

# Braids and Fibers: Language Constructs with Architectural Support for Adaptive Response to Memory Latencies

David F. Bacon  
dfb@watson.ibm.com

Xiaowei Shen  
xwshen@us.ibm.com

IBM T.J. Watson Research Center  
P.O. Box 704  
Yorktown Heights, NY 10598

## ABSTRACT

As processor speeds continue to increase at a much higher exponential rate than DRAM speeds, memory latencies will soon exceed 1000 cycles.

With such non-uniform access times, the flat memory model that was made practical by deeply pipelined superscalar processors with multi-level cache memories will no longer be tenable due to the inexorable effects of Amdahl's Law. The most common approach to this problem is hardware multi-threading, but multi-threading requires either abundant independent applications, or well-parallelized monolithic applications, and neither is easy to come by.

We present *braids* and *fibers*, high-level programming constructs which facilitate the creation of programs that are *partially ordered*. These partial orders can be used to respond adaptively to hardware latencies. We show how these constructs can be effectively supported with very simple and inexpensive instruction set and micro-architectural extensions.

Braiding is much simpler than parallelizing, but yields many of the same benefits. We have developed braided versions of a number of important algorithms, including quicksort and the mark phase of a garbage collector. The braided code is easy to understand at the source level, yet can be translated into highly efficient code using our hardware extensions.

## 1. INTRODUCTION

In 1995, Wulf and McKee coined the term "memory wall" [12], and asked the question "Is it time to forego the model that access time is uniform to all parts of the address space?" We believe the answer is yes.

High performance uni- and multi-processors have for many years been designed to present a flat memory abstraction to the programmer. This greatly simplifies the programming model, and works well as long as caches are able to hide memory system latency.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.*

PAC<sup>2</sup>, October 2004, Yorktown Heights, New York  
Copyright © 2004 IBM Corporation.

However, this abstraction is beginning to break down due to current technological trends in which the memory hierarchy is becoming ever deeper while the relative speed of the processor to the memory continues to increase. Latencies for main memory accesses will soon exceed 1000 cycles.

Techniques like prefetching have been used successfully to hide significant amounts of latency, but prefetching only works well for highly predictable programs.

Ultimately, as the performance of the memory system becomes more and more non-uniform, such approaches to *tolerating* latency will no longer function. One needs ways to adaptively *avoid latency* and do other work.

If one goes to a restaurant and there is a line, one is given an estimate of the length of time one will have to wait. If one calls a company and all their operators are busy, one receives an automatic estimate of the amount of time one will have to remain on hold.

Explained in these practical, real-world contexts, it seems absurdly obvious that one ought to find out how long a variable, potentially lengthy delay will take. So why isn't this obvious in the context of memory hierarchy design?

We argue that what is obvious in the restaurant business should be obvious in the computer business. We present a high-level programming model that allows programs to respond to long latencies by performing other work while high-latency operations are in progress.

The fundamental approach is to divide the program into *fibers*, which are sections of sequential code that can be interleaved in a *partial order*. Although fibers are partially ordered with respect to each other, they execute sequentially. This greatly reduces the conceptual complexity of the programming model, since the programmer must not worry about locking or arbitrary interleavings of parallel threads. Instead, the interleaving occurs at intuitive points that are specified and controlled by the programmer.

We call a collection of fibers with the same scope of execution a *braid*. Braids are object-like abstractions, and fibers are method-like abstractions.

After presenting the high-level abstractions and an example algorithm that uses them, we present the extensions to the hardware and instruction set architecture that are required to support braided code. The fundamental hardware abstractions are *inquiring operations*, which return information about potentially lengthy instructions instead of blocking and waiting for the result. They allow the program to respond adaptively and perform other work while waiting for the operation to complete (for instance, loading a memory

```

void histogram(int vals[], int hist[]) {
    for (int i = 0; i < vals.length; i++) {
        int d = vals[i] % hist.length;
        hist[d]++;
    }
}

```

**Figure 1: Sequential Histogram Calculation**

location into the cache, or computing an address translation).

To allow efficient interaction between software and hardware resources, the hardware associates *memory transaction identifiers* with in-flight operations. The software can then poll for completion, typically when some other memory operation is deferred.

We present a number of braided algorithms, and provide examples of the corresponding compiled code that makes use of our instruction set extensions.

The rest of this paper is organized as follows: Section 2 presents the high-level-language constructs for expressing braided code. Section 3 describes our instruction set architecture (ISA) extensions. Section 4 describes how the ISA extensions can be implemented inexpensively at the micro-architectural level. Section 5 presents a number of braided algorithms. Section A presents the compiled code for a simple braided program. Finally, we discuss related work and present our conclusions.

## 2. PROGRAMMING CONSTRUCTS

We begin by describing a set of high-level language constructs for expressing latency-adaptable programs. In subsequent sections we will describe the necessary instruction set architecture (ISA) and micro-architecture enhancements to support these constructs, and then show how the constructs can be efficiently compiled to the extended ISA.

Fundamentally, a program that is able to adapt to memory latency must defer some portion of its work while necessary data is being fetched. To do this it must be able to operate on a *partial order* over its data. Therefore, the programmer must give up explicit control of the ordering of some operations in exchange for higher performance.

### 2.1 Explicit Programming for Availability

The fundamental primitive notion that allows programmers to express latency-dependent partial orders is that the program can *inquire* whether an object is in local memory before committing to perform some work on that object. The exact definition of “local memory” is implementation-defined, but the difference between “local” and “non-local” should be large. For instance, “local” might be on-chip (for instance L1 or L2 cache) while “non-local” is off-chip (L3 cache or DRAM).

Statements that query for memory locality are called *inquiring operations*.

