A RISC-V Extension for the Fresh Breeze Architecture

Jack B. Dennis
MIT CSAIL
Cambridge, MA
dennis@csail.mit.edu

Willie Lim
MIT CSAIL
Cambridge, MA
wlim@csail.mit.edu

ABSTRACT

We report on a RISC-V extension for a novel multi-core computer organization able to execute applications with high performance and energy efficiency. Novel features of this architecture include support for data objects represented by trees of 128-byte memory *chunks*, and hardware implementation of task scheduling and load balancing. We call our project *Fresh Breeze*[1] in view of its novelty and potential.

User programs are written in *funJava*, a functional subset of the Java programming language, compiled into independent blocks of instructions called *codelets*, and run on an architecture model using our *Kiva* simulator.

Extensions to the RISC-V core will consist of special instructions for creating and accessing memory chunks, and for spawning and coordinating tasks for codelet execution. Also, the core processor will include an *AutoBuffer* that holds memory chunks for direct access that are automatically loaded in response to read instructions.

These extensions will permit us to build an FPGA Fresh Breeze prototype using the BlueDBM facility of the Computation Structures Group in the MIT Computer Science and Artificial Intelligence Laboratory.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**;
• **Hardware** → *Simulation and emulation*;

KEYWORDS

Fresh Breeze, RISC-V, trees of chunks, fine grain tasking

ACM Reference format:
Jack B. Dennis and Willie Lim. 2017. A RISC-V Extension for the Fresh Breeze Architecture. In *Proceedings of First Workshop on Computer Architecture Research with RISC-V, Boston, Massachusetts USA, October 2017 (CARRV2017)*, 6 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Over the past several years the authors have worked with Prof. Guang Gao of the University of Delaware and Prof. Huang Lei of Prairie View A & M University on a multi-core computer organization able to execute applications with high performance and energy efficiency. Novel features of this architecture include support for data objects represented by trees of 1024-bit (128-byte) memory *chunks*, and hardware implementation of task scheduling and load balancing. We call our project *Fresh Breeze* in view of its novelty and potential. The merits of these features have been shown through early simulation experiments[5, 12].

Fresh Breeze user programs are written in *funJava*, a functional restricted subset of the Java programming language, are compiled [4] into independent blocks of instructions called *codelets*, and run on an architecture model using our *Kiva* simulator. Our recent simulation experiments using a neural network computation [7] have shown that the architecture can exploit fine grain parallelism and rapidly distribute thousands of tasks over many processing units. Linear scaling of performance has been demonstrated for up to 256 processing units.

We wish to advance our work toward the design and fabrication of a Fresh Breeze multi-core processor chip by building a prototype using FPGA technology. The RISC-V technology provides an ideal basis for this step because RISC-V is amenable to the needed processor extensions, and RISC-V support is available with the BlueDBM [8] facility at the Computation Structures Group in the MIT Computer Science and Artificial Intelligence Laboratory.

The following sections provide an overview of the Fresh Breeze architecture and programming model, and explain the RICS V extensions needed to implement trees of chunks representation of data objects and the fine grain tasking of the programming model.

2 FRESH BREEZE ARCHITECTURE

Figure 1 shows a Fresh Breeze system [2] with four *Processing Units* connected to four *Memory Units* through two 4×4 (4-input, 4-output) packet switching networks – one for routing memory commands from the Processing Units to the Memory Units and the other for routing memory responses from the Memory Units back to the Processing Units. A large system with N Processing Units and N Memory Units will use $N \times N$ packet networks.

Each Processing Unit has a *Task Scheduler* that holds a queue of *TaskRecords* of tasks awaiting execution by the Processing Unit. The *Load balancer* distributes tasks evenly over the Processing Units by directing the Task Schedulers of busy Processing Units to send TaskRecords of pending tasks to less busy ones.

