University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Caching Single-Assignment Structures to Build a Robust Fine-Grain Multi-Threading System

Wen-Yen Lin†
José Nelson Amaral
Jean-Luc Gaudiot†
Guang R. Gao

CAPSL Technical Memo 34
October 13, 1999

Copyright © 1999 CAPSL at the University of Delaware

†Department of Electrical Engineering, University of Southern California, Los Angeles, CA

Abstract

We present the design, implementation, and evaluation of single assignment data structures and of a software controlled cache in an existing multi-threaded architecture platform – the Efficient Architecture for Running Threads (EARTH). In EARTH, split-phase operations enable the tolerance of remote communication latency by a decoupling between the initiators and the receivers of communication/synchronization transactions. Under EARTH, the requester and receiver of a split-phase transaction are actually placed in separate threads.

We focus on split-phase memory operations under a global shared address space. A software-controlled cache is proposed to exploit the temporal and spatial locality of single-assignment structure memory references. Our experimental evaluation using a set of benchmarks indicates that the EARTH system with both the single-assignment storage and its caching mechanism is more robust. As a consequence the system can be ported to a wider range of machine platforms and deliver speedup for both regular and irregular application.

Keywords: Single Assignment Primitives, Split-Phase Transactions, Software Cache, Fine-Grain Multi-Threading, EARTH.

Contents

1 Introduction **1**

2 Fine Grain Multi-Threading **2**

3 Split Phase Communication and Synchronization **3**

4 Single Assignment Storage Structures **5**

5 I-Structure Software Cache **6**
5.1 Features of ISSC 7
5.2 ISSC Implementation on EARTH 8

6 Performance Measurements **10**
6.1 Highlights of Experimental Results 10
6.2 Latency and Bandwidth in the Hardware Platform 11
6.3 Description of Benchmarks 13
6.4 Robustness to Latency Variation 15

7 Related Work **18**

8 Concluding Remarks **18**

List of Figures

1 Split-phase transaction 4
2 State transition diagram for the I-structure Implementation 5
3 State transition diagram for an element in the I-structure software cache 7
4 Threaded-C with ISSC program example 9
5 Speedup in the MANNA machine 15
6 Absolute speedup with 10 μs communication interface overhead 16
7 Execution time with synthetically variable communication interface overhead . . 17

List of Tables

1 Latency in cycles (20ns), EARTH and ISSC operations on EARTH-MANNA-SPN 11
2 Measurements of bandwidth for the EARTH-MANNA-SPN implementation . . . 12
3 I-structure software cache hit ratios (%) 14
4 Average number of remote memory requests per node 14

1 Introduction

The design of a fine grain multi-threading system can be evaluated with many different metrics. What is the cost of switching between threads? How many instructions must a thread run, on average, in order to enable the implementation of an efficient latency hiding mechanism, and keep the processing units usefully busy? How capable is the system at automatically balancing the computation load across many processing nodes? How much latency can the system tolerate without suffering significant performance reduction?

Considerable effort is required for the design and implementation of a fine-grain multi-threading system. Therefore one would desire that such a system perform well in a class of hardware platforms. We say that a multi-threading system is *robust* if it performs well for both regular and irregular applications in a range of machines with varying latencies for remote operations. A robust system greatly benefits the development of high-performance applications because it enables the use of the same code in many hardware platforms. In this paper, we demonstrate that the addition of both a single-assignment structure (I-Structure) and a temporary storage for these structures (I-Structure Software Cache) to the fine-grain multi-threading systems improves the robustness of the system both in relation to latency tolerance and application characteristics.

The Efficient Architecture for Running Threads (EARTH) [15, 25] is an architecture and program execution environment that defines a fine-grain multi-threading model. The distinguishing characteristics of the EARTH threading model are the existence of two levels of threads (threaded function and fibers), the implementation of inter-node communication through *split phase* operations, and the definition of a light-weighted synchronization mechanism based on *synchronization slots*. Multi-threading programs for EARTH are written in Threaded-C [24], an explicitly multi-threaded extension of the C language. Threaded-C offers constructs that enable the definition of threaded functions and of fibers within the threaded functions. Split phase operations are coded in the program and are performed by a *synchronization unit*. This synchronization unit might be implemented as a special hardware module. It can be emulated by a second processor in the same processing node, or its functions might be emulated in software in the same processor that executes the threaded function's fibers [25].

We compare the performance of two extensions to the original programming environment in the EARTH system: (i) the programming environment is extended with an implementation of I-structures, a single-assignment data structure that facilitates the implementation of synchronizations across multiple processing nodes; (ii) the programming environment is extended with I-structures and an I-Structure Software Cache (ISSC) that enables the exploitation of temporal and spatial locality in the I-structures. The motivation to introduce both I-structures and ISSC to the original EARTH programming environment stems from the single-assignment nature of I-structures. When single-assignment storage structures are used, the need for consistency related transactions in the network is eliminated.

Our studies are based on an implementation of EARTH on the MANNA machine [14]. MANNA is a 20 node, 40 processor machine. The nodes are interconnected through a crossbar

switch network. In the EARTH-MANNA version that we use, the functions of the synchronization unit are emulated in the same processor that executes the fibers. Because neither MANNA nor the original programming environment for EARTH provide direct support for single-assignment storage such as I-structures, we emulate both the I-structure operations and the ISSC in software. Our study focuses on the robustness of the resulting programming model to latency tolerance.

The main contributions and results of this paper can be summarized as follows:

1. **I-Structures:** The introduction of I-structures to an imperative, explicitly multi-threaded language (Threaded-C) offers advantages in the number of synchronization operations required in the program.

2. **ISSC:** The combination of I-structures with a caching mechanism, as originally proposed [9], yields performance gains for all benchmarks that we tested.

3. **Experimental Study:** We measure the performance of EARTH, I-structures, and ISSC operations. We code four benchmarks — dense matrix multiplication, Hopfield network, conjugate gradient, and sparse matrix multiplication — and present a detailed performance study.

4. **Latency Tolerance Robustness:** The extension of the Threaded-C programming environment with ISSC is robust to latency variations. This robustness is reflected in better speedup curves for machines with higher costs for remote operations.

The remainder of the paper is organized as follows. Sections 2 and 3 discuss modern multi-threading system and their split-phase transactions. Section 4 presents our implementation of I-structures, and Section 5 presents the software cache design and implementation. Section 6 is a detailed performance study based on a set of real benchmarks. Section 7 presents related work.

2 Fine Grain Multi-Threading

Modern multi-threaded systems can be classified into two broad classes according to the granularity of the threads that they can efficiently support while yielding good performance: coarse grain multi-threading and fine grain multi-threading. Typically in a coarse grain multi-threading system (1) the thread switching mechanism involves interactions with the operating system; and (2) there is a limited number of light-weight processes to which threads must be bound. In a coarse grain multi-threading system, a thread can be viewed as a refinement of an operating system process. In contrast, in a fine grain multi-threading system: (1) the unit of computation is a collection of instructions grouped in a code block; (2) the system does not impose limits on the number of threads that can be active at the same time; (3) the system does

not require binding to any sort of limited resources; [1] and (4) the thread switching mechanism is quite efficient and does not involve the operating system, typically requiring only a small amount of state information to be saved in each switching. In a fine grain multi-threading system, a thread can be viewed as the coarsening of an instruction.

