Braids and fibers: Language constructs with architectural support for adaptive responses to memory latencies

D. F. Bacon
X. Shen

As processor speeds continue to increase at a much higher rate than memory speeds, memory latencies may soon approach a thousand processor cycles. As a result, the flat memory model that was made practical by deeply pipelined superscalar processors with multilevel caches will no longer be tenable. The most common approach to this problem is multithreading; however, multithreading requires either abundant independent applications or well-parallelized monolithic applications, and neither is easy to come by. We present high-level programming constructs called ***braids*** *and* ***fibers****. The programming constructs facilitate the creation of programs that are partially ordered, in which the partial orders can be used to support adaptive responses to memory access latencies. Braiding is simpler than parallelizing, while yielding many of the same benefits. We show how the programming constructs can be effectively supported with simple instruction set architecture extensions and microarchitectural enhancements. We have developed braided versions of a number of important algorithms. The braided code is easy to understand at the source level and can be translated into highly efficient instructions using our architecture extensions.*

Introduction

High-performance microprocessors 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 access latencies. However, this abstraction is beginning to break down because the current trend of memory hierarchy is becoming ever deeper while the relative speed of communications between the processor and memory continues to increase. Latencies for DRAM accesses may soon approach a thousand processor cycles. The term *memory wall* was coined to refer to the increasing gap between CPU and memory speeds [1].

Although optimization techniques, such as prefetching, have been used successfully to hide significant numbers of memory latencies, they usually work well only for highly predictable programs. Ultimately, as the performance of the memory system becomes more and more nonuniform, such techniques for tolerating memory latencies will no longer function.

If there is a line at a restaurant, patrons are given an estimate of the length of time they will have to wait. Similarly, if callers to a company's customer service line are put on hold, they are updated on how long they will remain there. In these real-world scenarios, it seems obvious that information about the length of a delay should be provided. We argue that what is obvious in the restaurant business should be obvious in the computer business.

In this paper, we present a high-level programming model that enables programs to respond to long memory latencies by performing other work while high-latency memory operations are in progress. The fundamental approach is to divide the program into *fibers*—sections of sequential code that can be interleaved in a partial order. Although fibers are partially ordered with respect to one

©**Copyright** 2006 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8646/06/$5.00 © 2006 IBM

another, they execute sequentially. This greatly reduces the semantic complexity of the programming model because the programmer does not have to worry about locking or arbitrary interleaving of parallel threads. Instead, the interleaving occurs at intuitive points that are specified and controlled by the programmer.

A *braid* is a collection of fibers with the same scope of execution. We present high-level abstractions based on braids and fibers and show an exemplary histogram computation algorithm that uses them. Braids are object-like abstractions, and fibers are method-like abstractions. We show instruction set architecture (ISA) and microarchitecture extensions that are required to support braided code. The fundamental architecture abstractions are memory inquiry operations that provide latency prediction information about potentially lengthy memory operations instead of blocking and waiting for results of the memory operations. This allows the program to respond adaptively and perform other work while waiting for the memory operations to complete (for instance, loading data from the memory into a cache or computing an address translation). To allow efficient interaction between software and hardware resources, the hardware associates memory transaction identifiers with in-flight memory operations. The software can then poll for completion, typically when some other memory operation is deferred.

Programming constructs

A program that is able to adapt to memory latencies generally has to defer some portion of its work while requested data is being fetched. This implies that the program should be able to operate on a partial order. As a result, the programmer must give up explicit control of the ordering of some operations in exchange for higher performance.

In this section, we begin by describing a set of high-level language constructs for expressing latency-adaptive programs. In subsequent sections, we describe the necessary ISA and microarchitecture enhancements to support these constructs and show how the language constructs can be efficiently compiled to the extended ISA.

Programming with explicit memory inquiry operations

The fundamental primitive notion that allows programmers to express latency-adaptive 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 can be implementation-dependent, but the difference between local memory latency and nonlocal memory latency is generally large. For instance, local can mean on-chip SRAM cache, while nonlocal can mean off-chip DRAM memory.

There are two memory inquiry operators, `readnow` and `updatenow`. The `readnow` operator takes an expression as a memory address and returns *true* if data with the memory address can be read now with low memory latency. Similarly, `updatenow` takes an expression as a memory address and returns *true* if data with the memory address can be updated now with low memory latency.

In the event that the accessed data cannot be read or updated with low memory latency (e.g., the data is not local), the program can choose to defer the memory operation to a later time. This is often accompanied by a prefetch operation so that when the memory access operation is retried, the data can be accessed with low memory latency. The operator `readprefetch` prefetches data for read, and the operator `updateprefetch` prefetches data for write.