Each Processing Unit also has an associated *AutoBuffer* that holds active memory chunks for direct access, as in a conventional cache memory. However use of a cache tag memory is avoided by the scheme described in Section 5.2. When a read instruction references a chunk not present in the AutoBuffer, the chunk is loaded by sending a read command to the appropriate Memory Unit. Write instructions only refer to newly created chunks in

[1]The 'Fresh Breeze' name is chosen as a continuation of the MIT practice of naming computer projects after weather patterns in the spirit of Whirlwind [11] and Monsoon [10].

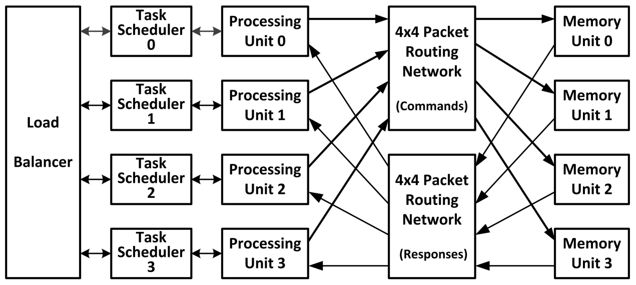

Figure 1: A four-core Fresh Breeze system.

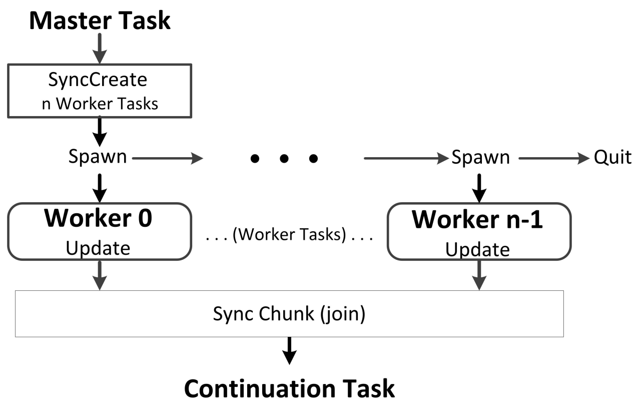

Figure 2: Spawning a team of workers in the Fresh Breeze tasking model.

the AutoBuffer; when the task that writes to elements of a new chunk terminates, the new chunk is written to its assigned Memory Unit and no further writes of the chunk are permitted. The least recently accessed chunks in the AutoBuffer are chosen for eviction as needed. Note that the write-once property permits eviction of chunks without the need to write the chunks back to Memory Units.

To provide latency tolerance in memory access, a number of *execution slots* is used. In our simulated Fresh Breeze system, each Processing Unit has six execution slots. Each of these slots has its own complete set of registers, program counter, and a set of chunk buffers in the AutoBuffer. When execution of instructions in a task cannot proceed due to its waiting for a response from a Memory Unit, that task is suspended and the Task Scheduler switches the core to an active task in another execution slot, or starts a new task if there are no slots with active tasks.

The RISC-V extensions needed to implement mutiple execution slots are described in Section 5, together with the special instructions that will be implemented for chunk-based representations of data objects and fine grain task scheduling.

3 FRESH BREEZE PROGRAMMING MODEL

The Fresh Breeze system architecture directly supports a programming model designed for natural expression of all forms of parallelism present in applications. This includes expression parallelism, data parallel computation, producer-consumer parallelism and concurrent transaction processing. In addition, the programming model satisfies requirements for modular software construction [2], ensuring that any program module may be used without change as a component of other program modules. This section describes data representation and tasking in the Fresh Breeze programming model.

3.1 Data Object Representation

All data objects in a Fresh Breeze system are represented by trees of fixed-size *chunks* of memory. A chunk holds up to 16 items, each of which may be either a data value or the *handle* of another memory chunk. The handle of a memory chunk serves as a globally valid means to locate the chunk within the storage system. The collection of all memory chunks forms a multi-rooted directed acyclic graph (DAG) that is the "heap" held by the Fresh Breeze multi-level memory system. Chunks are created and filled with data, but are frozen before being shared with concurrent tasks. This *write-once* policy eliminates data consistency issues and simplifies memory management by precluding creation of cycles in the graph of chunks and references.