The fine grain multi-threading system studied in this paper, EARTH, is derived from the data-flow model of computation. In the classical strict data-flow model an instruction is enabled for execution when all its operands are available [12, 13, 16, 25]. To enforce this enabling condition, the instructions that produce operands must be able to send a synchronization signal to all the instructions that will consume their results. This model proved unwieldy for the implementation of machines based on current standard off-the-shelf hardware and compiler technology. In EARTH, the unit of computation is not an instruction, but a code-block formed by many instructions. An instantiation of the code-block running on a processing node is called a *fiber*, and multiple code-blocks are grouped into *threaded functions*. A successful program written in Threaded-C, the programming language for EARTH, will produce enough fibers to keep the local processor busy while remote computations and data fetching operations are performed.

In the next section we detail an split-phase transaction operation and in Section 4 we present the design of I-structures for Threaded-C. The main advantage of introducing I-structures to Threaded-C is the elimination of synchronization operations that would otherwise be required to ensure that reading operations were requested after the corresponding writing operations were performed.

3 Split Phase Communication and Synchronization

A cornerstone of the EARTH model is the mechanism that enables the superposition of local computation and remote operations: the *split-phase transaction*. Whenever an operation involves a long and/or unpredictable latency, the statement that requests that the operation be performed is issued in one fiber and the statement that depends on the result of the operation is issued on a different fiber. A dormant fiber receives synchronization signals from other fibers — executing either in the same processor or on a remote processor — through a synchronization slot.

The EARTH model assumes that each processing node has an Execution Unit (EU) that executes the fibers and a Synchronization Unit (SU) that is responsible for the following functions:

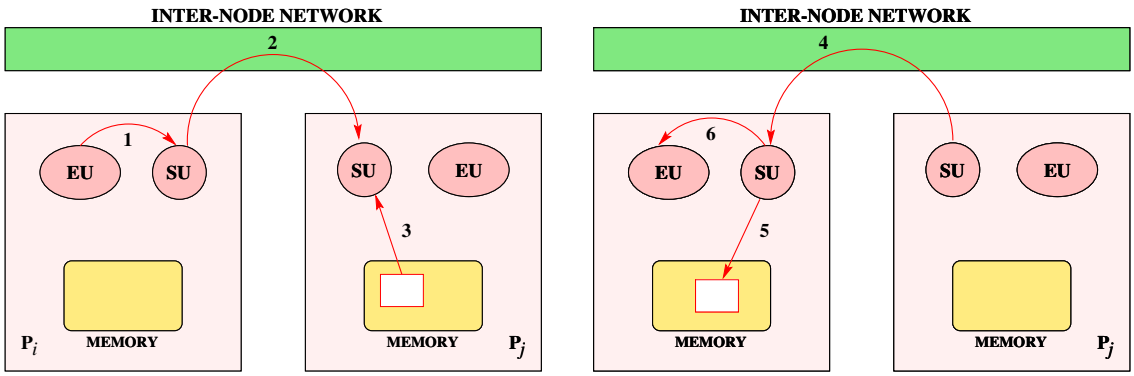
1. The emulation of a global address space;
2. The communications through the network;

[1] The only limitations on the number of active threads in a fine grain multi-threading system are caused by the memory space available to store active thread descriptors. If the data structure that holds these descriptors is stored in virtual memory, a very large number of active threads can indeed be supported.

3. The inter-fiber synchronization;

4. The implementation of a load balancing mechanism.

When the model is implemented on machines with a single processor per processing node, the functions of the SU are emulated in software by a Run Time System (RTS).



(a) (b)

Figure 1: Split-phase transaction

A typical split-phase operation, an EARTH block-move-sync operation, is illustrated in Figure 1. The steps 1-3 required for the completion of an split-phase transaction are illustrated in Figure 1(a). Figure 1(b) illustrates steps 4-6:

1. A fiber running on the execution unit of processor P_i issues a request that a block of data be copied from the memory of a processor P_j to its local memory. The requesting fiber may continue performing operations that do not depend on the arrival of the requested block, but will eventually terminate and allow the EU of processor P_i to run other enabled threads. The block move request must specify the source and destination addresses for the movement as well as the address of a synchronization slot that will receive a synchronization signal when the data transfer is complete.

2. The SU of P_i, having received the request for the block move, sends a block request to the SU of P_j through the network.

3. The SU of P_j reads the requested block from the local memory of P_j.

4. The SU of P_j sends, through the network, the requested block to the SU of P_i.

5. The SU of P_i writes the block into the destination address.

6. Finally, the synchronization slot indicated in the block move request receives a synchronization signal and causes the fiber that will use the transferred data to be spawned and executed in the EU of processor P_i.

In this example we assume that the destination of the block move and the synchronization slot that received the signal upon the completion of the data transfer were in the same processor that requested the data movement. However the EARTH model is general enough to allow each one of these addresses to be in a different processor.

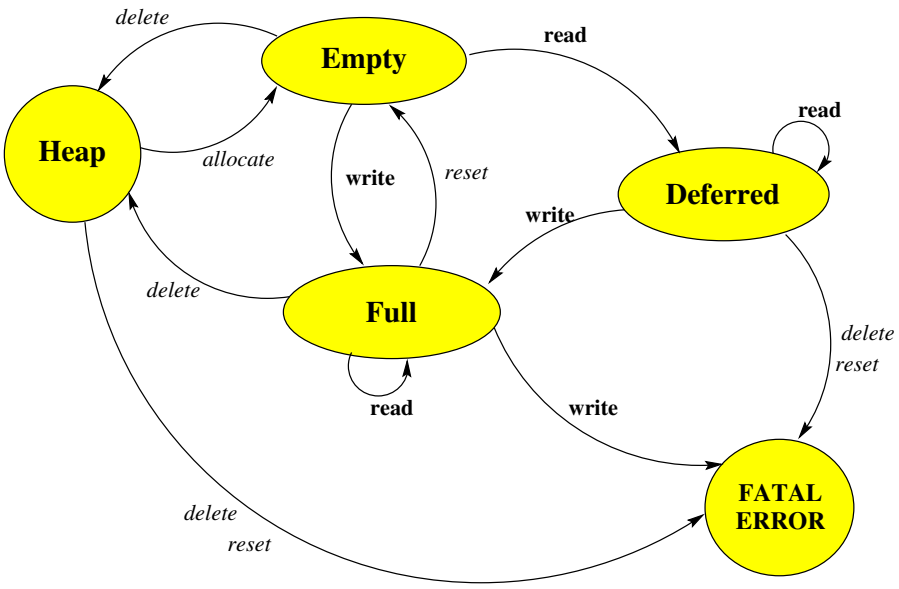


Figure 2: State transition diagram for the I-structure Implementation

Observe in the example presented in Figure 1 that the EU of processor P_j is never involved in the data transfer requested by processor P_i. Thus if two processing units are actually available in the machine to support the EU-SU model of EARTH, the only impact of the data movement on the execution of fibers in P_j would be possible conflicts on accesses to the memory between the SU and the EU. Moreover, during the steps (2) to (6) in Figure 1, the EU of P_i is not involved and is free to execute other enabled thread. The capacity to overlap the remote data transfer with the execution of other fibers in the EU is a distinguishing characteristic of a fine grain multi-threading system.