We use a simple histogram calculation example to demonstrate how the memory inquiry operators can be used to explicitly program latency-adaptive computations. As shown in **Figure 1(a)**, the histogram calculation function iterates over an array of values, and for each value it increments an associated bucket in the histogram array. The iteration over the array of values is sequential, and the updates to the histogram array can 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 can be dramatic.

Figure 1(b) shows how cache misses can be avoided using explicitly programmed memory inquiry operations. Before updating the histogram array, the function uses `updatenow` to check whether the target location can be updated with small memory access latency. If so, the update is performed as usual; otherwise, the target index is stored in a deferral queue and `updateprefetch` is used to retrieve the target array element. When the queue is full, it must be drained before the iteration continues.

With appropriate tuning of the queue size, most accesses 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 may already have been evicted from the cache by the time they are processed.

Programming with braids and fibers

While explicit use of memory inquiry operations can be a practical solution, the necessity for the user to tune the deferral queue size indicates a potential weakness of the approach. It is generally desirable for the memory system to automatically manage the deferral queue of outstanding prefetches. This requires that the program be

broken up into computation units that can be executed out of order. Such computation units are referred to as *fibers*, since they behave somewhat like very fine-grained threads. A *braid* is a collection of fibers with the same scope of execution. Intuitively, one can think of the fibers as being attached at the beginning and at the end of the braid, but freely intertwined between the two ends.

A major difference between braiding and other fine-grained multithreading program abstractions is that braiding ensures, at execution time, 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, because all component fibers of a braid must terminate in order for the braid to terminate, the construct of braids provides an abstraction over concurrency: Two successive braids can be viewed as completely sequential at the macro level, and any internal concurrency in either braid will never have an effect on the other braid.

A braid is declared as a special type of class and defines a set of braid instance variables. The only variables accessible inside a braid 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, pointers to mutable objects are not allowed). Instance variables of a braid are not accessible outside the braid, including other braids of the same class. Therefore, the instance variables are even more restricted than private fields in the sense of the term used in the Java** language.

The `braid` statement creates a braid object and executes the associated fibers within the braid. Braids may not be created with the `new` operator that is normally used for allocating objects. 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 the braid statement is terminated, any attempt to invoke a braid method will raise a `BraidTerminated` exception.

A fiber is declared as a special type of method. Fibers of a braid are co-routine-scheduled: A fiber can run immediately if all of the data with memory addresses specified by its `readnow` and `updatenow` parameters can be accessed with low memory latencies, or the fiber is deferred. If a fiber is deferred, another previously deferred fiber can run. Any fiber within the braid may run at any fiber call point, or at the end of the braid, but nowhere else. Furthermore, a fiber executes atomically (that is, it is not interrupted by other fibers) until it terminates or until it makes another fiber call. Semantically, each fiber runs with its own stack. Elimination of a separate stack, the store of the return address, and inlining are all

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

(a)

```
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 {
        updateprefetch hist[d];
        q.put(d);
      }
    }

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

(b)

```
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;
      BUCKETADD(hist[d]);
    }
  }

  void fiber BUCKETADD(updatenow int bucket) {
    bucket++;
  }
}
```

(c)

Figure 1

Histogram calculations: (a) sequential; (b) with explicit memory inquiries; (c) with braids and fibers.

optimizations of the baseline model. These optimizations are shown in the exemplary assembly code for braid constructs section below.

Braids may be nested; that is, a fiber in a braid may itself create a braid whose internal fibers are interleaved. However, when a fiber of the nested braid is deferred, only other fibers of that braid are eligible to run, not the fibers of the enclosing braid.

A braid object can be referred to within the braid using the reference `this`. A `break braid` statement can be used, if needed, to terminate the current braid block and abort any pending fibers. Note that because 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 that issues the break braid statement.

Figure 1(c) shows the histogram calculation that is encapsulated in a braid class. The braid class `Histogram` has a constructor, which takes a value array and a histogram array as parameters. The `braid` statement in the `computeHistogram` creates a braid scope, instantiates the braid object within that scope, and executes its constructor. The `Histogram` object is not bound to any variable, since there is no meaningful operation that can be performed on it once it terminates. The fiber method `BUCKETADD` takes an integer bucket value to update as an `updatenow` parameter, signifying that the method will be invoked when that memory location can be updated with low memory latency. It is worth noting that the parameter is passed by reference, in contrast to the conventional Java by-value parameter passing.

Generally, to optimize fiber methods, as few parameters as possible should be employed. Values that are constant throughout the execution of the braid should be stored in `final` braid instance variables. Multiple objects should be consolidated if they involve no extra computation on the nondeferring execution path. Using such techniques, a program can often be optimized to run with a few words of stored state for each deferred fiber. For instance, consider a trivial hash table insertion program as follows:

