Memory-Mapping Support for Reducer Hyperobjects

I-Ting Angelina Lee [*]
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139 USA
angelee@csail.mit.edu

Aamir Shafi [†]
National University of Sciences and Technology
Sector H-12
Islamabad, Pakistan
aamir.shafi@seecs.edu.pk

Charles E. Leiserson
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139 USA
cel@mit.edu

ABSTRACT

Reducer hyperobjects (reducers) provide a linguistic abstraction for dynamic multithreading that allows different branches of a parallel program to maintain coordinated local views of the same nonlocal variable. In this paper, we investigate how ***thread-local memory mapping (TLMM)*** can be used to improve the performance of reducers. Existing concurrency platforms that support reducer hyperobjects, such as Intel Cilk Plus and Cilk++, take a hypermap approach in which a hash table is used to map reducer objects to their local views. The overhead of the hash table is costly — roughly 12× overhead compared to a normal L1-cache memory access on an AMD Opteron 8354. We replaced the Intel Cilk Plus runtime system with our own Cilk-M runtime system which uses TLMM to implement a reducer mechanism that supports a reducer lookup using only two memory accesses and a predictable branch, which is roughly a 3× overhead compared to an ordinary L1-cache memory access. An empirical evaluation shows that the Cilk-M memory-mapping approach is close to 4× faster than the Cilk Plus hypermap approach. Furthermore, the memory-mapping approach admits better locality than the hypermap approach during parallel execution, which allows an application using reducers to scale better.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent programming*; D.3.3 [**Software**]: Language Constructs and Features—*Concurrent programming structures*.

General Terms

Design, Experimentation, Performance.

This work was supported in part by the National Science Foundation under Grant CNS-1017058. Aamir Shafi was funded in part by a Fulbright grant during his visit at MIT.

[*]I-Ting Angelina Lee is currently affiliated with Intel Labs, Hillsboro, OR.

[†]Aamir Shafi is currently an Assistant Professor at NUST SEECS and was a Visiting Fulbright Scholar at MIT during the course of this research.

Keywords

Cilk, dynamic multithreading, memory mapping, reducers, reducer hyperobjects, task parallelism, thread-local memory mapping (TLMM), work stealing.

1. INTRODUCTION

Reducer hyperobjects (or ***reducers*** for short) [12] have been shown to be a useful linguistic mechanism to avoid determinacy races [11] (also referred as ***general races*** [28]) in dynamic multithreaded programs. Reducers allow different logical branches of a parallel computation to maintain coordinated local views of the same nonlocal variable. Whenever a reducer is updated — typically using an associative operator — the ***worker*** thread on which the update occurs maps the reducer access to its local view and performs the update on that local view. As the computation proceeds, the various views are judiciously ***reduced*** (combined) by the runtime system using an associative ***reduce*** operator to produce a final value.

Although existing reducer mechanisms are generally faster than other solutions for updating nonlocal variables, such as locking and atomic-update, they are still relatively slow. Concurrency platforms that support reducers, specifically Intel's Cilk Plus [19] and Cilk++ [25], implement the reducer mechanism using a ***hypermap approach*** in which each worker employs a thread-local hash table to map reducers to their local views. Since every access to a reducer requires a hash-table lookup, operations on reducers are relatively costly — about 12× overhead compared to an ordinary L1-cache memory access on an AMD Opteron 8354. In this paper, we investigate a ***memory-mapping approach*** for supporting reducers, which leverages the virtual-address translation provided by the underlying hardware to yield a close to 4× faster access time.

A memory-mapping reducer mechanism must address four key questions:

1. What operating-system support is required to allow the virtual-memory hardware to map reducers to their local views?
2. How can a variety of reducers with different types, sizes, and life spans be handled?
3. How should a worker's local views be organized in a compact fashion to allow both constant-time lookups and efficient sequencing during reductions?
4. Can a worker efficiently gain access to another worker's local views without extra memory mapping?

Our memory-mapping approach answers each of these questions using simple and efficient strategies.

1. The operating-system support employs ***thread-local memory mapping (TLMM)*** [23]. TLMM enables the virtual-memory

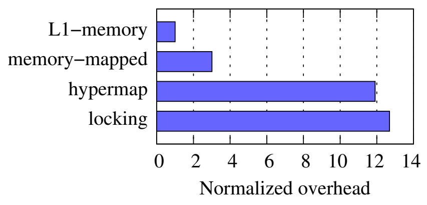


Figure 1: The relative overhead for ordinary L1-cache memory accesses, memory-mapped reducers, hypermap reducers, and locking. Each value is calculated by the normalizing the average execution time of the microbenchmark for the given category by the average execution time of the microbenchmark that performs L1-cache memory accesses.

hardware to map the same virtual address to different views in the different worker threads, allowing reducer lookups to occur without the overhead of hashing.

2. The thread-local region of the virtual-memory address space only holds pointers to local views and not the local views themselves. This ***thread-local indirection*** strategy allows a variety of reducers with different types, sizes, and life spans to be handled.
3. A ***sparse accumulator (SPA)*** data structure [14] is used to organize the worker-local views. The SPA data structure has a compact representation that allows both constant-time random access to elements and sequencing through elements stored in the data structure efficiently.
4. By combining the thread-local indirection and the use of the SPA data structure, a worker can efficiently transfer a view to another worker. This support for efficient ***view transferal*** allows workers to perform reductions without extra memory mapping.

We implemented our memory-mapping strategy by modifying Cilk-M [23], a Cilk runtime system that employs TLMM to manage the "cactus stack," to make it a plug-in replacement for Intel's Cilk Plus runtime system. That is, we modified the Cilk-M runtime system to replace the native Cilk runtime system shipped with Intel's C++ compiler by making Cilk-M conform to the Intel Cilk Plus Application Binary Interface (ABI) [17]. We then implemented the memory-mapping strategy in the Cilk-M runtime system and compared it on code compiled by the Intel compiler to Cilk Plus's hypermap strategy.