Such a memory model provides a global addressing environment, a virtual one-level store, shared by all user jobs and all cores of a many-core multi-user computing system. It covers the entirety of online storage, replacing the separate means of accessing files and databases in conventional systems, thereby eliminating the distinction between "in core" and "out of core" versions of algorithms.

3.2 Tasking Model: Codelets

The basic unit of parallelism in a Fresh Breeze system is a *task*, the activity of performing a single execution of a block of instructions called a *codelet*. The organization of multiple tasks is expressed in a way similar to the spawn/join model for parallel programming of Cilk [6]: As shown in Figure 2, a *master task* may spawn one or more *worker tasks* to execute independent instances of the same or different codelets. Worker tasks may receive data objects as arguments provided by the master task, and each worker task contributes the results of its activity to a *continuation task* using a special type of memory chunk called a *sync chunk* [3]. The Fresh Breeze tasking model differs from Cilk in that the master task does not continue after spawning the workers and there is no interaction between the master and the worker or among the workers other than the contribution of each worker to the continuation task. The scheme matches the data parallel features of a programming language such as Sisal [9] or NESL [1]. Through recursive use of this scheme, a program can generate an arbitrary hierarchy of concurrent tasks corresponding to available parallelism in the computation being performed. Sync chunks can also implement the synchronization required for the producer/consumer parallelism prevalent in processing data streams.

4 FRESH BREEZE CODELETS

To illustrate how codelets are constructed to implement the scheme of Figure 2 we present in this section three of the Fresh Breeze codelets generated to perform the dot product computation for two vectors of equal but arbitrary length. The funJava method for

```
long DotProduct(long[] a, long[] b, int len) {
  long sum = 0;
  for (int i = 0; i < len; i++) {
    sum = sum + a[i] * b[i];
  }
  return sum;
}
```

Figure 3: The dot product method written in funJava.

this computation is shown in Figure 3. In the Fresh Breeze implementation each of the given vectors is represented by a tree of chunks such that each vector is divided into a series of corresponding 16-element segments represented by leaf chunks of the two trees.

Figure 5 shows the codelet (Codelet 2) that performs the dot product of a pair of leaf segments of the given vectors, and in Figure 4 the portion of Codelet 1 that spawns up to 16 worker tasks that execute Codelet 2. The principal arguments of Codelet 1 are the handles of the two-level subtrees whose root nodes are the parent chunks of the leaf nodes. Figure 6 shows Codelet 3, the continuation codelet that is executed after all worker tasks have completed their computations.

These figures show the Fresh Breeze instructions of each codelet. Instructions access machine registers using register indexes. Registers are 64 bits wide, so operands that are full words are accessed using even indexes; 32-bit half-word operands are accessed using even integers for the left half and odd integers for the right half of a register.

The following abbreviations are used:

`S0`:	source zero;
`S1`:	source one;
`LV`:	literal value;
`D`:	destination;
`H`:	handle;
`Off`:	offset or index;
`Lab`:	label of statement to jump to.

Most instructions of the Fresh Breeze ISA are similar to typical RISC instructions; the exceptions are the special instructions for memory operations and special instructions for implementing the tasking scheme of Figure 2. In Figure 4, The *SyncCreate* (instruction # 22) creates a sync chunk set to spawn continuation Codelet 3 when all worker tasks have terminated. This is followed by a loop that executes *TaskSpawn* (instruction # 38) to initiate each worker task using Codelet 2. Then *TaskQuit* (instruction # 41) terminates the task.