```
int i = hash (key);
INSERT (table [i], key, value);
...
void fiber INSERT (updatenow Nod head, Key key, Val val)
{
    Nod n = new Nod (key, val);
    n.nxt = head;
    head = n;
}
```

Rather than passing three parameters to the `INSERT` method, it is generally more efficient to pass a single `Nod` object that contains both the `key` and `value` pointers, because this operation is performed anyway if the method is executed immediately:

```
int i = hash (key);
Nod n = new Nod (key, val);
INSERT (table [i], n);
...
void fiber INSERT (updatenow Nod head, Nod node) {
    n.nxt = head;
    head = n;
}
```

Instruction set extensions

In this section, we describe the ISA extensions required to support braids and fibers. Without losing generality, we present the ISA extensions in the context of the IBM Power Architecture* [2], although the same technique can be applied to other modern microprocessor architectures. **Figure 2** shows the ISA extensions and their semantics. The ISA extensions include memory inquiry operations and split-phase memory prepare operations.

As with normal memory access instructions, the issue of which addressing modes to support is also an issue with our new memory instructions. There could be as few as a single addressing mode or as many as are supported by the normal memory load and store instructions. To avoid unnecessary details, we simply use a memory address `addr` to represent a memory address generated by some memory addressing mode.

Memory inquiry operations

A memory inquiry operation can be used to determine whether a given memory address can be accessed in a timely fashion, before the memory address is actually accessed. This is in contrast to the *informing memory operation*, which informs the program of the length of time a memory access operation has taken once it has completed [3]. A memory inquiry operation is able to provide appropriate latency information that can subsequently be used for conditional execution of memory access operations.

The Power Architecture comprises prefetch instructions for memory load and store operations, referred to respectively as Data Cache Block Touch (`dcbt`) and Data Cache Block Touch for Store (`dcbtst`). A prefetch instruction speculatively accesses a cache line that contains data for the corresponding effective address. The prefetch operation is speculative in the sense that no exception is generated if, for example, the effective address is illegal or accessing the effective address would cause a page fault. We present our ISA extensions as additional "data cache block" operations.

In keeping with Power Architecture conventions, a condition register (CR7) is assigned to hold the outcome

of memory inquiry operations. An alternative would be to use a dedicated set of memory inquiry registers. Because the condition register is 4 bits wide, we assign the following meanings to the condition register fields:

- Available (`EQ`). This bit indicates that requested data of the memory address is available for memory load or store, while the exact meaning of "available" can be implementation-dependent. In general, the data is considered available if it can be accessed by the processor with low memory access latency. Although the most common cause of unavailability is a cache miss, any source of memory latency can be considered—for instance, an address translation delay due to a translation lookaside buffer (TLB) miss.
- Time1, Time2 (`LT`, `GT`). These bits provide a 2-bit encoding as an estimate for the memory access latency. The interpretation of the four possible values can be implementation-dependent. For example, the latency estimates can encode an L1 hit (00), L2 hit (01), L3 hit (10), and main memory access (11).
- Overflow (`SO`). This bit indicates that no split-phase memory prepare operations can be initiated because of resource constraint.

The memory inquiry instruction determines whether requested data of a memory address is available. There are two variants: one for reading and one for writing. The latter is primarily for multiprocessor systems, in which necessary cache coherence operations may cause delay to a write operation, although the data is found in a local cache. In the simplest form of memory inquiry operations, the ISA is extended with two new instructions, Data Cache Block Query (`dcbq`) and Data Cache Block Query for Store (`dcbqst`). The semantics of these instructions are given in Figure 2(a).

More sophisticated forms of memory inquiry instructions can be supported. For example, a composite memory inquiry load instruction can—in addition to the memory inquiry operation—perform a memory load operation if the data is available or, if the data is unavailable, a memory prefetch operation.

Split-phase memory prepare operations

Split-phase memory prepare operations allow software to initiate a memory access operation in case of a cache miss and be notified semi-asynchronously when the requested data becomes accessible with low memory access latency. The notification is semi-asynchronous in the sense that hardware asynchronously sets a flag when the data is available, and software must poll the flag to be notified.

To support split-phase memory prepare operations, a 64-bit Memory Transaction Register (MTR) is defined

```
dcbq addr
  CR7[SO] <- 0
  CR7[LT,GT] <- Latency(addr)
  if Local(addr)
    CR7[EQ] <- 1
  else
    CR7[EQ] <- 0

dcbqst addr
  CR7[SO] <- 0
  CR7[LT,GT] <- Latency(addr)
  if Local(addr) and CachedExclusive(addr)
    CR7[EQ] <- 1
  else
    CR7[EQ] <- 0
```

(a)