#### 2.1.1 The Readnow and Updatenow Operators

There are two inquiry operators, which take an expression and return true if it is possible to read or update the expression (typically something like a C *lvalue*) “now” — that is, with minimal delay.

The syntax of these unary boolean operators is

```

readnow expr
updatenow expr

```

They are typically used in conditional statements, for instance

```

if (readnow a[i])

```

```

void histogram(int vals[], int hist[]) {
    IntQ q = new IntQ(32);
    int i = 0;

    while (i < vals.length) {
        for (; i < vals.length && ! q.full(); i++) {
            int d = vals[i] % hist.length;
            if (updatenow hist[d])
                hist[d]++;
            else {
                want hist[d];
                q.put(d);
            }
        }

        while (! q.empty()) {
            int d = q.get();
            hist[d]++;
        }
    }
}

```

**Figure 2: Histogram Calculation with Explicit Work Deferral**

```

        process(a[i]);
    else
        enqueue(a[i]);

```

#### 2.1.2 The Want Statement

The **want** statement is a declaration that the program intends to use a memory location in the near future; it is essentially a prefetching declaration.

```

want expr

```

#### 2.1.3 Example: Histogram Collection

To see how the inquiry and prefetch operators can be used to program explicitly latency-adaptive computations, consider the simple example of histogram calculation, as shown in Figure 1. This function iterates over an array of values, and for each value increments an associated “bucket” in the histogram array. The iteration over the array of values is sequential, and memory latencies will be avoided by the hardware caching and prefetching mechanisms. However, the updates to the histogram array could be completely random.

If the histogram array is too large for the cache, the program will suffer a cache miss on a very high percentage of the histogram updates. If the penalty for a miss is large, the slowdown will be dramatic.

Figure 2 shows how many of the cache misses can be avoided using explicitly programmed inquiring operations. Before updating the histogram array, the function uses the **updatenow** operator to check whether the target location can be written without delay. If so, the update is performed as before. Otherwise, the target index is stored in a queue and the **want** statement is used to prefetch the target array element. When the queue is full, it is drained and then the iteration continues.

With the appropriate tuning of the queue size, most accesses in will hit in the cache. However, tuning the queue size is tricky: if it is too small, the function will try to update the queued histogram elements before they have been prefetched. If it is too large, the prefetched elements will already have been evicted again by the time they are processed.

## 2.2 Programming with Braids and Fibers

```

computeHistogram(int vals[], int buckets) {
    int hist[] = new int[buckets];
    braid Histogram(vals, hist);
    displayHistogram(hist);
}

braid class Histogram {
    final int vals[];
    final int hist[];

    public Histogram(int v[], int h[]) {
        vals = v; hist = h;

        for each (int i = 0 : vals.length-1) {
            int d = vals[i] % hist.length;
            UPDATE(hist[d]);
        }
    }

    void fiber UPDATE(update int bucket) {
        bucket++;
    }
}

```

**Figure 3: Braided Histogram Calculation**

While explicit use of inquiring operations, as illustrated above, is sometimes useful, the need for the user to tune the queue size also illustrates a potential weakness of the approach. Therefore, it is desirable for the system (consisting of the hardware, the compiler, and the run-time system) to manage the work queue of outstanding prefetches implicitly.

In order to do this, the program must be broken up into units that can execute out-of-order. We call these units *fibers*, since they behave somewhat like very fine-grained threads.

The fundamental abstraction over fibers is the *braid*. Intuitively, with our braid construct, one can think of the fibers as being attached at the beginning and end of the **braid**, but freely intertwined between the two ends.

The major difference between braiding and other fine-grained multi-threading abstractions is that at execution time there is a well-defined total order of high-level operations. Programmers therefore do not need to concern themselves with the complexities of concurrency and synchronization, but only with specifying a relaxed ordering in which the high-level functions of the program can occur.

Furthermore, since all of its component fibers must terminate for a braid to terminate, braids provide *abstraction over concurrency*: two successive braids can be viewed as completely sequential at the macro level. Any internal concurrency in either braid will never have any effect on the operation of the other braid.

### 2.2.1 Braid Classes

A braid is declared as a special type of class, and a fiber is declared as a special type of method.

A braid defines a set of braid instance variables. Inside of a braid, the only variables accessible are the braid instance variables, local variables, and formal parameters of methods. The braid class may define static variables, but they must be constants (for instance, **final** pointers to mutable objects are not allowed).

A braid's instance variables are not accessible outside of the braid. This includes other braids of the same class; therefore, the instance variables are not **private** in the Java sense of the term.

Braids are scoped and block-structured: they are created with the **braid** statement, which creates a braid object and executes the associated fibers with the braid; braids may *not* be created with the

**new** operator. The **braid** statement terminates when the main control fiber and all deferred fibers of that braid have terminated. Braid objects can be passed as parameters and stored in other objects, but once terminated any attempt to invoke a braid method will raise a **BraidTerminated** exception.

It is easiest to explain braids in the context of an example. Figure 3 shows a braided version of the histogram calculation. The histogram calculation is encapsulated in a **braid class** named **Histogram**. It has a normal constructor, which takes the value and histogram arrays as parameters, but in runs in the context of its own braid, so it may spawn fibers.

### 2.2.2 Fiber Methods

The **fiber** method **UPDATE()** takes two parameters: an integer **lvalue** and an integer index. Since **lvalue** is a **update** parameter, the fiber is only executed if the parameter satisfies the **updatenow** predicate. Since **index** is a **use** parameter, its value is made available inside the fiber. Note that since **lvalue** is not declared as **use update**, it is actually not in scope inside of the fiber. By convention, fiber names are all upper case to emphasize fiber calls.