Figure 1 graphs the overheads of ordinary accesses, memory-mapped reducer lookups, and hypermap reducer lookups on a simple microbenchmark that performs additions on four memory locations in a tight `for` loop, executed on a single processor. The memory locations are declared to be `volatile` to preclude the compiler from optimizing the memory accesses into register accesses. Thus, the microbenchmark measures the overhead of L1-cache memory accesses. For the memory-mapped and hypermap reducers, one reducer per memory location is used. The figure also includes for comparison the overhead of locking — one `pthread_spin_lock` per memory location is employed, where the microbenchmark performs lock and unlock around the memory updates on the corresponding locks. The microbenchmark was run on a AMD Opteron processor 8354 with 4 quad-core 2 GHz CPU's with a total of 8 GBytes of memory and installed with Linux 2.6.32. As the figure shows, a memory-mapped reducer lookup is only about 3× slower than an ordinary L1-cache memory access and almost 4× faster than the hypermap approach (and as we shall see in Section 8, the differences between the two increases with the number of reducers). The overhead of locking is similar but slightly worse than the overhead of a hypermap reducer lookup.

A memory-mapped reducer admits a lookup operation that can be performed using only two memory accesses and a predictable

```
bool has_property(Node *n);
std::list<Node *> l;
// ...
void walk(Node *n) {
   if(n) {
       if(has_property(n))
          l.push_back(n);
       cilk_spawn walk(n->left);
       walk(n->right);
       cilk_sync;
   }
}
```

(a)

```
bool has_property(Node *n);
list_append_reducer<Node *> l;
// ...
void walk(Node *n) {
   if(n) {
       if(has_property(n))
          l->push_back(n);
       cilk_spawn walk(n->left);
       walk(n->right);
       cilk_sync;
   }
}
```

(b)

Figure 2: **(a)** An incorrect parallel code to walk a binary tree and create a list of all nodes that satisfy a given property. The code contains a race on the list `l`. **(b)** A correct parallelization of the code shown in Figure 2(a) using a reducer hyperobject.

branch, which is more efficient than a hypermap reducer. An unexpected byproduct of the memory-mapping approach is that it provides greater locality than the hypermap approach, which leads to more scalable performance.

The rest of the paper is organized as follows. Section 2 provides the necessary background on reducer semantics, which includes the linguistic model for dynamic multithreading and the reducer interface and guarantees. Section 3 reviews the hypermap approach to support the reducer mechanism. Sections 4, 5, 6, and 7 describe our design and implementation of the memory-mapped reducers, where each section addresses one of the four questions we mentioned above in details. Section 8 presents the empirical evaluation of the memory-mapped reducers by comparing it to the hypermap reducers. Section 9 summarizes related work. Lastly, Section 10 offers some concluding remarks.

2. CILK LINGUISTICS

This section reviews Cilk's linguistic model for dynamic multithreading in general and reducer hyperobjects in particular using the Cilk Plus [19] formulation. The results reported in this article should apply to other dynamic-multithreading[1] concurrency platforms — including MIT Cilk [13], Cilk++ [25], Cilk Plus , Fortress [2], Habanero [4, 9], Hood [7], Java Fork/Join Framework [21], OpenMP 3.0 [29], Task Parallel Library (TPL) [24], Threading Building Blocks (TBB) [31], and X10 [10] — but to our knowledge, Cilk++ and Cilk Plus are the only platforms that currently support reducers.

The dynamic multithreading support in Cilk Plus [19] augments serial C/C++ code with two principal keywords: `cilk_spawn` and `cilk_sync`.[2] The term dynamic multithreading alludes to the fact

[1]Sometimes called ***task parallelism***.

[2]Cilk Plus also includes a `cilk_for` keyword, which provides the parallel counterpart of a **for** loop, allowing all iterations of the loop to operate in parallel. We omit it here, since the Cilk Plus compiler simply "desugars" the `cilk_for` into code containing `cilk_spawn` and `cilk_sync` that

that these keywords expose the *logical* parallelism of the computation without mentioning the number of processors on which the computation should run. The underlying runtime system efficiently schedules the computation across available worker threads, which serve as processor surrogates, in a manner that respects the logical parallelism specified by the programmer.

We shall illustrate how these keywords work using an example. The code in Figure 2(a) walks a binary tree in preorder fashion to create a list of all nodes that satisfy a given property. The code checks and appends the current node onto an output list if the node satisfies the given property and subsequently walks the node's left and right children. The code is parallelized to walk the left and right children in parallel, which is achieved by preceding the call to walk the left child in line 8 with the `cilk_spawn` keyword. When a function invocation is preceded by the keyword `cilk_spawn`, the function is ***spawned***, and the scheduler may continue to execute the continuation of the caller in parallel with the spawned subroutine without waiting for it to return. Thus, in this example, the walk of the left child may execute in parallel with the continuation of the parent, which invokes the call to walk the right child (line 9).

The complement of `cilk_spawn` is the keyword `cilk_sync`, which acts as a local barrier and joins together the parallelism forked by `cilk_spawn`. The underlying runtime system ensures that statements after a `cilk_sync` are not executed until all functions spawned before the `cilk_sync` statement have completed and returned. Thus in this example, a `cilk_sync` statement is inserted at line 10 to ensure that the function does not return until the walk of the left and right children are done. This parallelization is incorrect, however, since it contains a ***determinacy race*** [11] (also referred a as ***general race*** [28]) on the list `l`, because the logically parallel subcomputations — the walks of the left and right subtrees — may potentially access the list in parallel.

Reducer hyperobjects (or ***reducers*** for short) [12] provide a linguistic mechanism to avoid such determinacy races in dynamically multithreaded computations. Figure 2(b) shows a correct parallelization of the `walk` function using a reducer. The code simply declares `l` in line 2 to be a reducer with a ***reduce operation*** that performs list append. By declaring list `l` to be a reducer, parallel accesses to `l` are coordinated by the runtime system, and the resulting output list produced by the code is identical to the result produced by a serial execution.