4 Single Assignment Storage Structures

Our motivation to introduce a single assignment structure in Threaded-C stems from the observation that the use of such structures significantly reduces the number of synchronization operations required in some programs. The single-assignment characteristic of I-structures eliminates the need for consistency related network operations when these structures are enhanced with temporary storage buffers. The former makes it easier to code problem solutions in Threaded-C, and the latter makes it easier to implement software caches for I-structures.

Although nothing prevents the implementation of I-structure with other data structure organizations, an I-structure was originally defined as an array of elements, where each element

of the array can be in one of three states: *empty*, *full*, and *deferred*. Each element of the array can only be written once, thus the name *single-assignment*, but it can be read many times. When the I-structure is created, all the elements of the array are empty. If a read occurs before the write, the element goes into the deferred state and the read operation is kept in a queue associated with that element. Subsequent reads are also queued. When a write to an empty element occurs, the value is written and the element becomes full. If the element was in the deferred state, all the reads that were queued for that element are serviced before the writing operation is complete, and the element goes into the full state. A read to a full element returns immediately with the value previously written. A write to a full element is considered a *fatal error* and causes the program to terminate. Figure 2 shows the state diagram of an I-structure. Notice that this state diagram includes the operations *delete* and *reset* that were not in the original definition of an I-structure. These operations eliminate the need for garbage collection, whose efficient implementation has been a long-standing open research problem [9]. Therefore the programmer must delete data structures after they are no longer needed. The reset operation allows reuse of I-structures avoiding frequent deletion/allocation in some applications.

Observe that in order for the I-structure to work correctly, the state transitions on the I-structure must be atomic. For instance if a write is performed in a deferred element, all reads in the queue of the deferred element must be served the value written before another operation to the same element can be performed. In the current implementations of Threaded-C this atomicity is derived from the fact that fibers are non-preemptive and that with a single processor in each processing node, only a single thread can run on a node at a time.

5 I-Structure Software Cache

Split-phase transactions for remote data memory accesses provide the ability to tolerate communication latency in a multi-threaded system. The data obtained through a split-phase transaction is managed by the programmer, and is not automatically cached by the system. Therefore if repeated requests for the same data are issued, they will be sent through the network to the source of the data requested.

In 1994, Dennis and Gao proposed the caching of elements of single-assignment data structures [9]. We designed and implemented an I-Structure Software Cache (ISSC) [19, 20, 21] to cache I-Structure elements on multi-threaded systems that support split-phased transactions. The ISSC takes advantage of the spatial and temporal localities of memory operations in I-Structure memory systems.

The single assignment property of the I-Structure memory system enables the implementation of the ISSC as a *software cache* without any hardware support. The ISSC intercepts all the read operations to the I-Structure. A remote memory request is sent out to the remote host only if the requested data is not available on the software cache of the local host. We explore the spatial locality in the references to the I-structure through a blocking mechanism. Instead of requesting a single element of the structure, an entire block of data including the requested

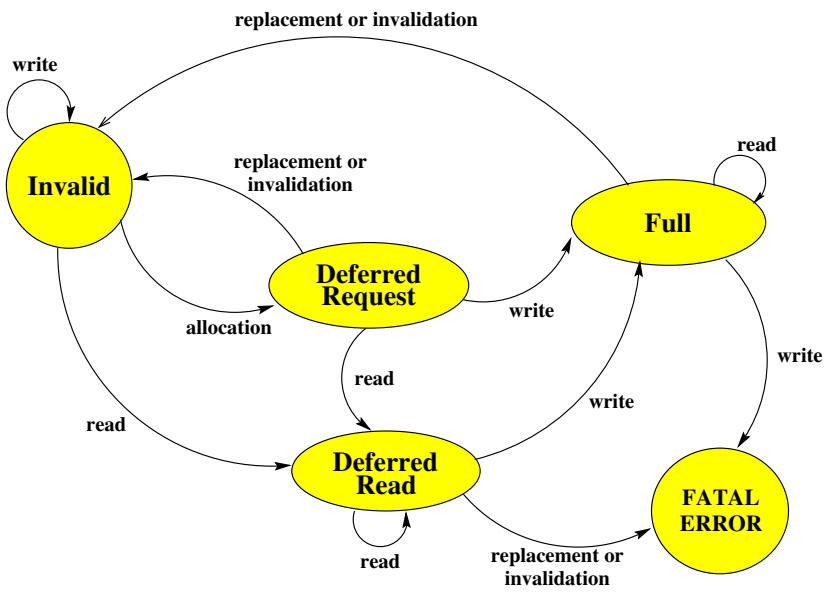


Figure 3: State transition diagram for an element in the I-structure software cache

element is requested to the node that hosts the I-structure.

The state transition diagram for an element of the I-Structure in the ISSC is shown in Figure 3. There is no space allocated in the ISSC for *invalid* elements. Space is allocated in the ISSC for an I-structure element in two situations: (1) the element itself is read; (2) another element in the same block is read. In (1), the element becomes a *deferred read* element; in (2) the element becomes *deferred request*. If a read to a *deferred request* element is issued, there is no need for a new request to be issued to the I-structure host node and the element goes into the *deferred read* state. Further read operations to a *deferred read* element are queued and do not cause further state transitions. When the value for a *deferred read* or *deferred request* element becomes available, the element goes into the *full* state. Read operations for *full* elements are serviced immediately and do not cause any state transition. A write to a *full* element is a fatal error. Both a *full* element and a *deferred request* element can be evicted from the ISSC either by a replacement operation or by an invalidation operation. A *deferred read* element is *irreplaceable*. An invalidation or a replacement of such an element is a fatal error. A write to an *invalid* element is ignored and the element is not placed in the cache.

5.1 Features of ISSC

The most relevant features of our I-Structure software cache implementation can be summarized as follows:

1. **Write-direct policy.** Our design adopts a simple *write-direct* policy. In other words, there is no caching for write operations. This policy prevents the node that issued the write from replying to a read request before the legality of the write is verified in the host node.

2. **Cache advance and deferred read sharing.** When a read miss occurs, a cache line is allocated in the software cache before the new request for the block of data containing the requested element is sent to the remote host. The original request is then queued locally in the corresponding element in this pre-allocated cache line. This new request is shared by all the empty data elements of the requested data block in the I-Structure by appending the request to all empty locations. A presence bit associated with each cache element provides a second level of data synchronization: subsequent requests for data elements in the same cache block can be deferred in the pre-allocated cache line even when the first request has not been replied.

3. **Centralized deferred requests and distributed deferred reads.** A simple "centralized" method is used for the implementation of the queues of deferred requests. The length of the queue of deferred requests for each element in the I-Structures is bounded by the number of nodes in the system. This is because only one request is sent from each node to the host node. Future deferred reads are kept locally in the node.[2]

5.2 ISSC Implementation on EARTH

The *ISSC* is implemented in the Threaded-C language [24] for EARTH systems [25, 15]. Our implementation of ISSC builds on the I-Structure user library [2, 1]. The layout of our software cache is the one of a set-associative cache. Set-associative software caches have faster cache entry searching time than fully associative caches and better cache utilization than direct mapped caches. The caching address consists of the node number of the host node, the I-Structure I.D. and the index of the element for which a read is requested. Upon receiving a read request, the caching address is mapped to a set by a hash function, and a software search is performed to see if there is a match for the address in the set. In our simulation studies [19, 20], we determined that a cache block of 8 data elements would yield reasonable cache hit ratio. Therefore, in our experiments discussed in section 6, we use a cache block size of 8 and implemented the software cache with 256 sets and 8 cache blocks within each set. That would be $16K$ elements in the cache.

