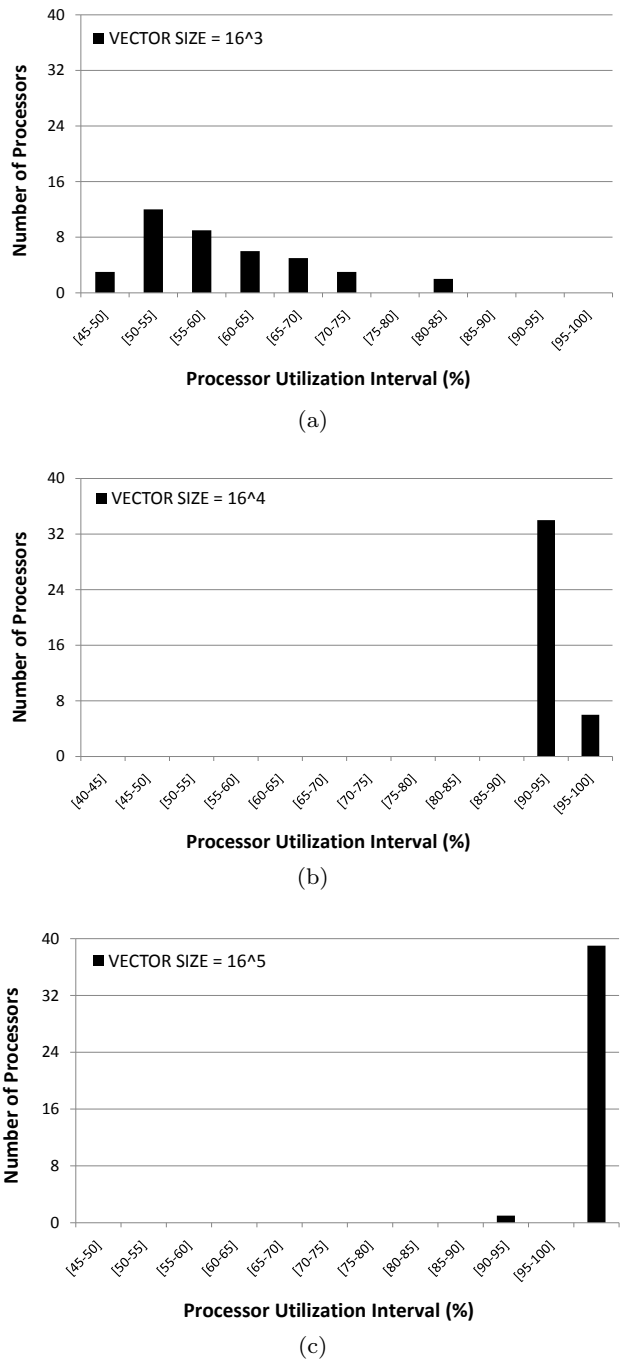


Fig. 7: Load distribution performance of work stealing for Main Memory

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 practical scalable architectures for memory hierarchies. It follows that the locations of data structures must be considered in the design of any scalable, gen-

eral task distribution scheme. This is expected to be a challenge for future 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 further 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 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 which will be studied for the Fresh Breeze architecture. In general, the cache mechanism will likely be an important contribution to overall performance in future Fresh Breeze designs.

### 8.4 Excess Parallelism

The fine-grained hardware-implemented resource management results in that computations will generally 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 under main memory model with 100 cycles of access latency (which is a typical value for the current main memory technology), all but 412 of which 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 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 [12], and a successful commercial implementation is used in the IBM AS/400 systems [13]. A software implementation of capabilities is available [6] and a successor Coyotos is under development. However, these are software implementations that do not have the tight security feature of hardware-based capabilities. The write-once storage concept has been 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 [8,14], EARTH [15,16] and the scheduling of parcels in HTMT [17]. 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 task to processor, to granting the system the responsibility of making the assignment. This work is a revision of the X10 programming language, which uses the asynch/finish concurrency control primitives [18–21]. Related work appears in the HPC language Cascade [22].

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 [23]. The work stealing technique used in the reported simulations may be regarded as an implementation of Cilk ideas using similar principles to the Sigma 1.

Tools for conducting system evaluation through simulation and emulation is an area of active work [24, 25]. The RAMP project [26] system developed at Berkeley is a good example. It is a FPGA-based many core emulation platform. This system deploys Xilinx Vertex-II Pro FPGAs on 16-21 BEE2 boards to implements a many core system composed of 1000 plus 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 simulation tool used in this paper is an industry-strength system that can simulate the entire logic of the IBM Cyclops-64 chip with

its 160 cores [11]. An implementation of a system emulation facility equivalent to the FAST simulator has been constructed using FPGA devices and is used for the validation of both architecture and system software implementation.