Fibers are *coroutine scheduled*: a fiber either runs immediately if all of its **read** and **update** parameters are available, or it is deferred. If a fiber is deferred, other previously deferred fibers may run. Thus any fiber within the braid may run at any fiber call point, or at the end of the braid, but nowhere else. Furthermore, fibers execute atomically until they terminate or until they make another fiber call.

Semantically, each fiber runs with its own stack. Elimination of separate stack, store of return address, and inlining are all optimizations of the basic model.]

### 2.2.3 The Braid Statement

The **braid** statement in the **computeHistogram()** method (which is part of some other class) creates a braid scope, instantiates the **braid class** object of type **Histogram** within that scope, and executes its constructor. The **Histogram** object is not bound to any variable since there are not any meaningful operations that can be performed on it once it terminates. However, within the braid the object can be referred to normally using the **this** reference, which can be stored in stack or heap variables of arbitrary lifetimes.

### 2.2.4 Fiber Optimizations

Programmers should attempt to use as few **use** parameters to **fiber** methods as possible. Values that are constant during the execution of the braid should be stored in **final** braid instance variables, and when it does not involve extra computation on the fast (non-deferring) execution path, multiple objects should be consolidated. For instance instead of passing three variables in this braided hash table insertion:

```

int i = hash(key);
INSERT(table[i], key, value);
:
:
INSERT(update Nod head, Key key, Val val) {
    Nod n = new Nod(key, val);
    n.next = head;
    head = n;
}

```

it is more efficient to pass a single **Nod** object that contains both the key and value pointers, since this operation is performed anyway if the **INSERT()** method is executed immediately:

```

int i = hash(key);
Nod n = new Nod(key, val);
INSERT(table[i], n);

```

```

:
:
INSERT(update Nod head, Nod node) {
    n.nxt = head;
    head = n;
}

```

Using these types of optimizations, the histogram example can easily be optimized to run with only a single word of stored state per deferred fiber.

### 2.2.5 The Break Braid Statement

The **break braid** statement can be used to terminate the current **braid** block and abort any pending fibers. Note that since fibers execute sequentially, there is no danger of a **break** statement asynchronously interrupting a running fiber — the only fiber that it interrupts is the one which itself issues the **break braid** statement.

## 3. INSTRUCTION SET ARCHITECTURE

In this section we describe the extensions to the instruction set architecture to support our approach. To make the design concrete, we present the ISA as extensions to the 64-bit IBM POWER [10] architecture, but the same principles can be applied to any modern instruction set architecture. We call our extended ISA the POWER<sup>P</sup> architecture.

As with all instructions that address memory, the issue of which addressing modes to support is an issue with our memory instructions. We will abstract this issue by simply presenting instructions that take `addr` as a parameter. There could be as few as one addressing mode per instruction or as many as are supported by all the various load/store instructions in the machine ISA. Which addressing modes to support is primarily a tradeoff between orthogonality and instruction set encoding space.

### 3.1 Inquiring Memory Operations

An *inquiring memory operation* is one that finds out whether a given address can be accessed in a timely fashion. This is in contrast to the informing memory operations of Horowitz et al. [3], which inform the program how long a memory operation has taken once it completes.

Inquiring memory operations return some state information about the memory location in question, which can subsequently be used for conditional execution of loads and stores.

The POWER instruction set [6] already contains prefetch instructions, called Data Cache Block Touch (`dcbt`) and Touch for Store (`dcbtst`). These instructions generate speculative loads of the cache line containing the effective address of the instruction. They are speculative in the sense that no faults are generated if the target address is not in virtual memory or is an illegal address. We will model our instruction set extensions as further “data cache block” operations.

In keeping with POWER architecture conventions, we assign a condition register (CR7) to hold the outcome of inquiring memory operations. An alternative implementation would be to have a completely separate set of memory query condition registers. Separating the memory query results may be beneficial for branch prediction, which may be more accurate if memory-dependent branches are treated differently.

Since the condition register is four bits wide, we assign the following meanings to the condition register fields:

**Available (EQ).** Indicates that the data at the memory address requested is available for load or store. The meaning of “available” is implementation-defined. In general, data is available if it can be supplied to the CPU without a large latency. The

```

dcbq. addr
CR7[SO] ← 0
CR7[LT,GT] ← Latency[addr]
if Local[addr]
    CR7[EQ] ← 1
else
    CR7[EQ] ← 0

```

(a) Data Cache Block Query

```

dcbqst. addr
CR7[SO] ← 0
CR7[LT,GT] ← Latency[addr]
if Local[addr] ∧ CachedExclusive[addr]
    CR7[EQ] ← 1
else
    CR7[EQ] ← 0

```

(b) Data Cache Block Query for Store

Figure 4: Inquiring Memory Instructions

most common cause of unavailability is a cache miss, but any source of latency can be considered, for instance a TLB miss or other address translation delay.

**Time1, Time2 (LT,GT).** These two bits together provide a two-bit value which estimates the total latency of the memory operation. The interpretation of the four values is implementation-defined, but roughly corresponds to an L1 hit (00), L2 hit (01), L3 hit (10), and main memory/remote CPU access (11).

**Overflow (SO).** For split-phase memory operations (see Section 3.2), indicates that hardware resources to initiate the split-phase operation are unavailable. Set to 0 by all other inquiring memory operations.

#### 3.1.1 Data Cache Block Query for load/Store

The simplest inquiring instruction simply asks whether the data at the given address is available. There are two variants: one for reading and one for writing. The latter is primarily for multiprocessor systems, where the operation of the cache-coherence protocol may delay a write.

Query operations which return “available” may optionally prefetch the data into the L1 cache or pipeline the data to the CPU.