A simple *Pseudo-Least Recently Used (LRU)* policy is adopted as the replacement policy in our implementation of ISSC. A cache block that has any element in the *deferred read* state is irreplaceable. A single reference bit is attached to each cache block. The reference bit is set whenever there is the block is read. Within a set, a victim pointer is used to select a block to be replaced. When a cache replacement is needed, the block pointed by the victim pointer is tested to verify if it is replaceable and if its reference bit is zero. If either condition is not satisfied, the reference bit of the element is reseted and the victim pointer is advanced. When a replaceable block with a zero reference bit is found, it is replaced and the victim pointer is advanced to the next block.

[2] In the rare situation in which all the lines in the ISSC are irreplaceable, reads bypass the ISSC and are sent directly to the host node. This is a distinction with the caching mechanism proposed in [9], where the read processing would be postponed.

```
1:  #include <stdio.h>
2:  #define EXTERN
3:  #include "issc.h"
4:  #define   I_NODE   NUM_NODES-1
5:
6:  THREADED ARRAY_INIT(int i_node, int i_id, int length)
7:  {
8:     int i;
9:     for(i=0; i < length; i++)
10:       INVOKE(i_node, I_WRITE_F, i_id, (float)i);
11:    END_FUNCTION();
12: }
13:
14: THREADED MAIN()
15: {
16:    SLOT SYNC_SLOTS[3];
17:    int F_str, i;
18:    float F_variable[8];
19:
20:   INIT_SYNC(0,NUM_NODES+1,NUM_NODES+1,1);
21:    INVOKE(I_NODE, I_INIT, SLOT_ADR(0));
22:    /* Allocate cache space on each node */
23:    for(i=0; i<NUM_NODES; i++)
24:       INVOKE(i, InitCache, SLOT_ADR(0));
25:    END_THREAD();
26:
27:  THREAD_1:
28:    INIT_SYNC(1, 1, 1, 2);
29:    /* Allocate I-Structure */
30:    INVOKE(I_NODE, I_ALLOCATE, 8, TO_GLOBAL(&F_str), SLOT_ADR(1));
31:    END_THREAD();
32:
33:  THREAD_2:
34:    TOKEN(ARRAY_INIT, I_NODE, F_str, 8);
35:    INIT_SYNC(2, 8, 8, 3);
36:    /* Read from F_str[0:7]*/
37:    for ( i=0; i<8; i++)
38: #ifdef CACHE
39:          INVOKE(NODE_ID, SC_I_READ, I_NODE, F_str, i, F,
40:                TO_GLOBAL(&F_variable[i]), SLOT_ADR(2));
41: #else
42:          INVOKE(I_NODE, I_READ_F, F_str, i,
43:                TO_GLOBAL(&F_variable[i]), SLOT_ADR(2));
44: #endif
45:    END_THREAD();
46:
47:  THREAD_3:
48:    for(i=0; i<8; i++) printf("%f ", F_variable[i]);
49:    RETURN();
50: }
```

Figure 4: Threaded-C with ISSC program example

A simple example to show how the ISSC library is used in a Threaded-C program is shown

in Figure 4. In this example, an I-Structure floating point array of length 8 is allocated on the last node of the system. The data of these I-Structure elements are then generated by a node, and node 0 reads back the value of those 8 data elements.

In line 3, the header file, *issc.h*, which defines the data structure of ISSC is included. In line 4, we define the I-Structure host, I_NODE, as the last node of the system, NUM_NODES-1. In Thread_0 (lines 20-25), we initialize the I-Structure in I_NODE (line 21) and software caches on each node (line 24). In Thread_1 (lines 27-31), a floating point I-Structure array of 8 elements is allocated in I_NODE (line 30). The handle for the allocated I-Structure is stored in F_str. In Thread_2 (lines 33-45), the data of I-Structure array *F_str* are generated by the *ARRAY_INIT* function (line 6-12) which is invoked by the *TOKEN* function in line 34. Then, data of this I-Structure array are read back in line 35-44. In line 38, we use a compiler flag to activate/deactivate the ISSC. If the *CACHE* flag has been defined in the compilation, the function *SC_I_READ* is invoked locally (at node NODE_ID) to read an I-Structure element located in the I-Structure host node *I_NODE* through the utilization of ISSC, otherwise, the function *I_READ_F* without utilizing ISSC is invoked in the I-Structure host (node I_NODE). After all 8 data elements are read back from node I_NODE, THREAD_3 is activated, prints out the data, and terminates the program. In this program, if ISSC is not used, 8 data requests for the I-Structure array F_str are sent to the remote node, I_NODE, by invoking eight I_READ_F functions on I_NODE. However, if ISSC is used, even though eight SC_I_READ functions are invoked locally, only one data request is sent to the remote node I_NODE.

6 Performance Measurements

This section presents a detailed experimental study comparing the performance of four benchmarks with and without I-Structure Software Cache.

6.1 Highlights of Experimental Results

To study the effectiveness of our implementations of both I-structures and ISSC, we coded four benchmarks (see Section 6.3) in both versions of the system (Threaded-C with I-Structure and Threaded-C with I-Structures and ISSC). Our experimental results were obtained on the MANNA machine. We also measured the latency for basic EARTH operations and for I-structure based operations. Our main results can be summarized as follows:

1. **The addition of ISSC to the EARTH system results in increased robustness to latency variation.** The speedup obtained with ISSC increases for machines with higher costs for remote operations (see Figures 5 and 6 for details).

2. **ISSC significantly reduces the amount of traffic in the network.** As shown in Table 4 in all applications the number of remote requests for I-structure elements was reduced from one up to four orders of magnitude.

Operation	Sequential		Pipelined	
	Local	Remote	Local	Remote
Sync Thread	116	199	42.0	49.7
Spawn Thread	113	213	—	—
Get_Sync	141	348	56.8	94.0
Data_Sync	138	333	53.0	90.7
Fun. Call (1)	250	451	159	140
Fun. Call (18)	410	628	276	223
I_READ_F	317	492	296	318
ISSC hit	405	479	341	413
ISSC miss	2690	3027	2958	2577
ISSC deferred	—	—	1143	1238

Table 1: Latency in cycles (20ns), EARTH and ISSC operations on EARTH-MANNA-SPN

3. **The sole addition of I-Structures (without ISSC) decreases the performance of the EARTH system.** Even for machines with higher latencies, the overhead of the software emulation of I-structures hurts performance (as shown in the graphs of Figure 7) unless it is offset by the benefits of ISSC (see Section 6.3 for details).

4. **ISSC operations can be implemented very efficiently in the MANNA machine.** In 6.2 we demonstrate that the MANNA machine network interface is very efficient. Our experiments demonstrated that our implementation of ISSC on top of the EARTH operations is also efficient (see Table 1).