In order for a reducer to produce a deterministic output, the reduce operation must be associative. More precisely, a reducer is defined in terms of an algebraic ***monoid***: a triple $(T, \otimes, e)$, where T is a set and $\otimes$ is an binary associative operation over T with the identity e. Example monoids include $(\mathbf{Z}, +, 0)$ (addition on integers), $(\{\text{TRUE}, \text{FALSE}\}, \wedge, \text{TRUE})$ (logical AND on Booleans), and list append with the empty list as the identity, as in the example. The Cilk Plus runtime coordinates concurrent accesses to a reducer, guaranteeing that the output is the always the same as in a serial execution, as long as its reduce operation is associative.

3. SUPPORT FOR HYPERMAP REDUCERS

This section overviews the implementation of the Cilk++ [25] and Cilk Plus [19] reducer mechanism, which is based on hypermaps. Support for reducers was first proposed in [12] and implemented in Cilk++. The implementation in Cilk Plus closely follows that in Cilk++. This section summarizes the runtime support relevant for comparing the hypermap approach to the memory-mapping approach. We refer interested readers to the original article [12] for full details on the hypermap approach.

Support for reducers in Cilk Plus is implemented as a C++ template library. The user invokes functions in the runtime system, and the runtime system calls back to user-defined functions according to an agreed-upon API [18]. The type of a reducer is dictated by the monoid it implements and the type of data set that the monoid operates on. The reducer library implements the monoid interface and provides two important operations that the runtime invokes: IDENTITY, which creates an identity view for a given reducer, and REDUCE, which implements the binary associative operator that reduces two views. A user can override these operations to define her own reducer types.

During parallel execution, accesses to a reducer hyperobject cause the runtime to generate and maintain multiple views for the given reducer, thereby allowing each worker to operate on its own local view. A reducer is distinctly different from the notion of ***thread-local storage*** (or ***TLS***) [33], however. Unlike TLS, a worker may create and operate on multiple local views for a given reducer throughout execution. New identity views for a given reducer may be created whenever there is parallelism, because the runtime must ensure that updates performed on a single view retain serial semantics. In that sense, a local view is associated with a particular execution context but not with a particular worker. Consequently, a hypermap that contains local views is not permanently affixed to a particular worker, but rather to the execution context.

To see how local views are created and maintained, we first review how a work-stealing scheduler operates. In Cilk Plus (as well as in Cilk-M), the runtime creates a ***frame*** for every instance of a ***Cilk*** function that can spawn, which provides storage for bookkeeping data needed by the runtime. Whenever a worker encounters a `cilk_spawn`, it invokes the child and suspends the parent, pushing the parent frame onto the bottom of its ***deque*** (double-ended queue), so as to allow the parent frame to be stolen. When the child returns, it pops the bottom of the deque and resumes the parent frame. Pushing and popping frames from the bottom of the deque is the common case, and it mirrors precisely the behavior of a serial execution.

The worker's behavior departs from the serial execution if it runs out of work. This situation can arise, for example, when the code executed by the worker encounters a `cilk_sync` and children of the current frame have not returned. In this case the worker becomes a ***thief***, and it attempts to steal the topmost (oldest) frame from a randomly chosen ***victim*** worker. If the steal is ***successful***, the worker resumes the stolen frame. Another situation where a worker runs out of work occurs if it returns from a spawned child to discover that its deque is empty (the parent has been stolen). In this case, it checks whether the parent is stalled at a `cilk_sync` and if this child is the last child to return. If so, it performs a ***joining steal*** and resumes the parent function, passing the `cilk_sync` at which the parent was stalled. Otherwise, the worker engages in random work-stealing as in the case when a `cilk_sync` was encountered.

A worker's behavior precisely mimics the serial execution between successful steals. Logical parallelism morphs into true parallelism when a thief steals and resumes a function (the continuation of the parent after a spawn). Whenever a Cilk function is stolen, its frame is ***promoted*** into a ***full frame***, which contains additional bookkeeping data to handle the true parallelism created, including hypermaps that contain local views. Specifically, each full frame may contain up to 3 hypermaps — the ***user hypermap***, ***left-child hypermap***, and ***right-sibling hypermap*** — each of which respec-

recursively subdivides the iteration space to execute in parallel. Cilk Plus also includes support for vector operations, which are not relevant to the discussion here.

tively contains local views generated from computations associated with the given frame, its leftmost child, and its right sibling.

During parallel execution, a worker performs reducer-related operations on the user hypermap stored in the full frame sitting on top of its deque (since everything below the full frame mirrors the serial execution). The hypermap maps reducers to their corresponding local views on which the worker operates. Specifically, the address of a reducer is used as a key to hash the local view. Whenever a full frame is stolen, its original user hypermap is left with its child executing on the victim, and an empty user hypermap is created on the thief. When a worker encounters a reducer declaration, it creates a reducer hyperobject if one does not yet exist, and it inserts a key-value pair into its hypermap. The key is the address of the reducer, and the value is the initial identity view, referred to as the ***leftmost view***. When a reducer goes out of scope, at which point its leftmost view should reflect all updates, the worker removes the key-value pair from its hypermap. Finally, whenever the worker encounters an access to a reducer in the user code, the worker performs a lookup in its hypermap and returns the corresponding local view. If nothing is found in the hypermap (the user hypermap starts out empty when the frame is first promoted), the worker creates and inserts an identity view into the hypermap and returns the identity.

The other two hypermaps are placeholders. They store the accumulated values of the frame's terminated right siblings and terminated children, respectively. Whenever a frame is promoted, an additional set of local views may be created to accumulate updates from the computation associated with the full frame. These views are eventually reduced either with views from the worker's left sibling or parent in an appropriate order consistent with a serial execution. When a frame F_1 executing on W_1 terminates (i.e., returns), however, its sibling or parent F_2 may still be running, executed by another worker W_2. To avoid interfering with W_2 executing F_2, W_1 simply deposits its set of local views stored in F_1's user hypermap into F_2's left-child or right-sibling hypermap placeholder, depending on the relation between F_1 and F_2. We refer to the process of one worker depositing its local views into a frame running on another worker as ***view transferal***, which more generally, refers to the process of transferring ownership of local views from one worker to another.