The code in Figure 5 is a straightforward loop for computing the dot product. On completion the *SyncUpdate* (instruction # 13) is used to forward the result in register # 16 to the sync chunk, after which the codelet terminates with a *TaskQuit* instruction (instruction # 14). Arguments zero and one of the *SyncUpdate* instruction provide the handle of the sync chunk and the index of the worker task for which the result is being submitted. Execution of the *SyncUpdate* by the last worker task to terminate causes Codelet 3 to be spawned with argument zero (register # 0) holding the handle of a chunk containing the results (**long** values in this case) of worker tasks.

```
Codelet 1:
   19]: LMove S0: 0; -> D: 26
   20]: IMove S0: 2; -> D: 28
   21]: IMove S0: 41; -> D: 29
   22]: SyncCreate Code: 3;
           sigCnt: 11; itemCnt: 41 -> D: 26;
           argsBase: 13 argsCnt: 2
   23]: ISet 0 -> D: 28
   24]: IfIGeq S0: 28; S1: 41; Lab: 41
   25]: ISub S0: 41; S1: 0; LV: 1; -> D: 52
   26]: IfINeq S0: 28; S1: 52; Lab: 29
   27]: IMove S0: 42; -> D: 53
   28]: Jump Lab: 30
   29]: IMove S0: 37; -> D: 53
   30]: ReadFull H: 4; Off: 28; -> D: 54
   31]: ReadFull H: 6; Off: 28; -> D: 56
   32]: IMul S0: 28; S1: 37; -> D: 58
   33]: IAdd S0: 8; S1: 58; -> D: 59
   34]: LMove S0: 54; -> D: 30
   35]: LMove S0: 56; -> D: 32
   36]: IMove S0: 59; -> D: 34
   37]: IMove S0: 53; -> D: 35
   38]: TaskSpawn Code: 2; argsBase: 13; argsCnt: 5
   39]: IAdd S0: 28; S1: 0; LV: 1; -> D: 28
   40]: Jump Lab: 24
   41]: TaskQuit
```

Figure 4: Portion of Fresh Breeze Codelet 1 for creating the sync chunk and spawning worker tasks.

```
Codelet 2:
    0]: ISet 1 -> D: 10
    1]: ISet 0 -> D: 11
    2]: LSet 0 -> D: 12
    3]: IMove S0: 11; -> D: 14
    4]: LMove S0: 12; -> D: 16
    5]: IfIGeq S0: 14; S1: 9; Lab: 13
    6]: IAdd S0: 14; S1: 8; -> D: 15
    7]: ReadFull H: 6; Off: 14; -> D: 18
    8]: ReadFull H: 4; Off: 14; -> D: 20
    9]: LMul S0: 18; S1: 20; -> D: 22
   10]: IAdd S0: 14; S1: 0; LV: 1; -> D: 14
   11]: LAdd S0: 16; S1: 22; -> D: 16
   12]: Jump Lab: 5
   13]: SyncUpdate Sync: 0; Off: 2; Data: 16
   14]: TaskQuit
```

Figure 5: Fresh Breeze Codelet 2 for performing the dot product of two leaf vector segments of 16 or fewer elements.

The instructions of Codelet 3 (Figure 6) implement a loop that reads (instruction # 4) and sums (instruction # 5) values from the results chunk and forwards the sum with a *SyncUpdate* (instruction # 8).

```
Codelet 3:
   0]: ISet CV: 1 -> D: 7
   1]: ISet CV: 0 -> D: 10
   2]: LSet CV: 0 -> D: 8
   3]: IfIGeq S0: 10; S1: 5; Lab: 8
   4]: ReadFull H: 0; Off: 10;  -> D: 12
   5]: LAdd S0: 8; S1: 12; -> D: 8
   6]: IAdd S0: 10; S1: 7; LV: 1; -> D: 10
   7]: Jump Lab: 3
   8]: SyncUpdate Sync: 4; Off: 4; Data: 8
   9]: TaskQuit
```

Figure 6: Fresh Breeze continuation codelet (Codelet 3) for summing the leaf segment results.

Implementation details of the memory and tasking instructions are presented in Section 5.3 and Section 5.6, respectively. The memory commands and responses exchanged between the Processing Units and Memory Units are briefly described in Section 5.5.

5 RISC-V EXTENSIONS

For the Processing Units of a Fresh Breeze multi-core processing chip, we propose using a basic RISC-V core with extensions to implement multiple execution slots, the AutoBuffer, the trees of chunks memory representation for data objects, and operations for tasking. This will create a Processing Unit able to perform all instructions used in the code examples in Figures 4, 5 and 6. All of the standard arithmetic and logic instructions will be retained as implemented in the basic RISC-V core with the double precision and floating point extensions. For Fresh Breeze memory operations and fine grain tasking, the RISC-V ISA will be extended to include special instructions as described in the paragraphs that follow.