5. **The performance of the system with ISSC improves for all benchmarks for machines with moderately high latency for remote operations.** As shown in Figure 6 for all four benchmarks if 10 μs (500 cycles) are added to the latency of MANNA (which is 3.5 μs = 175 cycles), the benchmarks running on the platform with ISSC produces greater speedup over the system with I-structures only.

6.2 Latency and Bandwidth in the Hardware Platform

The MANNA machine is a research platform of which only a few were constructed. With full control of the network interface in MANNA, the implementations of inter-node communication and synchronizations are very efficient as demonstrated by the measurements presented in this section. We measure the latency of some EARTH operations, in terms of the number of processor cycles, for the EARTH-MANNA-SPN machine. EARTH-MANNA-SPN is an implementation of the EARTH model on the MANNA machine in which the functions of both the SU and the EU are performed by the same processor [25].

Table 1 displays the latency of EARTH operations in the MANNA platform used in this comparative study. In the measurements: (1) in the "sequential" version the next EARTH

Block Size	Bandwidth		
	ns/byte	Mbytes/sec	% of Peak
5000 bytes	29.3	34.1	68.2
50000 bytes	28.25	35.5	71.0

Table 2: Measurements of bandwidth for the EARTH-MANNA-SPN implementation

operation is issued after the receipt of a synchronization signal confirming that the current operation is completed; (2) in the "pipelined" version, all operations are issued and when they are completed a synchronization signal is received to stop the clock. In a local measurement all operations are within a processor, while in a remote measurement synchronizations across the network are issued. Observe that the processor is not busy with the operation for the number of clock cycles shown in Table 1. Most of the remote operation time is spent either waiting on queues or in the network, thus releasing the processor to execute other ready fibers.

The other EARTH operations measured in Table 1 include the direct spawning of a fiber; a `get_sync` operation in which fiber 1 requests a word of data from fiber 2 and fiber 2 synchronizes fiber 1 when the data arrives; a `data_sync` operation in which fiber 1 sends a word of data to fiber 2 and fiber 2 synchronizes fiber 1 when the data arrives; and function calls with 1 and with 18 parameters, which represent the invocation of a threaded function either in the same node or in a remote node.

At the bottom of Table 1 are the measured latency of ISSC operations and of the basic I-Structure read function, I_READ_F. Note that ISSC deferred is the case that a read hits the cache block which has been requested and no data is available at the moment. The difference between sequential local and sequential remote cases of I_READ_F operation denotes the communication interface overhead. The remote operation takes only 175 processor cycles (3.5 μs) more than the local one to be completed. This measure indicates that the inter-node communication in MANNA machines is very efficient when compared with networks of workstations. The measurements in Table 1 are taken for completeness. Operations such as local hits and misses will never occur.

Table 2 presents the results for the measurement of the bandwidth across processing nodes on the MANNA machine. The measurement was taken between node 0 and node 19 according to the following procedure:

1. Node 0 starts the clock;
2. 1000 BLKMOV_SYNCs are issued from node 0 to node 19;
3. After receiving 1000 sync signals, node 19 synchronizes node 0;
4. Upon receiving the sync signal, node 0 stops the clock.

This procedure is repeated twenty times and the results reported in Table 2 are averaged over the twenty runs. Both the latency and the bandwidth numbers reported attest for the

efficiency of the implementation of EARTH on MANNA. Observe that both the latency and bandwidth measurements include all the software overheads involved in the communication. Theobald compares the bandwidth utilization of MANNA with those in a survey of parallel machines [10] and MANNA is one of the few systems with very high bandwidth utilization rate [25].

6.3 Description of Benchmarks

To measure the improvement in the system performance when both I-structures and ISSC are used, we selected four different benchmarks: dense matrix multiplication, Conjugate Gradient, Hopfield network, and sparse matrix multiplication. To compare the performance of the software cache with the original system, we implemented two versions of codes for each benchmark:

1. A Threaded-C code using the I-Structure library, *Threaded-C+IS*;
2. A Threaded-C code using both the I-structure library and the I-Structure Software Cache (*ISSC*), *Threaded-C+ISSC*.

All our experimental results were performed in the MANNA machine. The two processors of a processing node on MANNA share 32 Mbytes of DRAM. The nodes of MANNA are diskless, therefore all the code, runtime system, data, and the software emulations of the I-structure and the ISSC must fit in one node's memory. Therefore we were only able to test moderate data set size for the benchmarks. In a related research work, Theobald developed a detailed cycle-by-cycle simulation of the MANNA architecture and demonstrated that applications scale well for larger versions of the platform [25].

Dense Matrix Multiplication: Two 128x128 dense matrices are multiplied. The algorithm that we use in this study is a simple minded, non-blocking algorithm that computes $C = A \times B$. The computation of rows in the resulting matrix C is evenly distributed among all nodes. Node 0 invokes threads on each node to compute the rows that it is responsible for. The results of C elements are written directly to where they reside.

Conjugate Gradient: The Conjugate Gradient algorithm from the NAS benchmark suite [5] uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non-zeros. In our experiment, the problem size is 256 linear equations with 256 unknown variables. Calculations of matrix-vector multiplications are done in parallel across all the nodes and the calculations of vector-vector multiplication are done on node 0. In this algorithm, most of the computation consists in updating array elements. Therefore, the benchmark does not have much temporal data locality.

Hopfield Network: Hopfield is a kernel benchmark [2] based on the Hopfield Network. It is a recursive neural network that is often used in combinatorial optimization problems as well

Number of Nodes	Benchmarks			
	Dense M.M.	C.G.	Hopfield	Sparse M.M
2	99.71	93.70	99.90	99.92
4	99.52	93.69	99.80	99.87
8	99.13	93.52	99.61	99.76
16	98.35	92.92	99.22	99.53

Table 3: I-structure software cache hit ratios (%)

Number of Nodes	Benchmarks							
	Dense M.M.		C.G.		Hopfield		Sparse M.M	
	no ISSC	ISSC	no ISSC	ISSC	no ISSC	ISSC	no ISSC	ISSC
2	528384	1536	33536	2112	32768	32	986668	761
4	396288	1920	25152	1587	24576	48	731842	971
8	231168	2016	14672	950	14336	56	426002	1038
16	123840	2040	7860	557	7680	60	227979	1078

Table 4: Average number of remote memory requests per node

as an associative memory. The network is formed by a set of neurons that are connected by synapses. At time $k+1$, the activation value of each neuron is updated based on the activation values of neurons at time k weighted by synapse values. In the I-Structure and ISSC implementation, two I-Structure arrays are used to store the current and previous activation values of neurons. Before updating to the current value, the I-Structure array is reset and reassigned with a new I.D., therefore, the same memory space can be re-utilized and no cache flush would be needed. The problem size we tested is 256 neurons.

Sparse Matrix Multiplication: Sparse matrix multiplication is an application with irregular data access pattern. Two unstructured sparse 256x256 matrices A and B are randomly generated with density of 10%. Matrix A is then stored in Compress Row Storage (CRS) format and matrix B is stored in Compress Column Storage (CCS) format. A dense resulting matrix C is generated by multiplying A and B.