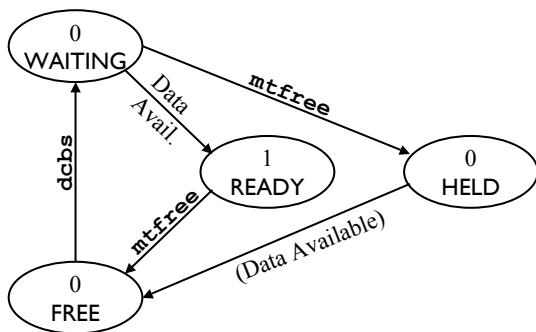
In the simplest incarnation of inquiring memory operations, only two instructions need to be added to the ISA (`dcbq` and `dcbqst` with one addressing mode each). Generalizations include multiple addressing modes and combined query/load instructions that perform the load operation if the data is available.

### 3.2 Split-Phase Memory Operations

Split-phase memory operations allow the software to initiate a load or store and be notified semi-asynchronously when the resources are available to perform it. The notification is semi-asynchronous because the hardware asynchronously sets a flag indicating that the data is available, but the software must poll for this condition.

To support split-phase memory operations, the hardware maintains a *memory transaction register* (MTR), which we assume is the same size as the architected register size, in our case 64 bits. Each bit in the MTR may be associated with a split-phase memory operation, and if so associated, indicates whether or not the memory is available (1) or not available (0).

The MTR may be read by the program but may not be written directly. Instead, split-phase memory operations on data that is



**Figure 5: State transitions for a memory transaction identifier and the corresponding user-visible bit in the Memory Transaction Register.**

not immediately available are associated with a particular memory transaction identifier (returned in a general purpose register). When the data becomes available, the bit indexed in the MTR by the returned GPR value is set.

The MTR can be examined by the program. Memory transaction identifiers must also be freed explicitly by the program, but may not be immediately usable depending on the hardware implementation (in particular, if a transaction identifier is freed while a memory transaction is in progress, the hardware may not make the freed MTR transaction identifier usable until the transaction completes).

Figure 5 shows the state transitions associated with split-phase operations. The normal sequence of states is Free-Waiting-Ready-Free. However, if a transaction identifier is freed before the memory operation completes the Held state may be entered for some period of time (typically until the outstanding memory operation completes).

### 3.2.1 Data Cache Block Prepare for load

The data cache block prepare for load and prepare for store instructions begin split-phase memory operations. If the data is already available, they act like their corresponding query operations. However, when the data is not available the hardware allocates a memory transaction identifier, which is an index into a bit in the MTR. This bit will eventually be set to 1 when the data becomes available.

The program can therefore save a small amount of state indexed by the transaction identifier number and periodically poll for completion. When the operation completes, it can branch to a handler and use the saved state to perform a deferred operation.

### 3.2.2 Data Cache Block Prepare for Store

The `dcbpst` instruction is almost identical, except that it prepares for a store. This causes the cache line containing `[addr]` to be obtained in exclusive mode instead of shared mode. On a

```

dcbp. Rn, addr
CR7[LT,GT] ← Latency[addr]
if Local[addr]
    CR7[EQ] ← 1
    CR7[SO] ← 0
else if MTR ≠ 641
    CR7[EQ] ← 0
    CR7[SO] ← 0
    Rn ← GetTransactionID(MTR)
    InitiateLoad[addr]
    (On completion, MTR[Rn] ← 1)
else
    CR7[EQ] ← 0
    CR7[SO] ← 1
  
```

**(a) Data Cache Block Prepare for load**

```

dcbpst. Rn, addr
CR7[LT,GT] ← WriteLatency[addr]
if Local[addr] ∧ CachedExclusive[addr]
    CR7[EQ] ← 1
    CR7[SO] ← 0
else if MTR ≠ 641
    CR7[EQ] ← 0
    CR7[SO] ← 0
    Rn ← GetTransactionID(MTR)
    InitiateLoadExclusive[addr]
    (On completion, MTR[Rn] ← 1)
else
    CR7[EQ] ← 0
    CR7[SO] ← 1
  
```

**(b) Data Cache Block Prepare for Store**

**Figure 6: Split-phase Prepare Instructions**

uniprocessor, `dcbpst` has the same functionality as `dcbp`.

### 3.2.3 Move From Memory Transaction Register

The POWER architecture defines a variety of special-purpose registers (SPRs), and instructions to read and write these registers. The memory transaction register (MTR) can only be read; it may not be written directly, although there are instructions which modify its state indirectly.

The Move From Memory Transaction register instruction loads a copy of the current state of the MTR into a general purpose register, as shown in Figure 7(a).

Changes to the MTR caused by split-phase operations and transaction identifier freeing operations must be reflected in program order.

Once the contents of the MTR are in a general purpose register, standard ALU instructions are used to determine the course of action. Common cases are comparing the MTR contents to 0 to determine whether any data has become available, and using the count leading zeros (CLZ) instruction to get the index of the first memory operation which has become available.

### 3.2.4 Memory Transaction Register Free/Clear

Once a split-phase memory operation has been completed, the associated transaction identifier should be freed for re-use by subsequent split-phase operations. There are two forms: the `mtrfree` instruction releases a single memory transaction identifier named by a GPR, as shown in Figure 7(b).