To support tasking as discussed in Section 3, we will add instructions to support creation and management of tasks. Execution of a Fresh Breeze task requires the index of the codelet to be executed, and the set of up to sixteen argument values that constitute input data for the task. This information is contained in a data object called a *TaskRecord* which has four fields:

- *taskId (32 bits):* The task identifier; used for debugging purposes or to track movement of a TaskRecord.
- *codeIdx (16 bits):* The index of the codelet (set of instructions) to be executed.
- *argsCnt(4 bits):* The number of arguments needed for the task.
- *argsList (64 bits):* The handle of a chunk containing the arguments.

The simplicity of the TaskRecord is a consequence of using a common global address space (the collection of chunk handles) for all tasks and for all users of a Fresh Breeze system. The Task Scheduler for each Processing Unit is simply a FIFO queue in which the entries are TaskRecords. Furthermore load balancing is performed simply by moving TaskRecords from one Task Scheduler to another. This is why a Fresh Breeze System can scale to utilize large numbers of Processing Units effectively for normal problem sizes.

The RISC-V extensions needed for use as a Processing Unit of the Fresh Breeze architecture are:

- Several selectable register sets to support multiple execution slots.
- Cache memory extension adapted to implement the Fresh Breeze AutoBuffer.
- Instructions for building and accessing data objects represented as trees of memory chunks.
- Instructions to support spawning and coordination of worker tasks.

The subsections that follow describe these extensions and the needed special instructions. The version of RISC-V to be used as the base for these extensions will have 32 64-bit machine registers. Instruction fields for source and destination will be five bits wide as used in the standard instruction formats. Therefore the choice of which half of a register is to be used for reading and writing 32-bit values must be indicated by the opcode (instruction name). The values in source and destination fields will always be integers in the range 0, .., 31. Implementation of the memory instruction requires means for sending command packets to and receiving response packets from the Memory Units. For completeness the structure of command and response packets is described.

5.1 Support for Execution Slots

To support rapid switching of task execution between execution slots, a *Current Slot* register will be used to select the register set, program counter, and portion of the AutoBuffer that are active. The Current Slot is set by the Task Scheduler when it switches the processor between execution slots as a task becomes blocked or terminates. This scheme is similar to the *simultaneous multithreading* described by Tullsen, Eggers and Levy [13].

5.2 AutoBuffer Implementation

The Fresh Breeze architecture uses an AutoBuffer in place of the usual level one cache. The AutoBuffer has several chunk buffers for direct access by the processor for each execution slot. For implementation of direct access, each processor register has an extra *index* field and *valid* bit. If the valid bit is on, the index indicates which buffer holds the chunk identified by a handle held in the register. The index and valid flag are set by the **ChunkCreate** instruction, or when the chunk is loaded into the AutoBuffer in response to a read instruction. The RISC-V support for cache implementation will be adapted for realizing the AutoBuffer.

5.3 Memory Instructions for Trees of Memory Chunks

A memory chunk holds 16 data items, each of which can hold one value of long or double type, or two values of int or float type. Each chunk is identified by a 64-bit *handle*, which can be held by a data item of type long, providing the basis for building a tree as a hierarchy of memory chunks.

The memory instructions are these:

ChunkCreate `(dest)` A memory chunk is allocated and its handle is written in the `dest` register.

WriteFull `(handle, index, value)` The item selected by `index` in the chunk identified by `handle` is updated to hold the long or double value from the `value` register.

WriteLeft `(handle, index, value)` The left half of the item selected by `index` in the chunk identified by `handle` is updated to hold the int or float value from the left half of the `value` register.

WriteRight `(handle, index, value)` The right half of the item selected by `index` in the chunk identified by `handle` is updated to hold the int or float value from the right half of the `value` register.

ReadFull `(dest, handle, index)` The long or double value in the item selected by `index` in the chunk identified by `handle` is read and written in the `dest` register.