Table 3 shows the cache hit ratios of the four benchmarks in our experiments and Table 4 shows the average number of remote memory requests in each benchmark both with and without ISSC. ISSC did help the system to exploit global data locality. For three of the four benchmarks (except conjugate gradient), more than 99% of cache hit ratios could be achieved, and even in conjugate gradient algorithm which has poor temporal data locality, 93% of cache hit ratio could be achieved. Table 4 shows that ISSC reduces the number of remote memory requests actually sent to remote nodes. In all the cases, at least 93% of the original remote memory requests are eliminated out by the I-Structure Software Cache.

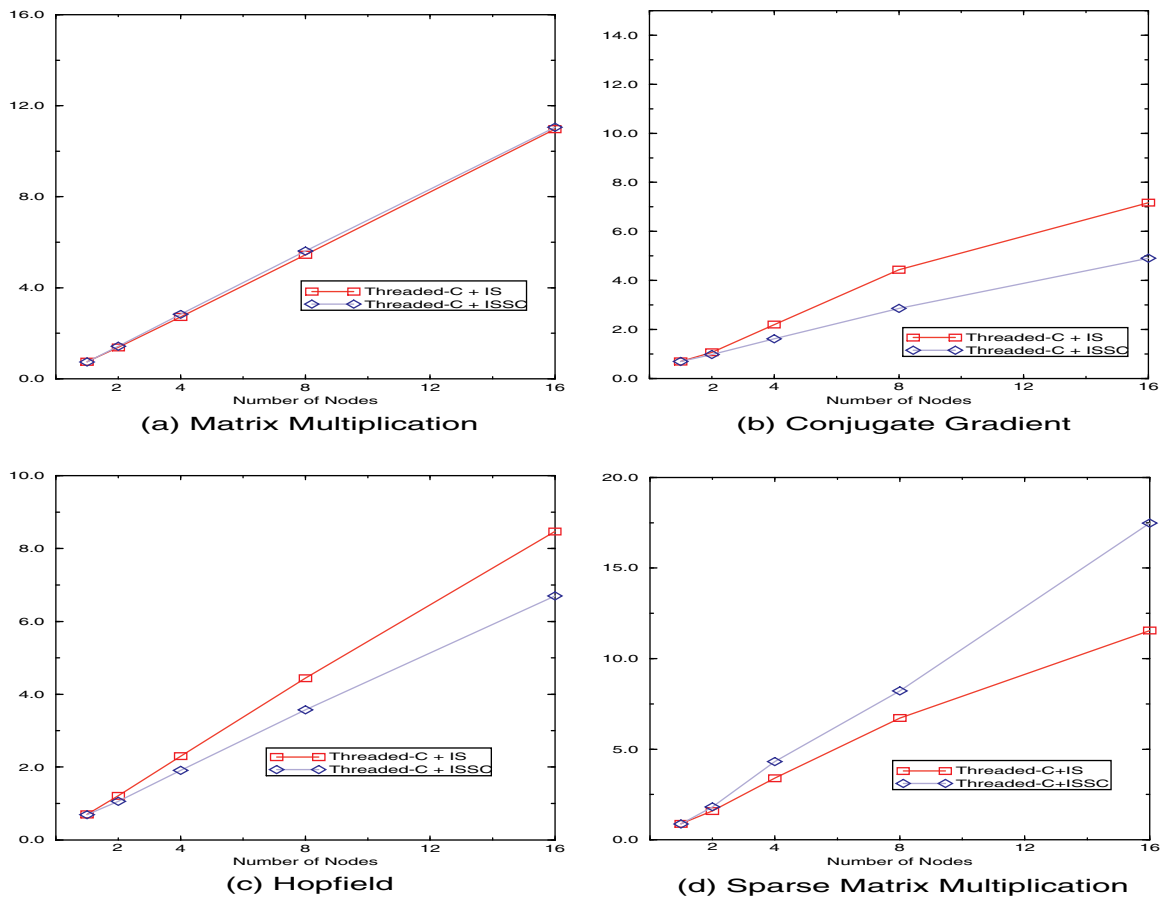


Figure 5: Speedup in the MANNA machine

6.4 Robustness to Latency Variation

The speedups reported in Figures 5, 6, and 7 are computed in relation to a version of the same benchmark written in plain Threaded-C and running on a single processing node. We performed two sets of experiments. The first set, shown in Figure 5 measures the performance on the MANNA machine. When the cost to execute split-phase operations is very low, the difference in speedup between the versions with and without ISSC is not very significative. In the Conjugate Gradient and Hopfield benchmarks, the *Threaded-C+IS* version has better performance than the *Threaded-C+ISSC* version. This is because of the poor temporal data locality in the algorithms which results in the high ratio of deferred cache hits in the cache hit situation. The overhead of deferred hits is much larger than the I-Structure access and the ISSC hit as reported in the beginning of this section.

Indeed in the measurements presented in Section 6.2 we show that the network implementation of MANNA is very efficient. However, this efficiency is usually not available in affordable and widely available networks of workstations. The sending and receiving of network packages