The `mtrclr` instruction shown in Figure 7(c) releases all memory transaction identifiers and may be used at context switches when software is not prepared to handle transaction identifiers gen-

- mfmtr Rn  
Rn ← [MTR]  
**(a) Move From MTR Instruction**
- mtfree Rn  
FreeTransactionID(MTR[Rn])  
**(b) MTR Free transaction identifier Instruction**
- mtrclr  
FreeTransactionID(MTR[0..63])  
**(c) MTR Clear transaction identifiers Instruction**
- cntzb Ra, Rs  
c ← 0  
for (n ← 0; n < 64; n++) c ← c + ¬(Rs)<sub>n</sub>  
Ra ← c  
**(d) Count Zero Bits Instruction**

**Figure 7: Memory Transaction Register (MTR) Instructions**

bma addr	Available
IFAR ← addr if CR7[EQ] = 1	
bmna addr	Not Available
IFAR ← addr if CR7[EQ] = 0	
bms addr	Slow
IFAR ← addr if CR7[LT] = 1	
bmfa addr	Fast
IFAR ← addr if CR7[LT] = 0	
bmta addr	Transaction Available
IFAR ← addr if CR7[SO] = 0	
bmnta addr	Trans. Not Available
IFAR ← addr if CR7[SO] = 1	

**Figure 8: Branch Mnemonics. Branch if Memory...**

erated by other contexts. In this case it is the software’s responsibility to either complete or re-initiate the split-phase operations when the context resumes execution.

There may be some delay between the freeing of a transaction identifier by software and its availability for re-use, especially if the transaction identifier is freed before the corresponding memory operation is complete.

### 3.2.5 Counting Free Transactions

In the event that multiple methods, threads, or processes are simultaneously issuing split-phase operations, it may be helpful for the application to determine how many split-phase operations can be issued before there are no more available transaction identifiers. If this is the case, the number of available transaction identifiers can be obtained by loading the MTR into a general purpose register and then counting the number of 0 bits in the register with the Count Zero Bits (cntzb) instruction shown in Figure 7(d).

## 3.3 Branch Mnemonics

To aid readability, a set of new mnemonics is added for branching based on the outcome of inquiring memory operations. These assembler mnemonics are shown in Figure 8.

## 4. MICROARCHITECTURE

We now describe the implementation of our instruction set extensions at the micro-architectural level. To make the design com-

plete, we describe it in terms of the micro-architecture of the IBM POWER4 chip [11], since it represents an aggressive design for which documentation is widely available. Obviously we are predicating our design on delay characteristics of future processor generations, but we expect the implementation to remain largely the same.

At the micro-architectural level, the inquiring instructions (dcbq and dcbqst), split-phase prepare instructions (dcbp and dcbpst), and memory transaction register instructions (mtrfree and mtrclr) primarily interact with the Load Miss Queue or LMQ.

In general, the hardware is free to take advantage of the fact that these operations are designed to improve performance, so a certain amount of imprecision may be tolerated (although this should not be abused, as it will otherwise negate the effect of the optimizations).

Thus the inquiry operations that set the EQ (memory available) bit and the LT, GT (memory delay) bit pair in the condition register are to some extent free to return arbitrary results. However, the hardware should err on the side of conservatism: it is better to report that a few in-cache data are not in cache, which will lead to spurious delay of work, than to report that out-of-cache data are in cache, which will lead to processor stalls.

Similarly, since bits in the MTR are set asynchronously by the hardware, the state of the MTR as observed by the program may be out-of-date with respect to completed transactions (but not with respect to freed transactions). This fact may be exploited by the implementation to avoid branch mis-predictions based on the MTR value by eagerly executing the mfmtr instruction. The only absolute restriction is that available data must eventually be detected by the program, but in practice it should be detected as soon as possible.

The number of memory transaction identifiers is implementation-dependent, anywhere from 0 to 64. It will generally be less than the number of LMQ entries.

## 4.1 Supplying Memory Latency Information

We can place the cache directory physically closer to the processor than the cache data array. This allows directory information to be accessed faster to reduce the overhead of latency inquiry operations. For example, the processor chip may contain the directory of the L3 cache, but not the data array of the L3 cache.

In addition, we do not have to distinguish cache latencies if the difference is small. Consider a computer system that employs on-chip L1 and L2 caches and off-chip L3 cache. Since the difference between on-chip and off-chip access latencies is much more than the difference between on-chip access latencies, we can treat L1 and L2 caches in the same way when a latency inquiry operation is performed. In particular, we may choose to report as “available” (EQ bit on) data that is in either L1 or L2 cache.

The cache state can be used to determine or predict the latency of memory access operations. For example, for a store operation, we can predict its latency based on whether and where the address is cached, and in what state the data is cached. If the cache state shows the address is cached with the exclusive ownership, a store operation can be executed on the cache with little cache coherence overhead. In contrast, if the cache state shows that the address is cached without the exclusive ownership, a store operation may not be executed before other cache copies are invalidated. The latency of a store operation is generally more important for the Sequential Consistency memory model than for relaxed memory models.

This implies that coherence information (such as MESI bits) which are normally stored at the coherence point (typically the on-chip L2 cache), should be propagated up to the L1 cache. However,

there is no synchronization requirement, since we are free to report stale information about the MESI bits.

If the desired address is in L1 cache it is immediately reported as available (EQ=1) with a latency of “very short” (LT,GT=00). If the L2 cache is on-chip, the inquiry operation is treated similarly to a load: it generates a specially tagged entry in the load miss queue (LMQ).

For inquiry operations (dcbq and dcbqst) the operation completes as soon as the L2 directory lookup has been performed. If the line is in the L2 cache, a prefetch to L1 cache is initiated, the line is reported as available, and the latency is reported as “short” (LT,GT=01). Otherwise, a prefetch is *not* initiated and the latency is reported as either “long” or “very long” (LT,GT=10 or 11). On an L2 hit, the LMQ entry is released when the prefetch completes; on a miss the LMQ entry is released immediately.

Note that when a load or store is issued subsequent to a corresponding cache query instruction, the load can be combined with the query in the LMQ. In fact, this can be done speculatively.