```
dcbp Rn, addr
  CR7[LT,GT] <- Latency(addr)
  if Local(addr)
    CR7[EQ] <- 1
    CR7[SO] <- 0
  else if TransactionIDAvailable(MTR)
    CR7[EQ] <- 0
    CR7[SO] <- 0
    Rn <- GetTransactionID(MTR)
    InitiateLoad(addr)
    (On completion, MTR[Rn] <- 1)
  else
    CR7[EQ] <- 0
    CR7[SO] <- 1

dcbpst Rn, addr
  CR7[LT,GT] <- WriteLatency(addr)
  if Local(addr) and CachedExclusive(addr)
    CR7[EQ] <- 1
    CR7[SO] <- 0
  else if TransactionIDAvailable(MTR)
    CR7[EQ] <- 0
    CR7[SO] <- 0
    Rn <- GetTransactionID(MTR)
    InitiateLoadExclusive(addr)
    (On completion, MTR[Rn] <- 1)
  else
    CR7[EQ] <- 0
    CR7[SO] <- 1
```

(b)

```
mfmtr Rn
  Rn <- [MTR]

mtrfree Rn
  FreeTransactionID(MTR[Rn])

mtrclr
  FreeTransactionID(MTR[0..63])
```

(c)

Figure 2

Instruction set architecture extensions: (a) Data Cache Block Query; (b) Data Cache Block Prepare; (c) MTR Move, Free, and Clear.

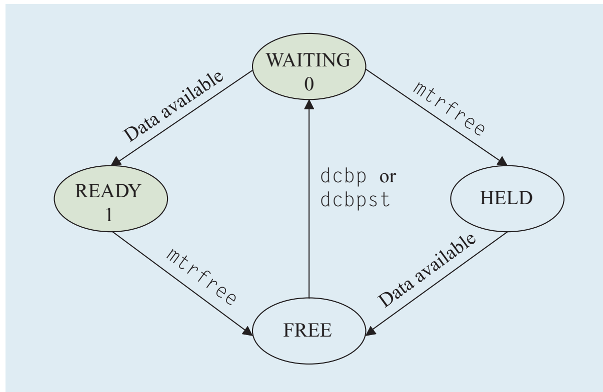

Figure 3

State transitions for memory transaction identifiers.

at the architecture level. Each bit in the MTR may be associated with a split-phase memory prepare operation; if so associated, it indicates whether data of the memory operation is available (1) or not available (0).

The MTR can be examined by the program but cannot be directly modified by it. When executing a split-phase memory prepare operation, if the requested data cannot be accessed with low memory access latency, the system associates the memory prepare operation with a memory transaction identifier. The memory transaction identifier can be returned to the program via a General Purpose Register (GPR). When the data becomes available, the system sets the corresponding MTR bit that is indexed by the memory transaction identifier. The program must free the memory transaction identifier once the data is accessed so that the memory transaction identifier can be used for other split-phase memory prepare operations. Note that if a memory transaction identifier is freed while the corresponding memory transaction is still in progress, the system may not make the memory transaction identifier usable until the memory transaction completes.

Figure 3 shows state transitions for memory transaction identifiers. Each memory transaction identifier corresponds to an MTR bit. The typical state transaction sequence for a memory transaction identifier is FREE–WAITING–READY, where the term FREE indicates that the memory transaction identifier is not associated with any memory operation, WAITING indicates that the memory transaction identifier is associated with an outstanding memory operation, and READY indicates that the memory transaction identifier is associated with a memory operation that has been completed. Note that the memory transaction identifier may enter the HELD state in the event that the memory transaction identifier is freed before the associated memory operation completes.

Data cache block prepare instructions

The Data Cache Block Prepare (`dcbp`) instruction initiates a split-phase memory operation. If the requested data is already available, it behaves as a memory inquiry operation. However, if the data is not available, the system allocates a memory transaction identifier. The memory transaction identifier is an MTR index that specifies an MTR bit that will eventually be set to 1 when the requested data becomes available. The program can therefore maintain a small amount of state comprising the memory transaction identifier and periodically poll for the completion of the outstanding memory transaction. When the memory transaction completes, the program can branch to an appropriate handler that uses the saved state to perform a deferred operation.

The Data Cache Block Prepare for Store (`dcbpst`) instruction is almost identical, except that it prepares the data for a store operation. This causes the cache line to be obtained in an exclusive cache coherence state. The use of the `dcbpst` instruction should generally be limited for read–modify–write operation sequences on a memory address. The write-buffer mechanism present in modern computer architectures is usually effective to hide the memory access latency of a store operation without a preceding read of the same memory address. The semantics of the `dcbp` and `dcbpst` instructions are shown in Figure 2(b).

MTR management instructions

Some other MTR-related instructions are shown in Figure 2(c). The Move from MTR (`mfmtr`) instruction loads the current state of the MTR into a GPR.

Once a split-phase memory prepare operation has been completed, the associated memory transaction identifier should be freed for reuse by future split-phase memory prepare operations. The MTR Free (`mtrfree`) instruction releases a memory transaction identifier referred to by a GPR. The MTR Clear (`mtrclr`) instruction releases all memory transaction identifiers and may be used at context switch times when software is not prepared to handle memory transaction identifiers generated by other contexts.

When multiple methods, threads, or processes are simultaneously issuing split-phase memory prepare operations, it may be helpful for the program to determine the number of split-phase memory prepare operations that can be issued before there are no more available memory transaction identifiers. If this is the case, the number of available memory transaction identifiers can be obtained by loading the MTR into a GPR and then counting the number of zeros in the GPR.