Similarly, before W_1 can perform view transferal from F_1 to F_2, it may find a second set of local views stored in F_1's left-child or right-sibling hypermap placeholders. If so, W_1 must reduce the two sets of views together — iterate through each view from one hypermap, look up the corresponding view in another hypermap, and reduce the two views into a single view. This process is referred as the ***hypermerge*** process. Worker W_1 performs hypermerges until it has only one set of local views left to deposit. We omit details on how the runtime maintains the correct orders of reduction and refer readers to [12].

4. THREAD-LOCAL MEMORY MAPPING

This section describes thread-local memory mapping (TLMM) [22, 23], which Cilk-M uses to cause the virtual-memory hardware to map reducers to local views. TLMM provides an efficient and flexible way for a thread to map certain regions of virtual memory independently from other threads while still sharing most of its virtual-memory address space. This section reviews the functionality that the TLMM mechanism provides and how we implemented it in a Linux operating system.

Cilk-M's memory-mapping reducer mechanism employs the virtual-address hardware to map accesses to a given reducer to different local views depending on the worker performing the access. Different workers must be able to map different physical pages

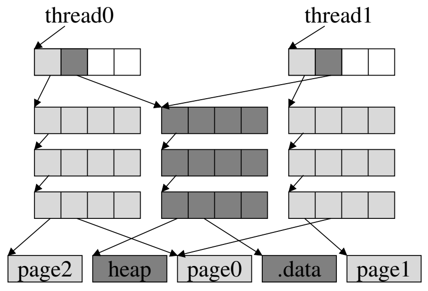


Figure 3: Example of a x86 64-bit page-table configuration for two threads on TLMM-Linux. The portion of the data structure dealing with the TLMM region is shaded light grey, and the remainder corresponding to the shared region is shaded dark grey. In the TLMM region, thread0 maps page2 first and then page0, whereas thread1 maps page1 first and then page0. The pages associated with the heap and the data segments are shared between the two threads.

within the same virtual address range so that the same global virtual address can map to different views for different workers. Given that a dynamic-multithreaded program typically executes on a shared-memory system, part of the address space must be shared to allow workers to communicate with each other and enable parallel branches of the user program to share data on the heap. In other words, this memory-mapping approach requires part of the virtual address space to be ***private***, in which workers can map independently with different physical pages, while the rest being ***shared***, in which different workers can share data allocated on the heap as usual.

By default, a traditional operating system does not provide such mixed sharing mode — either nothing is shared (each process has its own virtual-address space, and no two processes share the same virtual-address space), or everything is shared (all threads within a given process share the process's entire virtual-address space). TLMM designates a region, referred to as the ***TLMM region***, of a process's virtual-address space as private. This special TLMM region occupies the same virtual-address range for each worker, but each worker may map different physical pages to the TLMM region. The rest of the virtual-address space is shared among workers in the process as usual.

Cilk-M's TLMM mechanism [22, 23] was originally developed to enable a work-stealing runtime system to maintain a "cactus-stack" abstraction, thereby allowing arbitrary calling between parallel and serial code. To implement TLMM, we modified the Linux kernel, producing a kernel version we call ***TLMM-Linux***. The existing implementation is for Linux kernel 2.6.32 running on x86 64-bit CPU's, such as the AMD Opteron and Intel Xeon.

Figure 3 illustrates the design. TLMM-Linux assigns a unique root page directory to each thread in a process. The x86 64-bit page tables have four levels, and the page directories at each level contain 512 entries. One entry of the root-page directory is reserved for the TLMM region, which corresponds to 512 GBytes of virtual-address space, and the rest of the entries correspond to the shared region. Threads in TLMM-Linux share page directories that correspond to shared region. Therefore, the TLMM-Linux virtual-memory manager must synchronize the entries in each thread's root page directory but populate the shared lower-level page directories only once.

The TLMM mechanism provides a low-level virtual-memory interface organized around allocating and mapping physical pages. We omit the descriptions of its interface and refer interested readers to [23] for details. For the purpose of our discussion, simply note that the TLMM interface provides a way for workers to "name" a physical page using a ***page descriptor***, which is analogous to a file descriptor and accessible by all workers. The system call `sys_pmap` allows a worker to map its TLMM region with particular physical pages specified by an array of page descriptors. Thus, a worker can share its TLMM region with another worker by publicizing the page descriptors corresponding to the physical pages mapped in its TLMM region.

The TLMM mechanism supports the following system calls, which provides a low-level virtual-memory interface organized around allocating and mapping physical pages. `sys_palloc` allocates a physical page and returns its page descriptor. A page descriptor is analogous to a file descriptor, which "names" a physical page and can be accessed by any thread in the process. `sys_pfree` frees a page descriptor and its associated physical page. To control the physical-page mappings in a thread's TLMM region, the thread calls `sys_pmap`, specifying an array of page descriptors to map, as well as a base address in the TLMM region at which to begin mapping the descriptors. `sys_pmap` steps through the array of page descriptors, mapping physical pages for each descriptor to subsequent page-aligned virtual addresses, to produce a continuous virtual-address mapping that starts at the base address. A special page-descriptor value `PD_NULL` indicates that a virtual-address mapping should be removed.

We added the memory-mapping reducer mechanism to Cilk-M, which now utilizes the TLMM region for both the cactus stack and memory-mapped reducers. Since a stack naturally grows downward, and the use of space for reducers is akin to the use of heap space, at system start-up, the TLMM region is divided into two parts: the cactus stack is allocated at the highest TLMM address possible, growing downwards, and the space reserved for reducers starts at the lowest TLMM address possible, growing upwards. The two parts can grow as much as needed, since in a 64-bit address space, as a practical matter, the two ends will never meet.

5. THREAD-LOCAL INDIRECTION

This section describes how Cilk-M's memory-mapping strategy for implementing reducers exploits a level of indirection so that the reducer mechanism can handle a variety of reducers with different types, sizes, and life spans. We describe the problems that arise for a naive approach in which the TLMM region directly holds reducer views. We then describe how thread-local indirection solves these problems without unduly complicating the runtime system.