#### 4.1.1 Cache of Cache Latencies

Alternatively, the computer system can have a built-in prediction mechanism that can be used to predict whether an address is loaded in a cache or set of caches. The prediction mechanism is often based on a prediction table that is smaller than the cache directory and can therefore be accessed faster than the cache directory. For example, the prediction table could be a summary of the cache directory that contains only a subset of the address bits, or a hashed value from the address bits. When an address is loaded into the cache, a corresponding entry is created in the prediction table. The prediction table can be a set-associative table that uses an LRU replacement algorithm. For a latency inquiry operation, if the address is found in the prediction table, the latency is predicted to be the cache hit latency; otherwise the latency is predicted to be the cache miss latency.

## 4.2 Split-phase Operations and the MTR

The split-phase prepare instructions (dcbp and dcbpst) are implemented similarly to the inquiry instructions, but with more complex functionality. They always cause a speculative load of the cache line containing the effective address into the L1 cache. Therefore the corresponding LMQ entry is not released until the cache load is completed (although the instruction is retired as soon as the inquiry portion completes and MTR resources have been allocated).

If the cache line is not “available”, a split-phase operation is initiated and the hardware searches for an available transaction identifier. This is done by looking for a zero bit in a non-architected register that records reserved MTID’s (the Memory Transaction Reservation Register or MTRR)

If the MTRR is all ones (indicating no transactions available), the SO bit is set to 1, indicating an MTR overflow, and the instruction completes. The associated cache load operation can optionally be flushed. This is not required, but is perhaps best done asynchronously to avoid flooding the LMQ in pathological cases.

If there are transactions available, the search begins with the bit specified in the non-architected Memory Transaction Search Register (MTRSR). The MTRSR is rotated to the left (modulo the number of available transactions) until a free bit is found in the MTRR.

Assuming a there is a free bit in the MTRR, that bit position is assigned as the memory transaction identifier (MTID) of the request, and is return in register Rn. The MTID is associated with the LMQ entry, and that bit position is set to 1 in the MTRR and 0 in the MTR.

When a cache load completes, the cache logic looks for an MTID

associated with the completed LMQ entry. If present, that bit is set to 1 in the MTR. Then the LMQ entry is released. Note that the update of the MTR bit may be performed asynchronously without any particular timing constraint, and the bit need not be set before the LMQ entry is released.

## 4.3 Memory Transaction Register Instructions

The corresponding pair of bits in the MTR and the MTRR determine the four states in the state transition diagram for MTIDs (Figure 5). In fact, the value reported for the MTR by the mfmtr instruction is really the value of  $MTR \wedge MTRR$ .

The mtfree instruction releases an MTID. The MTID may, however, not be available for immediate re-use if the MTID is associated with an LMQ entry. The implementation of mtfree is simply to negate the associated bits in the MTR and MTRR, which has the effect of moving a READY transaction into the FREE state, while a WAITING transaction is moved into the HELD state.

The mtclr operation simply negates the entire MTR and MTRR registers.

## 5. ALGORITHMS

### 5.1 Garbage Collector Mark Phase

```
class MarkStack {
    private final ObjectStack stack;

    MarkStack(Object[] roots) {
        stack = ObjectStack.create(roots);
    }

    void mark() {
        while (! stack.empty()) {
            Object X = stack.pop();
            markObject(X);
        }
    }

    void markObject(X) {
        if (! X.mark) {
            X.mark = true;
            offsets = X.class.offsets();
            for (int i = 0; i < offsets.length; i++)
                {
                    pointer = Peek(ADDRESS(X)+offsets[i]);
                    if (pointer != null)
                        stack.push(pointer);
                }
        }
    }
}
```

Figure 9: Sequential Garbage Collector Mark Phase

The mark phase traverses the object graph, marking all objects that it encounters as live. It is therefore very memory intensive as well as containing many unpredictable, non-local memory accesses.

The standard formulation of the mark phase [7, 5] is shown in Figure 9. The markStack is initially the set of roots (pointers in registers and on the stack). The algorithm then recursively traverses the object graph, marking each new object it encounters and backtracking when it encounters a marked object.

```

void mark() {
    Queue deferredQ = new Queue(QUEUESIZE);

    while (! stack.empty()) {
        while (! stack.empty()) {
            X = stack.pop();

            if (READNOW(* X))
                markObject(X);
            else {
                WANT * X;
                deferredQ.add(X);
                if (deferredQ.full())
                    markDeferred(deferredQ);
            }
        }
        markDeferred(deferredQ);
    }
}

void markDeferred(Queue Q) {
    while (! Q.empty()) {
        Object X = Q.remove();
        markObject(X);
    }
}

```

**Figure 10: Garbage Collector Mark Phase With Explicit Work Deferral.**

The version using inquiring operations is shown in Figure 10. In this example, `markObject(X)` is only called if `X` is in local memory. Otherwise, `X` is prefetched placed in the `deferredQ`. When the queue fills up, its elements are processed with `markDeferred()`, which marks objects unconditionally to avoid repeated attempts to prefetch the same object, since that could cause cache pollution.

## 5.2 Sparse Matrix Algorithms

Figure 12 shows how braids and fibers can be used to implement a sparse matrix-vector multiply operation. Each product operation is executed in a fiber, so that when a memory operation would stall another row is consulted for concurrent work.

## 6. RELATED WORK

### 6.1 Instruction Set Extensions

Horowitz et al. [3] describe *informing* memory operations. While similar in spirit to our *inquiring* memory operations, informing memory operations provide feedback about memory performance to the program after the fact. They are therefore useful for profile-based approaches, but lack the ability to adapt dynamically in the manner of our inquiring operations, and they also lack the mechanism for associating operations with transaction identifiers.