For programming convenience at the assembly level, adequate opcode mnemonics can be provided for branch operations based on the content of the Condition

Register (CR7). The Condition Register can be set by memory inquiry and prepare instructions. Exemplary branch mnemonics are shown in Figure 2(d).

Microarchitecture support

We now discuss the microarchitecture support of the ISA extensions for memory inquiry operations and split-phase memory prepare operations. Without losing generality, we show simple implementations in the context of the IBM POWER4* microprocessor because it represents an aggressive microarchitecture implementation with publicly available documentation [4].

At the microarchitectural level, the memory inquiry instructions (`dcbq` and `dcbqst`) and split-phase memory prepare instructions (`dcbp` and `dcbpst`) interact primarily with the Load Miss Queue (LMQ). In general, the system can take advantage of the fact that these instructions are intended 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 memory inquiry operations that set the memory available bit and the memory latency bits in the Condition Register (CR7) are, to some extent, free to return prediction results without causing correctness issues. It is usually less harmful to report that some local data is nonlocal, which leads to a spurious delay of work, than to report that some nonlocal data is local, which leads to a processor stall.

Memory inquiry operation support

To support memory inquiry operations, the directory of a cache can be placed closer to the accessing CPU than the data array of the cache. This allows directory information to be accessed more rapidly to reduce the overhead of memory inquiry operations. For example, the processor chip may contain the directory of an L3 cache, but not the data array of the L3 cache.

In addition, it is generally unnecessary to distinguish cache access latencies if the difference between the latencies is small. Consider a computer system that employs on-chip L1 and L2 caches and an off-chip L3 cache. Because the difference between on-chip and off-chip access latencies is far greater than the difference between on-chip access latencies, we can treat L1 and L2 caches in the same way when a memory inquiry operation is performed. In particular, the system may report data as "available" (`EQ`) if the data is in either the L1 or the L2 cache.

The cache state can be used to determine or predict latencies of memory access operations. For example, the latency of a memory store operation can be predicted according to whether and where the address is cached and the state in which the data is cached. If the cache state shows that the address is cached with the exclusive ownership, a memory store operation can be executed on the cache with low cache coherence overhead. In contrast, if the cache state shows that the address is cached without the exclusive ownership, a memory store operation may not be executed before other cache copies, if they exist, are invalidated. The latency of a memory store operation is generally more important for strict memory models, such as sequential consistency, than for relaxed memory models, such as the PowerPC memory model.

For a memory inquiry operation, if requested data is in the L1 cache, it is immediately reported as available (`EQ` = 1) with a latency of "very short" (`LT`, `GT` = 00). A memory inquiry operation completes as soon as the L2 directory lookup has been performed. If the data is found in the L2 cache, a prefetch to the L1 cache is initiated, the requested data 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).

Alternatively, the computer system can comprise a built-in prediction mechanism that can be used to predict whether data of a memory address can be found 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 more rapidly 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 a least-recently-used (LRU) replacement algorithm. For a memory inquiry operation, if the address is found in the prediction table, the memory access latency is predicted to be the cache hit latency; otherwise, the memory access latency is predicted to be the cache miss latency.

Split-phase memory prepare operation support

The split-phase memory prepare instructions (`dcbp` and `dcbpst`) are implemented similarly to the memory inquiry instructions, but with additional functionality. A split-phase memory prepare operation always causes the cache line containing the corresponding memory address to be fetched into the L1 cache.

If the cache line is not available, a split-phase memory operation is initiated, and the system searches for an available memory transaction identifier. This is done by searching a 64-bit Memory Transaction Reservation Register (MTRR) that records reserved memory transaction identifiers (MTIDs). Each bit in the MTRR indicates whether the corresponding memory transaction identifier is reserved (1) or not reserved (0). If the MTRR indicates that no memory transaction identifier is

available, the SO bit in the Condition Register (CR7) is set to 1 to indicate an MTR overflow, and the memory prepare instruction completes.