We first examine a seemingly straightforward approach for leveraging TLMM to implement reducers. In this scheme, whenever a reducer is declared, the runtime system allocates the reducer at a virtual address in the TLMM region that is globally agreed upon by all workers. The runtime system instructs each worker to map the physical page containing its own local view at that virtual address. Thus, accesses to the reducer by a worker operate directly on the worker's local view.

Although this approach seems straightforward, it fails to address two practical issues. First, the overhead of mapping can be great due to fragmentation arising from allocations and deallocations of reducers in the TLMM region. Second, performing a hypermerge of views in TLMM regions is complicated and may incur heavy mapping overhead. We discuss each of these issues in turn.

Regarding the first issue, if views are allocated within a TLMM region, the runtime system must manage the storage in the region separately from its normal heap allocator. Since reducers can be allocated and deallocated throughout program execution, the TLMM region can become fragmented with live reducer hyperobjects scattered across the region. Consequently, when a worker maps in physical pages associated with a different worker's TLMM region, as must occur for a hypermerge, multiple physical pages may need to be mapped in, each requiring two kernel crossings (from user mode to kernel mode and back). Even though the remapping overhead can be amortized against steals (and the Cilk-M runtime already performs a `sys_pmap` call upon a successful steal to maintain the cactus stack), if the number of `sys_pmap` calls is too great, the kernel crossing overhead can become a scalability bottleneck, outweighing the benefit of replacing the hash-table lookups of the hypermap approach with virtual address translations.

The second issue involves the problem of performing hypermerges. Consider a hypermerge of the local views in two workers W_1 and W_2, and suppose that W_1 is performing the hypermerge. To perform a monoid operation on a given pair of views, both views must be mapped into the same address space. Consequently, at least one of the views cannot be mapped to its appropriate location in the TLMM region, and the code to reduce them with the monoid operation must take that into account. For example, if W_2's view contains a pointer, W_1 would need to determine whether the pointer was to another of W_2's views or to shared memory. If the former, it would need to perform an additional address translation. This "pointer swizzling" could be done when W_1 maps W_2's views into its address space, but it requires compiler support to determine which locations are pointers, as well as adding a level of complexity to the hypermerge process.

Since "any problem in computing can be solved by adding another level of indirection,"[3], we shall employ ***thread-local indirection***. The idea is to use the TLMM region to store pointers to local views which themselves are kept in shared memory visible to all workers. When a reducer is allocated, a memory location is reserved in the TLMM region to hold a pointer to its local view. If no view has yet been created, the pointer is null. Accessing a reducer simply requires the worker to check whether the pointer is null, and if not, dereference it, which is done by the virtual-address translation provided by the hardware. In essence, the memory-mapping reducer mechanism replaces the use of hypermaps with the use of the TLMM region.

The two problems that plague the naive scheme are solved by thread-local indirection. The TLMM region contains a small, compact set of pointers, all of uniform size, which precludes internal fragmentation. The storage management of reducers is simple and avoids pointer swizzling. The TLMM region requires only a simple scalable[4] memory allocator for single-word objects (the pointers). Since local views are stored in shared memory, the job of handling them is conveniently delegated to the ordinary heap allocator. The residual problem of managing the storage for the pointers in the TLMM region is addressed in Section 6.

Thread-local indirection also solves the problem of one worker gaining access to the views of another worker in order to perform hypermerge. Since the local views are allocated in shared memory, a worker performing the hypermerge can readily access the local views of a different worker. The residual problem of determining which local views to merge is part of the view-transferal protocol, addressed in Section 7.

6. ORGANIZATION OF WORKER-LOCAL VIEWS

This section describes how Cilk-M organizes a worker's local views in a compact fashion. Recall that after a steal, the thief resuming the stolen frame starts with an empty set of views, and

[3]Quotation attributed to David Wheeler in [20].

[4]To be scalable, Cilk-M memory allocator for the TLMM region allocates a local pool per worker and occasionally rebalances the fixed-size slots among local pools when necessary in the manner of Hoard [5].

whenever the thief accesses a reducer for the first time, a new identity view is created lazily. Once a local view has been created, subsequent accesses to the reducer return the local view. Moreover, during a hypermerge, a worker sequences through two sets of local views to perform the requisite monoid operations. This section shows how a worker's local views can be organized compactly using a "sparse accumulator (SPA)" data structure [14] to support these activities efficiently. Specifically, we show the following:

- Given (the address of) a reducer hyperobject, how to support a constant-time ***lookup*** of the local view of the reducer.
- How to ***sequence*** through all of a worker's local views during a hypermerge in linear time and ***reset*** the set of local views to the empty set.

A traditional SPA consists of two arrays:[5] an array of values, and an array containing an unordered "log" of the indices of the nonzero elements. The data structure is initialized to an array of zeros at start-up time. When an element is set to a nonzero value, its index is recorded in the log, incrementing the count of elements in the SPA (which also determines the location of the end of the log). Sequencing is accomplished in linear time by walking through the log and accessing each element in turn.

Cilk-M implements the SPA idea by arranging the pointers to local views in a ***SPA map*** within a worker's TLMM region. A SPA map is allocated on a per-page basis, using 4096-byte pages on x86 64-bit architectures. Each SPA map contains the following fields:

- a ***view array*** of 248 elements, where each element is a pair of 8-byte pointers to a local view and its monoid,
- a ***log array*** of 120 bytes containing 1-byte indices of the valid elements in the view array,
- the 4-byte number of valid elements in the view array, and
- the 4-byte number of logs in the log array.

Cilk-M maintains the invariant that empty elements in the view array are represented with a pair of null pointers. Whenever a new reducer is allocated, a 16-byte slot in the view array is allocated, storing pointers to the executing worker's local view and to the monoid. When a reducer goes out of scope and is destroyed, the 16-byte slot is recycled. The simple memory allocation for the TLMM region described in Section 5 keeps track of whether a slot is assigned or not. Since a SPA map is allocated in a worker's TLMM region, the virtual address of an assigned 16-byte slot represents the same reducer for every worker throughout the life span of the reducer and is stored as a member field `tlmm_addr` in the reducer object.