Mowry and Ramkisson [9] describe how informing memory operations may be used for compiler-controlled multi-threading on a processor core with simultaneous multi-threading (SMT). However, the programming model is significantly more complex since the programmer must be prepared to deal with arbitrary interleavings. The advantage of braiding over multi-threading is that the points at which fibers are interleaved are limited, well-defined, and obvious to the programmer.

```

braid class GCMark {
    final ObjectStack stack;

    GCMark(Object[] roots) {
        stack = ObjectStack.create(roots);

        WHILE EACH (! markStack.empty()) {
            Object X = markStack.pop();
            MARK(* X);
        }
    }

    void fiber MARK(read Object o) {
        markObject(o);
    }
}

```

**Figure 11: Braided Garbage Collector Mark Phase**

Morris and Hunt [8] describe a computer system with instructions that allow registers to be probed to determine if an attempt to use them will stall. This approach is significantly less flexible than ours, since the ability to schedule arbitrary code at arbitrary times is greatly restricted by having to use fixed registers. The ability of our system to allocate and free transaction identifiers gives the software many more degrees of freedom.

There are also some superficial similarities with the Intel IA-64 NAT bits [4].

## 6.2 Other Language Constructs

Split-C [2] provides split-phase memory operations (`get` and `put`) in a parallel, SPMD programming language. However, because the split-phase operations are assumed to have high overhead and there is no automatic notification of completion, the programmer must explicitly synchronize at various points, waiting for all outstanding split-phase operations to complete. Our hardware support makes the individual operations much lighter weight, which in turn allows synchronization at the level of individual split-phase operations.

I-structures [1] provide a split-phase functional abstraction in the form of an array of write-once elements, and an array element read operation which blocks until the write has occurred. This is fundamentally different from our approach in that the semantics are based on the availability of the value, rather than the memory location.

## 7. CONCLUSIONS

We have presented braids and fibers, high level programming constructs which allow a program to be expressed as sections of sequential code that can be interleaved in a *partial order*. Although fibers are partially ordered with respect to each other, they execute sequentially. This greatly reduces the conceptual complexity of the programming model, since the programmer must not worry about locking or arbitrary interleavings of parallel threads. Instead, the interleaving occurs at intuitive points that are specified and controlled by the programmer.

We have also demonstrated how braids and fibers can be supported by simple extensions to existing instruction set architecture and microarchitecture. The fundamental hardware abstractions are *inquiring operations*, which return information about potentially lengthy instructions instead of blocking and waiting for the result. They allow the program to respond adaptively and perform other

work while waiting for the operation to complete (for instance, loading a memory location into the cache, or computing an address translation).

To allow efficient interaction between software and hardware resources, the hardware associates *memory transaction identifiers* with in-flight operations. The software can then poll for completion, typically when some other memory operation is deferred.

We have presented a number of braided algorithms, and provided examples of the corresponding compiled code that makes use of our instruction set extensions.

## APPENDIX

### A. IMPLEMENTATION TECHNIQUES

In this appendix, we show how high level constructs can be compiled into machine language using the split-phase memory instructions we have proposed adding to the architecture.

Figure 13 how code is generated for a single memory operation that could miss in the cache. Figure 14 shows how this can be generalized to a sequence of dependent loads.

### B. REFERENCES

- [1] ARVIND, NIKHIL, R. S., AND PINGALI, K. K. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989), 598–632.
- [2] CULLER, D. E., ARPACI-DUSSEAU, A. C., GOLDSTEIN, S. C., KRISHNAMURTHY, A., LUMETTA, S., VON EICKEN, T., AND YELICK, K. A. Parallel programming in Split-C. In *Supercomputing* (1993), pp. 262–273.
- [3] HOROWITZ, M., MARTONOSI, M., MOWRY, T. C., AND SMITH, M. D. Informing memory operations: memory performance feedback mechanisms and their applications. *ACM Trans. Comput. Syst.* 16, 2 (1998), 170–205.
- [4] INTEL CORPORATION. *IA-64 Application Developer's Architecture Guide*, 1999.
- [5] JONES, R., AND LINS, R. *Garbage Collection*. John Wiley and Sons, 1996.
- [6] MAY, C., SILHA, E., SIMPSON, R., AND WARREN, H., Eds. *The PowerPC Architecture*, second ed. Morgan Kaufmann, San Francisco, California, 1994.
- [7] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM* 3, 4 (1960), 184–195.
- [8] MORRIS, D. C., AND HUNT, D. B. Computer system having an instruction for probing memory latency, Oct. 2001. United States Patent 6,308,261. See also European Patent Application 0-933-698.
- [9] MOWRY, T. C., AND RAMKISSOON, S. R. Software-controlled multithreading using informing memory operations. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture* (Toulouse, France, Jan. 2000), pp. 121–132.
- [10] SILHA, E., WOTTRENG, A., AND MAY, C. *PowerPC AS Instruction Set Architecture*, 1999.
- [11] TENDLER, J. M., ET AL. POWER4 system microarchitecture. *IBM J. Res. Dev.* 46, 1 (Jan. 2002), 5–25.
- [12] WULF, W. A., AND MCKEE, S. A. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News* 23, 1 (Mar. 1995), 20–24.

```

BRAID class SparseMatMuller {
  final SparseMatrix m;
  final double[] v;
  final double[] d;

  public SparseMatMuller(SparseMatrix mx,
                        double[] vx, double[] dx)
  {
    m = mx;  v = vx;  d = dx;

    FOR EACH (int i = 0 : m.rows-1)
      for (int j = m.rowpos[i];
           j < m.rowpos[i+1]; j++)
        // Live at call: i, j, j loop bound
        d[i] += PRODUCT(m.data[j].value,
                       v[m.data[j].index]);
  }

  double FIBER PRODUCT(READ double & melem,
                      READ double & velem)
  {
    return melem * velem;
  }
}

class SparseMatrix {
  int rows, cols;

  int[] rowpos;
  SparseElem[] data;
}

value SparseElem {
  double value;
  int index;

  SparseElem(double v, int i) {
    value = v; index = i;
  }

  SparseElem() { SparseElem(0.0, 0); }
}