## 10 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.

## References


1. J. B. Dennis, "A parallel program execution model supporting modular software construction," in *Massively Parallel Programming Models*, pp. 50–60, IEEE, 1997.
2. J. B. Dennis, "Fresh breeze: a multiprocessor chip architecture guided by modular programming principles," *SIGARCH Comput. Archit. News*, vol. 31, no. 1, pp. 7–15, 2003.
3. J. B. Dennis and E. C. V. Horn, "Programming semantics for multi-programmed computations," *Communications of the ACM*, vol. 9, Feb 1966.
4. H. Levy, *Capability-Based Computer Systems*. Newton, MA: Butterworth-Heinemann, 1984.
5. M. V. Wilkes, *The Cambridge CAP computer and its operating system (Operating and programming systems series)*. Operating and Programming Systems Series, Amsterdam, The Netherlands: North-Holland Publishing Co., 1979.
6. J. S. Shapiro, J. M. Smith, and D. J. Farber, "Eros: a fast capability system," in *Proceedings of the Seventeenth ACM symposium on Operating Systems Srinciples*, SOSP '99, (New York, NY, USA), pp. 170–185, ACM, 1999.
7. J. B. Dennis, "The Fresh Breeze model of thread execution," in *Workshop on Programming Models for Ubiquitous Parallelism*, IEEE, 2006. Published with PACT-2006.
8. M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *ACM SIGPLAN Notices*, vol. 33, pp. 212–223, May 1998.
9. J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Tiny threads: A thread virtual machine for the Cyclops 64 cellular architecture," in *International Parallel and Distributed Processing Symposium*, p. 265, IEEE, 2005.
10. B. Schmidt, "A shared memory system for fresh breeze," Master's thesis, MIT Department of Electrical Engineering and Computer Science, May 2008.
11. J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Fast: A functionally accurate simulation toolset for the cyclops64 cellular architecture," 2005.
12. A. Bensoussan, C. T. Clingen, and R. C. Daley, "The Multics virtual memory," in *Proceedings of the Second Symposium on Operating Systems Principles*, (New York), pp. 30–42, ACM, 1969.
13. F. G. Soltis, *Inside the AS/400*. Duke Press, 1996.
14. V.-Y. Vee and W.-J. Hsu, "Applying Cilk in provably efficient task scheduling," *The Computer Journal*, vol. 42, pp. 699–712, 1999.
15. K. B. Theobald, *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, University of Delaware, May 1999.
16. H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, and G. R. Gao, "A design study of the EARTH multiprocessor," in *Conference on Parallel Architectures and Compilation Techniques*, PACT, pp. 59–68, IEEE, 1995.
17. K. B. Theobald, G. R. Gao, and T. L. Sterling, "Superconducting processors for HTMT: Issues and challenges," in *ACM '87: The 7th Symp. on the Frontiers of Massively Parallel Computation: Today and Tomorrow*, (New York), pp. 260–267, ACM, 1999.
18. P. Charles, C. Grotho, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *2005 Conference on ObjectOriented Programming*, (New York), pp. 519–538, ACM, 2005.
19. V. Sarkar and J. Hennessy, "Compile-time partitioning and scheduling of parallel programs," in *86 Symposium on Compiler Construction*, SIGPLAN, (New York), pp. 17–26, ACM, 1986.
20. J. Shirako, D. Peixotto, V. Sarkar, and W. Scherer, "Phasers: A unified deadlock-free construct for collective and point-to-point synchronization," in *Twenty-second International Conference on Supercomputing*, IEEE, 2008.
21. Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *International Parallel and Distributed Processing Symposium*, IPDPS, IEEE, 2009.
22. D. Callahan, B. L. Chamberlain, and H. P. Zima, "The Cascade high productivity language," in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.
23. T. Yuba, K. Hiraki, T. Shimada, S. Sekiguchi, and K. Nishida, "The sigma-1 dataflow computer," in *ACM '87: Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, (Los Alamitos, CA, USA), pp. 578–585, IEEE Computer Society Press, 1987.
24. J. Darringer, E. Davidson, D. Hathaway, B. Koenemann, M. Lavin, J. Morrell, K. Rahmat, W. Roesner, E. Schanzenbach, G. Tellez, and L. Trevillyan, "Eda in ibm: past, present, and future," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, pp. 1476 –1497, dec. 2000.
25. M. Dubois, J. Jeong, Y. Song, and A. Moga, "Rapid hardware prototyping on rpm-2.," *IEEE Des. Test. Comput*, pp. 112–118, 1998.
26. J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanovic, "Ramp:

Research accelerator for multiple processors," *Micro, IEEE*, vol. 27, pp. 46 –57, mar. 2007.