A reducer lookup can be performed in constant time, requiring only two memory accesses and a predictable branch. A lookup entails accessing `tlmm_addr` in the reducer (first memory access), dereferencing `tlmm_addr` to get the pointer to a worker's local view (second memory access), and checking whether the pointer is valid (predictable branch). The common case is that the `tlmm_addr` contains a valid local view, since a lookup on an empty view occurs at most once per reducer per steal. As we shall see in Section 7, however, a worker resets its SPA map by filling it with zeros between successful steals. If the worker does not have a valid view for the corresponding reducer, the `tlmm_addr` simply contains zeros.

Sequencing through the views can be performed in linear time. Since a worker knows exactly where a log array within a page starts and how many logs are in the log array, it can efficiently sequence through valid elements in the view array according to the indices stored in the log array. The Cilk-M runtime stores pointers to a local view and the reducer monoid side-by-side in the view array, thereby allowing easy access to the monoid interface during the hypermerge process. In designing the SPA map for Cilk-M, we explicitly chose to have a 2 : 1 size ratio between the view array and the log array. Once the number of logs exceed the length of the log array, the Cilk-M runtime stops keeping track of logs. The rationale is that if the number of logs in a SPA map exceeds the length of its log array, the cost of sequencing through the entire view array, rather than just the valid entries, can be amortized against the cost of inserting views into the SPA map.

[5]For some applications, a third array is used to indicate which array elements are valid, but for our application, invalidity can be indicated by a special value in the value array.

7. VIEW TRANSFERAL

This section describes how a worker in Cilk-M can efficiently gain access to another worker's local views and perform view transferal efficiently. The Cilk-M runtime system includes an efficient view-transferal protocol that does not require extra memory mapping. This section brings all the pieces together and presents the complete picture of how the memory-mapping reducer mechanism works.

In the hypermap approach, view transferal simply involves switching a few pointers. Suppose that worker W_1 is executing a full frame F_1 that is returning. The worker simply deposits its local views into another frame F_2 executing on worker W_2 that is either F_1's left sibling or parent, at the appropriate hypermap placeholder. In the memory-mapping approach, more steps are involved. In particular, even though all local views are allocated in the shared region, their addresses are only known to W_1, the worker who allocated them. Thus, W_1 must ***publish*** pointers to its local views, making them available in a shared region.

There are two straightforward strategies for W_1 to publish its local views. The first is the ***mapping strategy***: worker W_1 leaves a set of page descriptors in frame F_2 corresponding to the SPA maps in its TLMM region, which W_2 then maps into its TLMM region to perform the hypermerge. The second strategy is the ***copying strategy***: W_1 copies those pointers from its TLMM region into a shared region. Cilk-M employs the copying strategy, because the number of reducers used in a program is generally small, and thus the overhead of memory mapping greatly outweighs the cost of copying a few pointers.

For W_1 to publish its local views, whose references are stored in the ***private SPA maps*** in its TLMM regions, W_1 simply allocates the same number of ***public SPA maps*** in the shared region and ***transfers*** views from the private SPA maps to the public ones. As W_1 sequences through valid indices in a view array to copy from a private SPA map to a public one, it simultaneously zeros out those valid indices in the private SPA map. When all transfers are complete, the public SPA maps contain all the references to W_1's local views, and the private SPA maps are all empty (the view array contains all zeros). Zeroing out W_1's private SPA maps is important, since W_1 must engage in work-stealing next, and the empty private SPA maps ensure that the stolen frame is resumed on a worker with an empty set of local views.

Since a worker must maintain space for public SPA maps throughout its execution, Cilk-M explicitly configures SPA maps to be compact and allocated on a per-page basis. Each SPA map holds up to 248 views, making it unlikely that many SPA maps are ever needed. Recall from Section 6 that the Cilk-M runtime system maintains the invariant that an entry in a view array contains either a pair of valid pointers or a pair of null pointers indicating that the entry is empty. A newly allocated (recycled) SPA map is empty.[6]

[6]To be precise, only the number of logs and the view array must contain zeros.

Name	*Description*
`add-n`	Summing 1 to x into n add-reducers in parallel
`min-n`	Processing x random values in parallel to find the min and accumulate the results in n min-reducers
`max-n`	Processing x random values in parallel to find the max and accumulate the results in n max-reducers

Figure 4: The three microbenchmarks for evaluating lookup operations. For each microbenchmark, the value x is chosen according to the value of n so that roughly the same number of lookup operations are performed.

The fact that a SPA map is allocated on the per-page basis allows the Cilk-M runtime system to recycle empty SPA maps easily by maintaining memory pools[7] of empty pages solely for allocating SPA maps.

In the memory-mapping approach, a frame contains placeholders to SPA maps, instead of to hypermaps, so that worker W_1 in our scenario can deposit the populated public SPA maps into F_2 without interrupting worker W_2. Similarly, a hypermerge involves two sets of SPA maps instead of hypermaps. When W_2 is ready to perform the hypermerge, it always sequences through the map that contains fewer view pointers and reduces them with the reduce operation into the map that contains more view pointers. After the hypermerge, one set of SPA maps contain pointers to the reduced views, whereas the other set (assuming they are public) are all empty and can be recycled. Similar to the transfer operation, when W_2 performs the hypermerge, as it sequences through the set with fewer views, it zeros out the valid views, thereby maintaining the invariant that only empty SPA maps are recycled.

View transferal in the memory-mapping approach incurs higher overhead than that in the hypermap approach, but this overhead can be amortized against steals, since view transferals are necessary only if a steal occurs. As Section 8 shows, even with the overhead from view transferal, the memory-mapping approach performs better than the hypermap approach and incurs less total overhead during parallel execution.

8. AN EMPIRICAL EVALUATION OF MEMORY-MAPPED REDUCERS

This section compares the memory-mapping approach for implementing reducers used by Cilk-M to the hypermap approach used by Cilk Plus. We quantify the overheads of the two systems incurred during serial and parallel executions on three simple synthetic microbenchmarks and one application benchmark. Our experimental results show that memory-mapped reducers not only admit more efficient lookups than hypermap reducers, they also incur less overhead overall during parallel executions, despite the additional costs of view transferal.