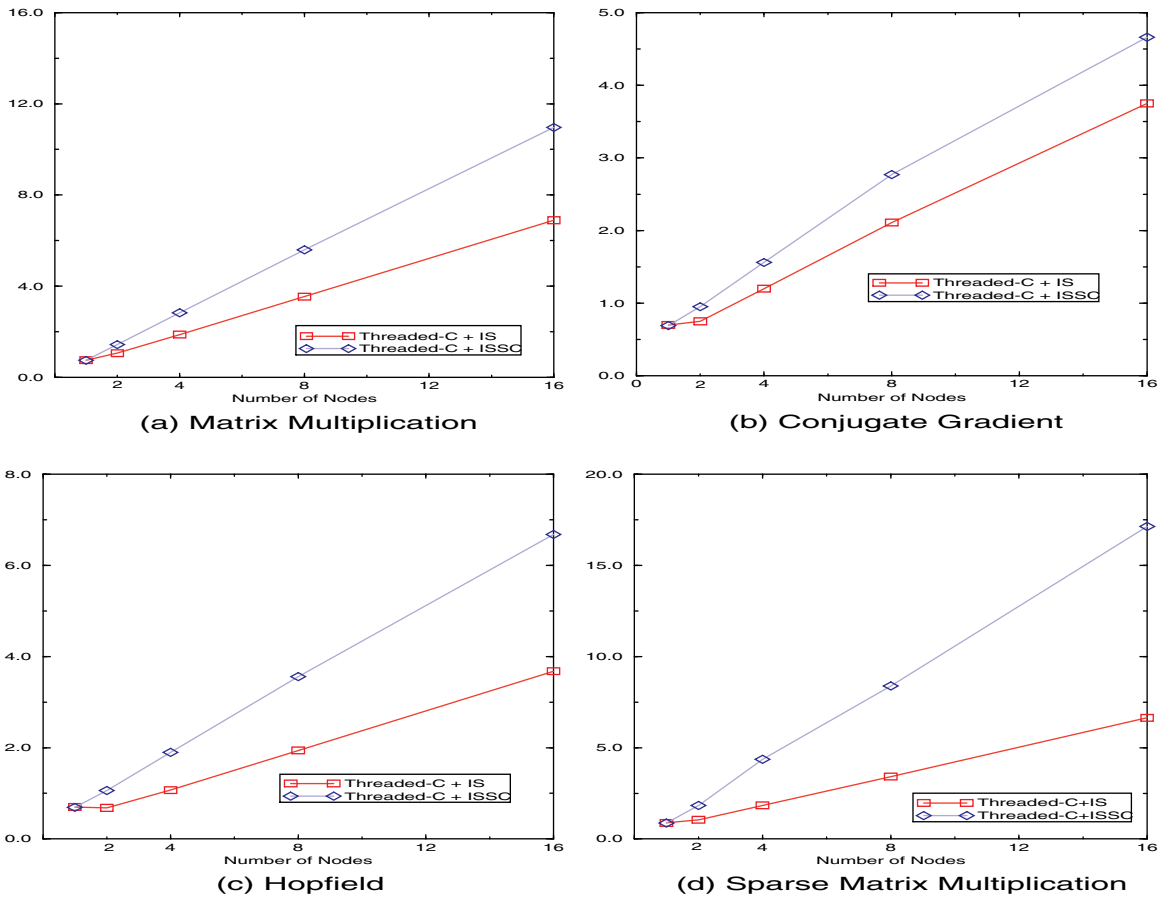


Figure 6: Absolute speedup with 10 μs communication interface overhead

may take from hundreds to thousands of cycles depending on the design of the network interface [7]. In some machines, a parallel environment is built on top of the TCP protocol and the communication interface overhead may be as high as hundreds of micro-seconds [17]. Even with some improved protocols, like Fast Sockets [23] and Active Messages [26], it still costs 40$\sim$60 micro-seconds to send a message to the network.

Thus, in our second set of experiments, shown in Figure 6, we add 10 μs (500 processor cycles) to the cost of both I-structure and ISSC operations. This is equivalent to a machine with a higher communication/computation cost ratio, i.e., a machine in which requesting a remote operation is more costly. In this set of experiments, we observe a significant speedup of the *Threaded-C+ISSC* version over the *Threaded-C+IS* version for all benchmarks. Notice that even with the addition of 10μs, the remote operation cost is far less than the communication interface overhead of fast local area network [23], which cost 40$\sim$60 micro-seconds. In this experiment, we show that in spite of the overhead associated with the software emulation of ISSC, by taking advantage of the global data locality in applications and reducing the number of requests sent in the network, the I-Structure Software Cache makes the system more robust

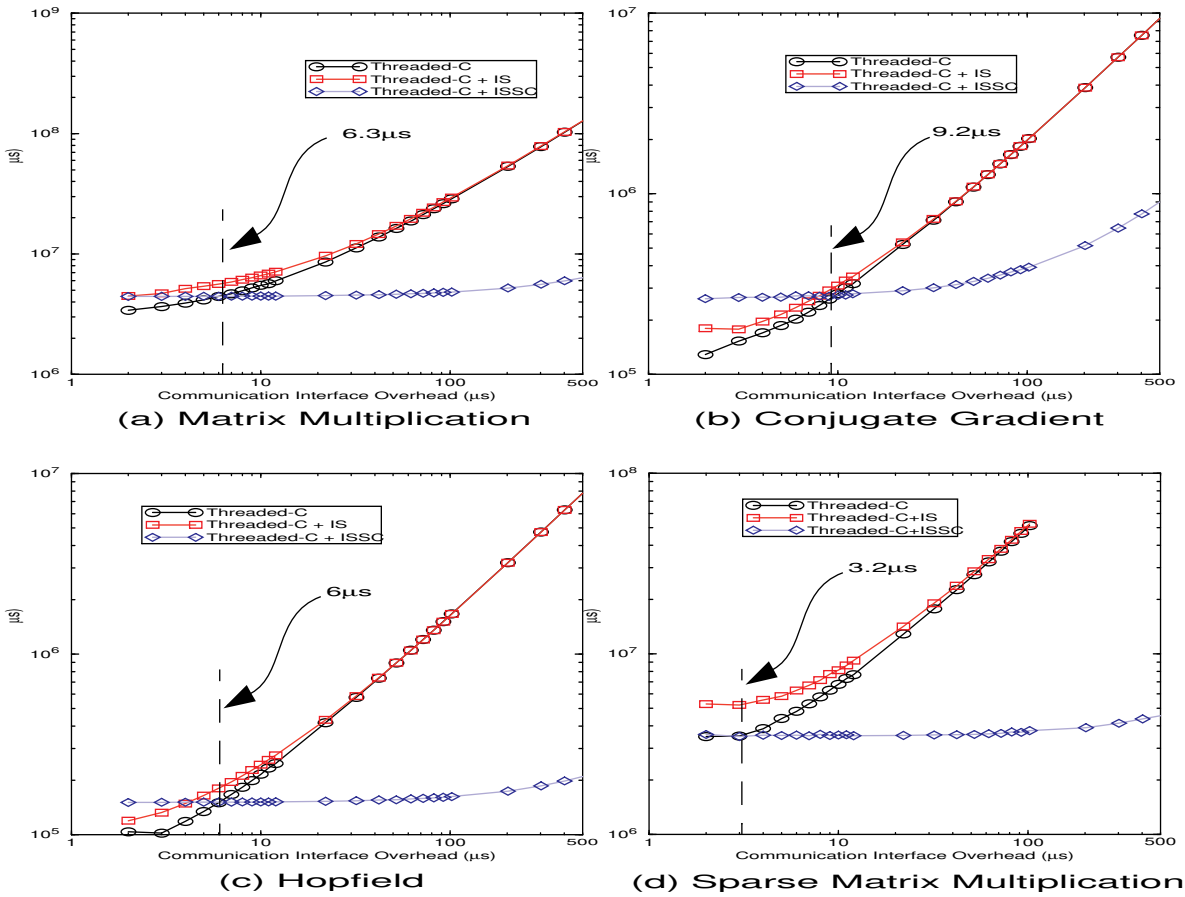


Figure 7: Execution time with synthetically variable communication interface overhead

to increased remote cost operations.

From the results presented in Figure 5 and 6, we can observe that the communication interface overhead is a determinant factor in the performance of I-Structure Software Caches. To have a better understanding of the relationship between the ISSC performance and the cost of remote operations, we ran our experiments on a 16 node system with a variable synthetic communication overhead. Figure 7 shows the execution time of applications as the communication overhead increases. For each benchmark, the graph shows the point where the *Threaded-C+ISSC* version starts to out-perform the *plain Threaded-C* implementation. ISSC starts to help the system when the communication interface overhead is greater than 6.3μs, 9.2μs, 6μs and 3.2μs respectively in dense matrix multiplication, conjugate gradient, Hopfield and sparse matrix multiplication. When the communication interface overhead exceeds 100μs, the *Threaded-C+ISSC* versions run almost 10 times faster than the *plain Threaded-C* versions in our benchmarks.

7 Related Work

The name *I-structure* was originally used by Arvind and Thomas in the context of functional languages to designate an array built with a fine-grained update operator with no repeated indexes [4]. Later, I-structures were proposed as separate data structures for functional programs. Arvind, Nikhil, and Pingali demonstrated, through several programming examples, that the introduction of I-structures in functional languages eliminates inefficiencies and increases the programmability of functional languages [3]. The proposition to incorporate I-structures in functional languages was derived from the observation that without the ability to store a *state*, it is very difficult to solve even simple problems in a manner that is efficient, easy to code, and enables the exploitation of parallelism [3].