```

**Figure 12: Braided Sparse Matrix-Vector Multiply**

```

BRAID
  for (int i=0; i < A.length; i++) {
    e = A[i];
    WHEN HERE (e.key) USE (e)
      f(e)
  }

```

(a) High-level Code that performs work  $f(e)$  for every element  $e$  of array  $A$ .

```

BEGIN:                                     ; R6 (on entry): array pointer A
  lwz R7, length(R6)                       ; get array length
  cmpwi R7, 0                               ; empty array?
  b END                                     ; skip the whole mess
  sldi R7, R7, 3                            ; convert length to word index
  li R31, 0                                 ; R31: deferred load mask
  li R8, 0                                   ; R8: index variable i
loop:   lwzx R0, R6, R8                      ; e = A[i]
  bl when                                   ; perform body of when statement
  addi R8, R8, 4                            ; i++
  cmpw R8, R7                               ; at end of array?
  blt loop                                  ; no. do next piece of work

finish: bl dodefer                          ; try to do deferred work
  cmpwi R31, 0                              ; any outstanding loads?
  bne finish                                ; yup. keep draining
END:

```

(b) POWER Assembly code for main loop and FINISH statement.

```

when:   DCBP. R1, key(R0)                   ; prepare for load
        BMNA miss                          ; if memory not available, handle
hit:    lwz R2, key(R0)                     ; load key (won't block)
        [ f() ]                             ; got data: process it
        blr                                 ; return

miss:   BMTNA notags                       ; handle tag unavailable
        bitset R31, R31, R1                 ; add new deferred load to mask
        sldi R1, R1, 3                      ; convert tag to word offset
        stw R0, deftbl(R1)                 ; place e in deferred table

dodefer: MFMTR R3                          ; get tag vector
        and R3, R3, R31                    ; mask out other deferred loads
        cntlzw R1, R3                      ; get index of first available value
        cmpwi CR3, R1, 64                  ; none found?
        beqlr CR3                          ; then done; return
        mtrfree R1                         ; got one. free memory tag
        bitclr R31, R31, R1                ; remove from mask of deferred loads
        sldi R1, R1, 3                      ; convert tag index to word offset
        lwz R0, deftbl(R1)                 ; get saved e from table
        b hit                              ; go back and do the work

notags: mflr R30                            ; save the return address
        mr R29, R0                         ; save e

; save R29/R30 in RAM???

work:   bl dodefer                          ; now try to do deferred work
        bne CR3, work                      ; more? then keep doing it
        mtlr R30                           ; restore return address
        mr R0, R29                          ; restore e
        b when                              ; retry original load

```

(c) POWER Assembly code for WHEN statement body using split-phase load, miss handler, and processing of deferred work.

Figure 13: The WHEN Statement and Its Translation.

```

BRAID
  for (int i=0; i < A.length; i++) {
    e = A[i];
    WHEN HERE (e.key.string[0]) USE (e)
      f(e.key.string)
  }

```

(a) High-level Code that performs work  $f(e)$  for every element  $e$  of array  $A$ .

```

when:   DCBP. R1, key(R0)           ; prepare for load of key field
        BMNA miss1                ; if memory not available, handle
hit1:   lwz R0, key(R0)            ; load key
        DCBP. R1, string(R0)      ; prepare for load of string field
        BMNA miss2                ; if memory not available, handle
hit2:   lwz R0, string(R0)        ; load string
        DCBP. R1, 0(R0)           ; prepare for load of first string char
        BMNA miss3                ; if memory not available, handle
hit3:   lha R2, 0(R0)             ; load first char of string
        [ f() ]                   ; got data: process it
        blr                       ; return

miss1:  li R2, #hit1              ; setup to return to hit1 when load completes
        b miss                    ; goto shared miss handler
miss2:  li R2, #hit2              ; setup to return to hit2 when load completes
        b miss                    ; goto shared miss handler
miss3:  li R2, #hit3              ; setup to return to hit3 when load completes
miss:   BMTNA notags              ; handle tag unavailable
        bitset R31, R31, R1       ; add new deferred load to mask
        sldi R1, R1, 3            ; convert tag to word pair offset
        stw R0, deftbl(R1)        ; place e in deferred table
        stw R2, deftbl+8(R1)      ; place restart address in deferred table

dodefer:MFMTTR R3                 ; get tag vector
        and R3, R3, R31           ; mask out other deferred loads
        cntlzw R1, R3             ; get index of first available value
        cmpwi CR3, R1, 64         ; none found?
        beqlr CR3                ; then done; return
        mtrfree R1                ; got one. free memory tag
        bitclr R31, R31, R1       ; remove from mask of deferred loads
        sldi R1, R1, 4            ; convert tag index to word pair offset
        lwz R0, deftbl(R1)        ; get saved e from table
        lwz R2, deftbl+8(R1)      ; get saved restart address from table
        mtctr R2                  ; prepare to jump to restart address
        bctr                       ; go back and do the work

notags: mflr R30                  ; save the return address
        mr R29, R0                ; save e
        addi R28, R2, -16         ; save restart address (adjust to DCBP instruction)
work:   bl dodefer                 ; now try to do deferred work
        bne CR3, work             ; more? then keep doing it
        mtlr R30                  ; restore return address
        mr R0, R29                ; restore e
        mtctr R28                 ; prepare to jump to restart address
        bctr                       ; retry original load

```

(b) POWER Assembly code for WHEN statement body illustrating handling of cascading split-phase loads.

Figure 14: A Complex WHEN Statement and Its Translation.