General setup. We compared the two approaches using a few microbenchmarks that employ reducers included in the Cilk Plus reducer library, as well as one application benchmark. Figure 4 describes the microbenchmarks, all of which are synthetic, which perform lookup operations repeatedly with simple REDUCER operations that perform addition, finding the minimum, and finding the maximum. The value n in the name of the microbenchmark dictates the number of reducers used, which is determined at compile-time. The value x is an input parameter chosen so that a given microbenchmark with different n performs 1024 million lookup operations, or 2048 million lookup operations in the case of the reduce overhead study.[8] The application benchmark is a parallel breadth-first search program [26] called PBFS.

All benchmarks were compiled using the Cilk Plus compiler version 12.0.0 with `-O2` optimization. The experiments were run on an AMD Opteron 8354 system with 4 quad-core 2 GHz CPU's having a total of 8 GBytes of memory. Each core on a chip has a 64-KByte private L1-data-cache, a 512-KByte private L2-cache, and a 2-MByte shared L3-cache.

Performance evaluation using microbenchmarks

Figure 5 shows the microbenchmark execution times for a set of tests with varying numbers of reducers running on the native Cilk Plus runtime system and the Cilk-M 1.0 runtime system. Figure 5(a) shows the execution times running on a single processor, whereas Figure 5(b) shows them for 16 processors. Each data point is the average of 10 runs with a standard deviation of less than 5%. Across all microbenchmarks, the memory-mapped reducers in Cilk-M consistently outperform the hypermap reducers in Cilk Plus, executing about 4–9 times faster for serial executions, and 3–9 times faster for parallel executions.

One interesting thing to note is that, every instance of `min-n` (for different n) took longer to execute than its corresponding counterpart of `max-n` on both runtime systems. At first glance, the two microbenchmarks `min-n` and `max-n` ought to perform similarly given that the same number of comparisons are executed in both microbenchmarks. That is not the case, however — due the to artifact of how reducer min and max libraries are implemented, more updates are performed on a given view in the execution of `min-n` than that in the execution of `max-n` for the same n. Thus, `min-n` took longer execution time than `max-n`.

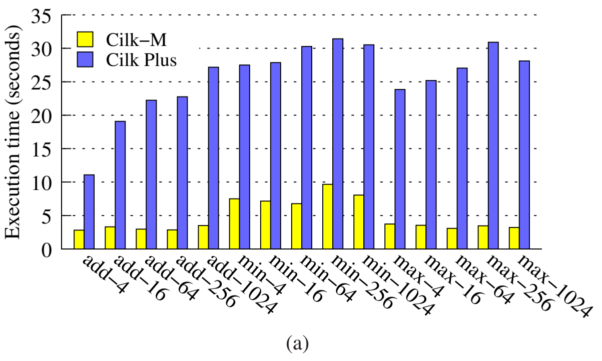
Lookup overhead. Figure 6 presents the lookup overheads of Cilk-M 1.0 and Cilk Plus on `add-n` with varying n. The overhead data was obtained as follows. First, we ran the `add-n` with x iterations on a single processor. Then, we ran a similar program called `add-base-n`, which replaces the accesses to reducers with accesses to a simple array, also running x iterations. Since hypermerges and reduce operations do not take place when executing on a single processor, `add-base-n` essentially performs the same operations as `add-n` minus the lookup operations. Figure 6 shows the difference in the execution times of `add-n` and `add-base-n` with varying n. Each data point takes the average of 10 runs with a standard deviation of less than 2% for Cilk-M and less than 12% for Cilk Plus.

The lookup overhead in Cilk-M stays fairly constant independent of n, but the lookup overhead in Cilk Plus varies significantly. This discrepancy can be understood by observing that a lookup operation in Cilk-M translates into two memory accesses and a branch irrespective of the value of n, whereas a lookup operation in Cilk Plus translates into a hash-table lookup whose time depends on how many items the hashed bucket happens to contain, as well as whether it triggers a hash-table expansion.

The reduce overhead during parallel execution. Besides the lookup overhead, we also studied the other overheads incurred by reducers during parallel executions. We refer to the overheads incurred only during parallel executions as the ***reduce overhead***, which includes overheads in performing hypermerges, creating views, and inserting views into a hypermap in Cilk Plus or a SPA map in Cilk-M. For Cilk-M, this overhead also includes view transferal. For both systems, additional lookups are performed during

[7]The pools for allocating SPA maps are structured like the rest of the pools for the internal memory allocator managed by the runtime. Every worker owns its own local pool, and a global pool is used to rebalance the memory distribution between local pools in the manner of Hoard [5].

[8]In the case of the reduce overhead study, we configured the microbenchmark to perform more lookup operations to prolong the execution time, because the runtime overhead measured in this study constitutes only a small part of the overall execution time.



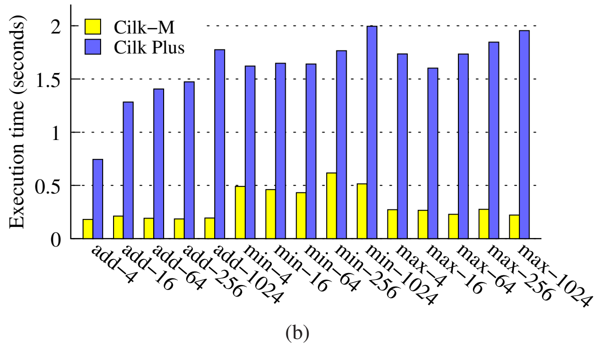


Figure 5: Execution times for microbenchmarks with varying numbers of reducers using Cilk-M and Cilk Plus, running on (a) a single processor and (b) on 16 processors, respectively.