If the MTRR indicates that there is a memory transaction identifier available, the corresponding memory transaction identifier is returned in a general register specified by the memory prepare instruction and is associated with the outstanding memory operation. As a result, the corresponding MTR bit is set to 0, indicating that the outstanding memory operation is not yet completed. Meanwhile, the corresponding MTRR bit is set to 1, indicating that the memory transaction identifier is reserved for use.

When an outstanding memory operation completes, if the outstanding memory operation is associated with a memory transaction identifier, it means that the outstanding memory operation is part of a split-phase memory prepare operation. Consequently, the corresponding MTR bit is set to 1 to indicate that the requested data can be accessed with small memory access latency. Note that the update of the MTR bit may be performed asynchronously without particular timing constraint.

Exemplary assembly code for braid constructs

We have presented the braids and fibers high-level programming constructs and the ISA and microarchitecture extensions needed to support them. We now show how they work together by describing the assembly code that implements the braided histogram computation shown in Figure 1(c).

The assembly code shown in **Figure 4** demonstrates results that are achievable via a high-quality optimizing compiler in conjunction with the strong isolation properties of braids, which are designed to increase opportunities for such optimizations. In particular, the isolation properties minimize the amount of state information that corresponds to an iteration of the `for each` loop. It is important to minimize the state information that must be saved when a `for each` iteration is deferred.

Figure 4(a) shows the translation of the `for each` loop of the histogram computation code in Figure 1(c). The translation is essentially the same as that for an equivalent `for` loop, except that we have quasi-inlined the code of `BUCKETADD` by passing all parameters in registers and performing global register allocation across the functions. As a result, registers are not reused (except for the scratch register R2). A smaller set of registers could be used, but we use unique registers to make the code easier to understand.

The loop loads `vals[i]` and hashes it to produce the resulting histogram index `d`, which is then converted into an array offset in R0. It then makes a call to the `fiber` code for `BUCKETADD`. The fiber method has been quasi-inlined; however, as we will see, it is necessary to use a call (branch-and-link) in order to accommodate the complex control flow that can result if a desired memory location is not available.

Beginning with the `fiber` code in Figure 4(b), a split-phase memory operation is initiated on the memory location `hist[d]`. If the memory location is available, the code falls through to the instruction labeled `hit`, and the operation of the `BUCKETADD` method is executed as it would be in the absence of fibers; i.e., the array element is loaded, incremented, and stored, and control is returned to the call in Figure 4(a). Otherwise, we branch to `miss` and verify that a valid transaction identifier for the split-phase operation was indeed obtained, and then store the state of the loop iteration (which, in this case, is the single register containing the array offset) in a table.

An attempt is then made to substitute a deferred value from a previous iteration. This substitution idiom is the key to obtaining high performance in the translation of braided code, since it results in a simple control flow and minimizes special-case code. The tag vector is checked for a completed operation. If none is available, we return to the calling loop in Figure 4(a), which can then continue with the work of subsequent iterations that might also be deferred.

If a deferred iteration has completed, the associated transaction identifier is released so that it may be reused, the state of the deferred iteration is loaded into R0, and the program then branches back to `hit` to perform the method body on the deferred operation. Control is then returned to the loop body in order to perform the next iteration—*next* meaning after the iteration that was just deferred, not *next* after the iteration that was just completed. This is the key to the iteration substitution idiom.

In the event that there are no more resources available for split-phase memory operations, control passes to `notags`, where it saves the return address and the state of the loop iteration and then tries to drain the split-phase operations by calling (via branch-and-link) the `dodefer` code. The use of a call here is necessary so that when the deferred work is completed, control returns to the `notags` handler, which can retry the iteration that failed owing to a lack of transaction resources.

Exemplary braided algorithms

In this section, we present two exemplary braided algorithms: the mark phase for garbage collection and a sparse matrix-vector multiply algorithm.

As shown in **Figure 5**, the mark phase for garbage collection traverses the object graph to mark all objects that it encounters as live. It is generally memory-intensive and comprises unpredictable memory accesses.