Dennis and Gao discussed the implementation of caching capabilities for single-assignment structures in distributed memory systems [9]. Culler *et al.* [8] implemented software I-Structure caching in the *Id90* compiler for the Threaded Abstract Machine (TAM) implemented on the CM-5. In their implementation, a cache block is formed by a single I-Structure element. Therefore, only temporal data locality could be exploited and no deferred read sharing problem occurs. Our experiments show that exploiting spatial data locality significantly improves performance.

Kavi *et al.* proposed a hardware supported cache memories for the Explicit Token Store (ETS) model of data-flow systems with an I-Structure memory system [18]. In their design, a cache line is allocated only when the data is brought back from the remote host. Therefore, in a read miss situation, the request is forwarded to the I-Structure directly without any checking on the local IS-Cache. Multiple requests to the elements in the same block are sent over the network. A write-back cache is adopted in their design. To guarantee the service of pending requests, data elements are written back to the I-Structure when there is any pending request. To implement this mechanism, a missing table is maintained in the producer's IS-Cache to indicate the pending status of the I-Structure elements and extra interrogation messages are passed on the network.

8 Concluding Remarks

The EARTH is a fine-grain multi-threaded architecture and execution environment. Inter-node communication is performed through *split phase* operations to overlap communication latencies with useful computations. We presented the designs and implementations of I-Structures and I-Structure Software Cache (ISSC) using the EARTH Threaded-C language. The inclusion of I-Structures in Threaded-C eliminates the need for synchronization operations that would be required to ensure that read operations are issued before the corresponding writing ones.

ISSC caches values obtained through split-phase transactions in the operation of an I-Structure. It also exploits spatial data locality by clustering individual element requests into blocks. Our experimental results show that the inclusion of ISSC in a parallel system that provides split-phase transactions reduces the number of remote memory requests and hence

reduces the traffic in the network. The resulting EARTH system is more robust to variations in the latency incurred by remote operations.

In this paper, we demonstrated that even a software emulation of an I-structure cache can yield performance improvements. We also demonstrated that such performance gains will be even more evident in machines with higher latencies, such as network of workstations [17, 23]. The concept of I-Structure caches is not limited on software implementation. While some researchers concentrate on the development of faster network interfaces [6, 11, 22], I-Structure caches can be implemented with dedicated hardware, a decoupled general purpose processor or even integrated into the design of network interface.

Acknowledgements

This research is supported by the National Science Foundation under grant award MIP-9707125 and INT-9815742. We would like to acknowledge all current and former members of CAPSL that designed, developed, and maintain the EARTH platform and its machines.

References

[1] J. N. Amaral, G. Gao, and X. Tang. An implementation of a Hopfield network kernel on EARTH. In *X Brazilian Symposium on Computer Architecture and High Performance Processing*, pages 223–232, Buzios, RJ, Brazil, Sept. 1998.

[2] José Nelson Amaral and Guang R. Gao. Implementation of I-structures as a library of functions in Portable Threaded-C. CAPSL Technical Note 04, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, June 1998.

[3] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[4] Arvind and R. E. Thomas. I-structures: An efficient data structure for functional languages. Technical Report MIT/LCS/TM-178, Massachusetts Institute of Technology, Cambridge, 1981. MIT Lab. for Computer Science.

[5] D. H. Bailey, J. T. Barton, T. A. Lasinski, and H. D. Simon. The NAS Parallel Benchmarks. Technical Report NASA TM 103863, NASA Ames Research Center, July 1993.

[6] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 22(2), April 1994.

[7] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–12, San Diego, California, May 19–22, 1993. *SIGPLAN Notices,* 28(7), July 1993.

[8] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. Empirical study of a dataflow language on the CM-5. In Guang R. Gao, Lubomir Bic, and Jean-Luc Gaudiot, editors, *Advanced Topics in Dataflow Computing and Multithreading*, pages 187–210. IEEE Computer Society Press, 1995. Book contains papers presented at the Second International Workshop on Dataflow Computers, Hamilton Island, Australia, May 24–26, 1992, held in conjunction with the 19th Annual International Symposium on Computer Architecture.

[9] Jack B. Dennis and Guang R. Gao. On memory models and cache management for shared-memory multiprocessors. ACAPS Technical Memo 90, School of Computer Science, McGill University, Montréal, Québec, December 1994. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[10] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1998.

[11] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Proceedings of the Hot Interconnects Symposium V*, August 1997.

[12] G. R. Gao. An efficient hybrid dataflow architecture model. *Journal of Parallel and Distributed Computing*, 19(4):293–307, December 1993.

[13] Guang R. Gao, Herbert H. J. Hum, and Yue-Bong Wong. Parallel function invocation in a dynamic argument-fetching dataflow architecture. In *Proceedings of PARBASE-90: International Conference on Databases, Parallel Architectures, and Their Applications*, pages 112–116, Miami Beach, Florida, March 7–9, 1990. IEEE Computer Society.

[14] W. K. Giloi, U. Bruening, and W. Schroeder-Preikschat. MANNA: Prototype of a distributed memory architecture with maximized sustained performance. In *Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing*, Braga, Portugal, January 24–26, 1996.

[15] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. A design study of the EARTH multiprocessor. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[16] Herbert Hing-Jing Hum. *The Super-Actor Machine: a Hybrid Dataflow/von Neumann Architecture.* PhD thesis, McGill University, Montréal, Québec, May 1992.

[17] K. Keeton, T. Anderson, and D. Patterson. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Hot Interconnects III: A Symposium on High Performance Interconnects*, August 1995.

[18] Krishna M. Kavi, A. R. Hurson, Phenil Patadia, Elizabeth Abraham, and Ponnarasu Shanmugam. Design of cache memories for multi-threaded dataflow architecture. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 253–264, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[19] Wen-Yen Lin and Jean-Luc Gaudiot. I-structure software cache: A split-phase transaction runtime cache system. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 122–126, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.

[20] Wen-Yen Lin and Jean-Luc Gaudiot. Exploiting Global Data Locality in Non-Blocking Multithreaded Architectures. In *Proceedings of the Third International symposium on Parallel Architectures, Algorithms and Networks*, Dec. 1997.

[21] Wen-Yen Lin and Jean-Luc Gaudiot. The Design of An I-Structure Software Cache System. In *Workshop on Multithreaded Execution, Architecture and Compilation, 1998. Held in conjunction with HPCA-4*, Feb. 1998.

[22] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 24(2), May 1996.

[23] S. Rodrigues, T. Anderson, and D. Culler. High-Performance Local Area Communication With Fast Sockets. In *USENIX 1997 Annual Technical Conference*, Jan 1997.

[24] Kevin B. Theobald, José Nelson Amaral, Gerd Heber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the Threaded-C language. CAPSL Technical Memo 19, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[25] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill University, Montréal, Québec, May 1999.

[26] Thorsten von Eicken, David E. Culler, Sech Copen Goldstein, and Klaus Eric Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.