ReadLeft `(dest, handle, index)` The int or float value in the left half of the item selected by `index` in the chunk identified by `handle` is read and written in the left half of the `dest` register.

ReadRight `(dest, handle, index)` The int or float value in the right half of the item selected by `index` in the chunk identified by `handle` is read and written in the right half of the `dest` register.

These instructions will replace the load and store instructions of the RISC-V ISA.

5.4 Garbage Collection

In the Fresh Breeze architecture the reclamation of free chunk space in each Memory Unit will be done automatically by the hardware. The reference count scheme will be used. Reference counts are held as metadata for each chunk in each Memory Unit and are accessed using the handle of the chunk. The reference count of a chunk is initially zero. It is incremented by one by the *ChunkCreate* instruction and whenever a copy of the handle is made, for example, when the handle is used as an argument of a *TaskSpawn* instruction. When a task terminates the reference count of any chunk for which a handle is present in a machine register is decremented. As usual, when a reference count becomes zero, the chunk is marked free and the reference counts of any chunks referenced by handles in the freed chunk are decremented. Implementation of the special instructions for memory operations and tasking will include actions to implement this garbage collection scheme.

5.5 Memory Commands and Responses

The Fresh Breeze Processing Unit interacts with the Memory Units by sending Memory Commands to the Memory Units for many of the special memory and tasking instructions, and receiving the responses. These interactions will be implemented in the corresponding RISC-V extensions and by adaptation of the RISC-V interrupt facility. The purpose of each command and response is provided in the following paragraphs.

Memory Commands: Fresh Breeze Processing Units send memory commands to Memory Units to instruct the latter to perform memory related tasks for the special memory and tasking instructions.

HandleRequest Return a chunk containing handles of free chunks.

ChunkSave Save a chunk at a specified chunk address.

ChunkLoad Read a chunk from the specified chunk address.

MemUpdate Update the item index of a Sync Chunk at the given chunk address and test whether all workers have submitted results.

DataUpdate Update the result data item at the specified item index in the sync results chunk at the specified address.

Memory Responses: After processing a memory command, the Memory Unit sends back to the requesting Processing Unit one of these memory responses:

FreeHandles Return a chunk containing 16 handles of free chunks in response to a **HandleRequest** command.

MemChunk Return the chunk at the given chunk address in response to a **ChunkLoad** command.

SyncData Return information from the Sync Chunk just updated: the handle of the results chunk and a flag to indicate whether all worker tasks have submitted their results. This response is sent for a **MemUpdate** command generated by a *SyncUpdate* instruction.

MemDone Indicate to the requesting Processing Unit that its memory command has been executed as a response to a **ChunkSave** or **DataUpdate** command.

5.6 Instructions for Tasking

Figure 2 shows the use of tasking instructions in codelelets to spawn a team of worker tasks to perform a data parallel computation. These instructions make use of the *Pending Task Queue (PTQ)* of the Task Scheduler. The following are descriptions of the steps in the execution of the tasking instructions.

SyncCreate `(dest, code, count)` A special sync chunk is created containing: 1. the index `code` of the continuation codelet; and 2. the number `count` of worker tasks that will contribute results. A data item containing the handle of the sync chunk is written to the `dest` register.

TaskSpawn `(code, args, sync, index)` A TaskRecord is constructed and entered in the PTQ. When a processor is given the TaskRecord for codelet execution, a task is initiated to execute codelet `code`, where `args` is the handle of a chunk containing argument data items for use by the codelet. Item zero of the arguments chunk must contain the handle of the sync chunk where the result of codelet execution will be registered, and item one the index to indicate which of several worker tasks has produced the result.

SyncUpdate `(sync, index, result)` This instruction is used by a codelet to register its result with a sync chunk for use by the continuation codelet: `sync` is the handle of the sync chunk; `index` tells the sync chunk which task is supplying the result; and `result` is a data item containing the result, which may be a scalar value or the handle of a data chunk.

TaskQuit `()` This is the last instruction executed by each codelet and terminates codelet execution.

These tasking instructions will be implemented as variants of RISC-V System call instructions.