```
BEGIN:  ; on entry R6: array pointer vals[], R9: array pointer hist[]
        lwz R10, length(R9)     ; R10: array length of hist[]
        lwz R7, length(R6)      ; get array length of vals[]
        cmpwi R7, 0             ; empty array?
        b END                   ; skip the whole mess
        slwi R7, R7, 2          ; convert length to word index
        li R31, 0               ; R31: deferred load mask
        li R8,  0               ; R8: index variable i

loop:   lwzx R2, R6, R8         ; R2: vals[i]
        divwu R0, R2, R10       ; compute d by taking
        mullw R0, R0, R10       ;    vals[i]
        subf R0, R0, R2         ;       modulo hist.length
        slwi R0, R0, 2          ; convert index to offset
        bl fiber                ; perform body of fiber method
        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:
```

(a)

```
fiber:  DCBPST. R1, R9(R0)      ; prepare for load of hist[d]
        BMNA miss               ; if memory not available, handle
hit:    lwz R2, R9(R0)          ; load histogram entry (won't block)
        addi R2, R2, 1          ; add 1 to histogram entry
        stw R2, R9(R0)          ; store updated histogram entry
        blr                     ; return

miss:   BMTNA notags            ; handle tag unavailable
        bitset R31, R31, R1     ; add new deferred load to mask
        slwi R1, R1, 2          ; convert tag to word offset
        stw R0, deftbl(R1)      ; place offset of d 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, 32       ; none found?
        beqlr CR3               ; then done; return
        mtrfree R1              ; got one. free memory tag
        bitclr R31, R31, R1     ; remove from mask of deferred loads
        slwi R1, R1, 2          ; convert tag index to word offset
        lwz R0, deftbl(R1)      ; get saved offset of d from table
        b hit                   ; go back and do the work

notags: mflr R2                 ; get return address
        stw R2, ntret           ; save the return address
        stw R0, ntoff           ; save element offset

work:   bl dodefer              ; now try to do deferred work
        bne CR3, work           ; more? then keep doing it
        lwz R2, ntret           ; load saved return address
        mtlr R2                 ; restore return address
        lwz R0, ntoff           ; restore element offset
        b fiber                 ; retry original load
```

(b)

Figure 4

Optimized code for braid implementation: (a) assembly code implementing the `for each` loop in Figure 1(c); (b) assembly code implementing the fiber method `BUCKETADD` in Figure 1(c).

```
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);
      }
    }
  }
}
```

(a)

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

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

      if (Updatenow(* X))
        markObject(X);
      else {
        UpdatePrefetch * X;
        deferredQ.add(X);
        if (deferredQ.full())
          markDeferred(deferredQ);
      }
    }
    markDeferred(deferredQ);
  }
}

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

(b)

```
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(updatenow Object o) {
    markObject(o);
  }
}
```

(c)

Figure 5

Mark phase of garbage collection: (a) sequential; (b) with explicit memory inquiries; (c) with braids and fibers.

Figure 5(a) shows a standard formulation of the mark phase for garbage collection [5, 6]. The algorithm recursively traverses the object graph, marking each new object it encounters and backtracking when it encounters a marked object.

Figure 5(b) shows the mark phase using explicit memory inquiry operations, in which `markObject` is invoked only if the object can be marked with low memory access latency. If the object is not in a cache with low memory access latency, the object marking is deferred while the object is prefetched to a deferral queue called `deferredQ`. When the deferral queue fills up, its elements are processed with `markDeferred`, which marks objects without further deferring any object marking. Figure 5(c) shows the braided version for the mark phase of the garbage collector.

Figure 6 shows how braids and fibers can be used to implement a sparse matrix-vector multiply calculation. Each product operation is executed in a fiber, so that if a memory operation would stall, another row would be consulted for concurrent work.

Related work

Horowitz et al. describe informing memory operations [3]. While similar in spirit to the memory inquiry operations, informing memory operations usually provide feedback about memory access operations to the program after the memory access operations are performed. The technique is therefore useful for profile-based approaches, but it lacks the ability to adapt dynamically in the manner of the memory inquiry operations. It also lacks a mechanism for associating outstanding memory operations with memory transaction identifiers.

Mowry and Ramkissoon describe how informing memory operations may be used for compiler-controlled multithreading on a processor core with simultaneous multithreading [7]. However, the programming model is complicated because the programmer must be prepared to deal with arbitrary interleaving. The major advantage of braiding over multithreading is that the points at which fibers are interleaved are limited, well-defined, and obvious to the programmer.

Morris and Hunt describe a computer system with instructions that allow registers to be probed to determine whether an attempt to use them would stall [8]. This approach is significantly less flexible than techniques based on the split-phase memory prepare operations, as the ability to schedule arbitrary code at arbitrary times is greatly restricted by the required use of fixed registers. The ability of our system to allocate and free memory transaction identifiers gives software more degrees of freedom.

```
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++)
        d[i] += PRODUCT(m.data[j].value, v[m.data[j].index]);
  }

  double fiber PRODUCT(readnow double & melem, readnow 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 6

Braided version of sparse matrix-vector multiply calculation.

Split-C provides split-phase memory operations in a parallel, single-program multiple-data programming language [9]. However, because the split-phase memory 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 memory operations to complete. Our architecture support makes split-phase memory operations very lightweight (having low overhead), and this in turn allows synchronization at the level of individual split-phase memory operations.