6 CONCLUSION

From our recent study of the performance of funJava programs for machine learning and our earlier studies of linear algebra kernels (matrix multiply, lower upper decomposition or LUD, and dot product), we observe that for sufficiently large input data the Fresh Breeze performance of these computations scales linearly as the number of Processing Units increases up to at least 256 Processing Units. We believe this is due to two capabilities of the Fresh Breeze architecture and compiler:

(1) The ability to decompose computations into many parallel data driven tasks.
(2) Efficient load balancing using hardware to ensure that tasks ready for execution are quickly and evenly distributed among all available Processing Units.

This observation also applies to our experiments with multi-program computations where more than one funJava program is run at the same time on a Fresh Breeze system. All the system needs to know is that there are tasks ready to be executed. It does not matter whether the tasks are from the same computation or from several different computations such as a machine learning program and a computational biology program running at the same time, or from different stages, iterations, or layers of a complex computation. This consistency in scaling of performance with number of Processing Units is very encouraging.

As we attempt to run larger applications using the Kiva simulator we are hitting several limitations. These include:

(1) Running out of JVM heap space.
(2) Long simulation time.
(3) The need for garbage collection by the Java VM between successive simulation runs.

To overcome these limitations, we are developing a multi-host version of Kiva so that simulations may be run on conventional multi-processor systems and the memory and processing requirements can be met by distributing components of the target system over the available host processors. In addition, the Fresh Breeze compiler will be enhanced to support the parallelism available in stream processing and transaction processing applications.

We look forward to building an FPGA-based prototype system that models a Fresh Breeze multi-core processor using extended RISC-V Processing Units. This prospect is especially appealing given the availability of the BlueDBM facility in the Computation Structures Group of the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL).

REFERENCES

[1] Guy E. Blelloch. 1992. *NESL: A Nested Data-Parallel Language (Version 3.1).* Technical Report CMU-CS-92-103. Pittsburgh, PA, USA.

[2] J. B. Dennis. 2003. Fresh Breeze: A multiprocessor chip architecture guided by modular programming principles. *ACM SIGARCH Computer Architecture News* 31, 1 (2003), 7–15.

[3] J. B. Dennis. 2006. The Fresh Breeze model of thread execution. In *Workshop on Programming Models for Ubiquitous Parallelism.* IEEE. Published with PACT-2006.

[4] Jack B. Dennis. 2007. Compiling Fresh Breeze Codelets. In *Proceedings of Programming Models and Applications on Multicores and Manycores (PMAM'14).* ACM, New York, NY, USA, Article 51, 10 pages. https://doi.org/10.1145/2560683.2560691

[5] J. B. Dennis, G. R. Gao, and X. X. Meng. 2011. Experiments with the Fresh Breeze tree-based memory model. In *International Symposium on Supercomputing, Hamburg.*

[6] M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices* 33 (May 1998), 212–223.

[7] Dennis Jack, Lei Huang, Willie Lim, Hsiang-Huang Wu, and Yuzhong Yan. 2017. Implementing Deep Neural Networks on Fresh Breeze. (2017). Presented at Parco2017, Bologna, Italy.

[8] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15).* ACM, New York, NY, USA, 1–13.

[9] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. 1985. *Sisal: Streams and iteration in a single assignment language.* Technical Report M-146, Rev. 1. Lawrence Livermore National Laboratory, Livermore, CA.

[10] Gregory M. Papadopoulos and David E. Culler. 1990. Monsoon: An Explicit Token-Store Architecture. In *ISCA.* 82–91.

[11] Kent C. Redmond and Thomas Malcolm Smith. 1980. *Project Whirlwind; The History of a Pioneer Computer.* Butterworth-Heinemann, Newton, MA, USA.

[12] Tom St. John, Jack B. Dennis, and Guang R. Gao. 2012. Massively Parallel Breadth First Search Using a Tree-structured Memory Model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '12).* ACM, New York, NY, USA, 115–123. https://doi.org/10.1145/2141702.2141715

[13] S. J. Tullsen, D. M.; Eggers and H. M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture (ISCAJ '95).* IEEE, 392–403.