The I-structures introduced by Arvind et al. [10] provide a split-phase functional abstraction in the form of an array of write-once elements. With I-structures, a memory read operation blocks until the corresponding memory write operation has occurred on the array element. This is fundamentally different from our approach in that the execution of the memory read operation is based on the availability of the data rather than the latency of the memory access operation.

The transactional coherence and consistency model provides a shared-memory model in which atomic transactions are always the basic units of parallel programming and memory consistency [11]. Supporting atomic transactions architecturally often requires expensive hardware and software enhancements, such as large on-chip buffers for atomic transactions and software-managed memory regions for on-chip buffer overflows.

Conclusions

We have presented braids and fibers—high-level programming constructs that facilitate the creation of programs that are partially ordered, in which the partial orders can be used to support adaptive responses to memory access latencies. Although fibers within a braid are partially ordered with respect to one another, they are executed sequentially. This greatly reduces the semantic complexity of the programming model because the programmer need not worry about locking or arbitrary interleaving of parallel computations. Instead, the interleaving occurs at intuitive points that are specified and controlled by the programmer.

We have demonstrated how braids and fibers can be effectively supported at the ISA level and at the microarchitecture level. The fundamental architecture abstractions are memory inquiry operations, which provide latency prediction information about potentially lengthy memory operations instead of waiting for the memory operations to complete. The memory inquiry operations allow the program to respond adaptively and perform other work while waiting for outstanding memory operations to complete. To further allow effective interaction between software and hardware, the system can associate memory transaction identifiers with in-flight memory operations so that software can poll for their completion if needed. We have presented exemplary assembly code that makes use of our ISA extensions and several exemplary braided algorithms.

Acknowledgments

This material is partially based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCH3039004. We thank Marc Auslander for helpful discussions.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

References

1. W. A. Wulf and S. A. Mckee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Arch. News* **23**, No. 1, 20–24 (March 1995).
2. C. May, E. Silha, R. Simpson, and H. Warren, Eds., *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, Morgan Kaufmann, San Francisco, 1994.
3. M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith, "Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Applications," *ACM Trans. Computer Syst.* **16**, No. 2, 170–205 (May 1998).
4. J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy, "POWER4 System Microarchitecture," *IBM J. Res. & Dev.* **46**, No. 1, 5–25 (January 2002).
5. J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine," *Commun. ACM* **3**, No. 4, 184–195 (April 1960).
6. R. Jones and R. Lins, *Garbage Collection*, John Wiley and Sons, Ltd., Chichester, England, 1996.
7. T. C. Mowry and S. R. Ramkissoon, "Software-Controlled Multithreading Using Informing Memory Operations," *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, 2000, pp. 121–132.
8. D. C. Morris and D. B. Hunt, "Computer System Having an Instruction for Probing Memory Latency," U.S. Patent No. 6,308,261, October 2001.
9. D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick, "Parallel Programming in Split-C," *Proceedings of the International Conference on Supercomputing*, November 1993, pp. 262–273.
10. Arvind, R. S. Nikhil, and K. K. Pingali, "I-Structures: Data Structures for Parallel Computing," *ACM Trans. Programming Lang. & Syst.* **11**, No. 4, 598–632 (October 1989).
11. L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004, p. 102.

Received June 29, 2005; accepted for publication August 8, 2005; Internet publication March 7, 2006

David F. Bacon *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (bacon@us.ibm.com).* Dr. Bacon is a Research Staff Member at the Thomas J. Watson Research Center. He leads the Metronome project, which produced the first hard real-time garbage collection system. His algorithms are included in most compilers and runtime systems for modern object-oriented languages, and his work on thin locks was selected as one of the most influential contributions in the twenty years of the Programming Language Design and Implementation Conference. Dr. Bacon received his A.B. degree from Columbia University and his Ph.D. degree in computer science from the University of California at Berkeley. His recent work focuses on high-level real-time programming, embedded systems, programming language design, and computer architecture. He holds six patents. Dr. Bacon is a member of the IEEE and the ACM, for which he is on the governing boards of the ACM Special Interest Group for Programming Languages (SIGPLAN) and the ACM Special Interest Group on Embedded Systems (SIGBED).

Xiaowei Shen *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (xwshen@us.ibm.com).* Dr. Shen is a Research Staff Member at the Thomas J. Watson Research Center, where he manages the Scalable Server Network and Memory Systems Department. He received his B.S. and M.S. degrees in computer science from the University of Science and Technology of China, and his M.S. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology. His research interests include computer architectures, compilers, networks, software–hardware co-design, and many aspects of parallel and distributed computing. His recent work focuses on commercially viable high-productivity computing systems, symmetric multiprocessing systems, and clusters of low-end servers. Dr. Shen has two issued patents and 15 pending patents in computer architecture and systems.