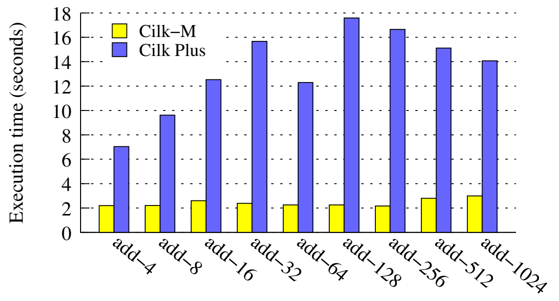


Figure 6: Reducer lookup overhead of Cilk-M and Cilk Plus running the microbenchmark using add reducers on a single processor. A single cluster in the *x*-axis shows the overheads for both systems for a given *n*, and the *y*-axis shows the overheads in execution time in seconds.

a hypermerge, and they are considered as part of the overhead as well.

Figure 7 compares the reduce overhead of the two systems. The data was collected by running `add-n` with varying *n* on 16 processors for both systems and instrumenting the various sources of reduce overhead directly inside the runtime system code. In order to instrument the Cilk Plus runtime, we obtained the open-source version of the Cilk Plus runtime, which was released with ports of the Cilk Plus language extensions to the C and C++ front-ends of gcc-4.7 [3]. We downloaded only the source code for the runtime system (revision 181962), inserted instrumentation code, and made it a plug-in replacement for the Cilk Plus runtime released with the official Intel Cilk Plus compiler version 12.0.0. This open-source runtime system is a complete runtime source to support the Linux operating system [3], and its performance seems comparable to the runtime released with the compiler. Given the high variation in

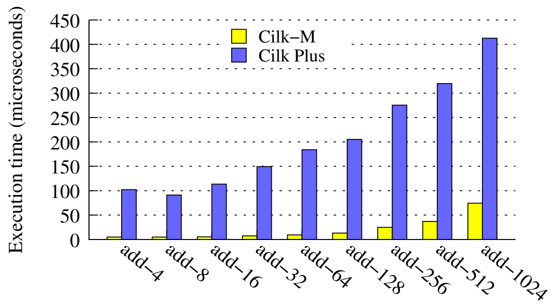


Figure 7: Comparison of the reduce overheads of Cilk-M and Cilk Plus running `add-n` on 16 processors. A single cluster in the *x*-axis shows the overheads for both system for a given *n*, and the *y*-axis shows the reduce overheads in milliseconds.

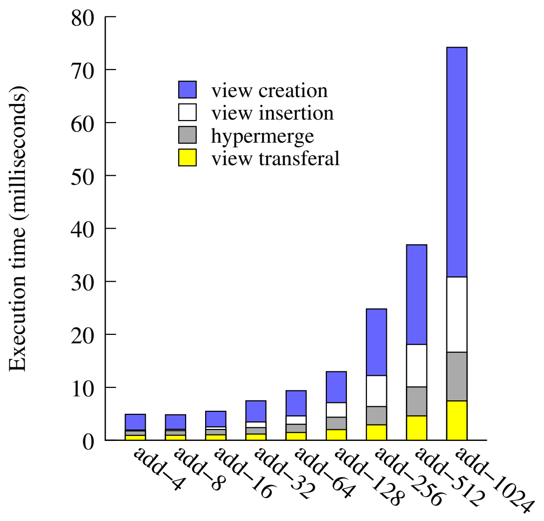


Figure 8: The breakdown of the reduce overhead in Cilk-M for `add-n` on 16 processors with varying *n*.

the reduce overhead when memory latency plays a role, the data represents the average of 100 runs. Since the reduce overhead is correlated with the number of (successful) steals, we also verified that in these runs, the average numbers of steals for the two systems are comparable.

As can be seen in Figure 7, the reduce overhead in Cilk Plus is much higher than that in Cilk-M. Moreover, the discrepancy increases with *n*, which makes sense, because a higher *n* means more views are created, inserted, and must be reduced during hypermerges. The overhead in Cilk Plus also grows much faster than that in Cilk-M. It turns out that the Cilk Plus runtime spends much more time on view insertions (inserting newly created identity views into a hypermap), which dominates the reduce overhead, especially as *n* increases. Thus, Cilk Plus incurs a much higher reduce overhead, even though the Cilk-M runtime has the additional overhead of view transferal. In contrast, Cilk-M spends much less time on view insertions than Cilk Plus. A view insertion in Cilk-M involves writing to one memory location in a worker's TLMM region, whereas in Cilk Plus, it involves inserting into a hash table. Moreover, a SPA map in Cilk-M stores views much more compactly than does a hypermap, which helps in terms of locality during a hypermerge.

Figure 8 breaks down the reduce overhead for Cilk-M. Overhead is attributed to four activities: view creation, view insertion, view transferal, and hypermerge, which includes the time to execute the

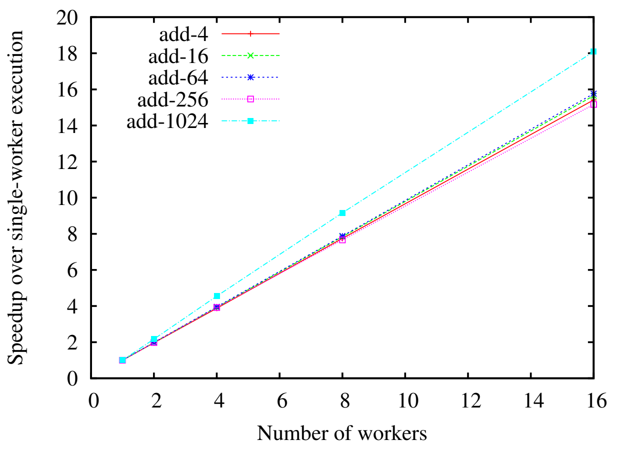


Figure 9: Speedups of `add-n` executing on Cilk-M with 1, 2, 4, 8, and 16 processors. The x-axis shows P, the number of processors used. The y-axis is the speedup, calculated by dividing the execution time running on a single processor with the execution time running on P processors.

monoid operation. As can be seen from the figure, the overhead due to view transferal grows rather slowly as n increases, demonstrating that the SPA map allows efficient sequencing. Furthermore, the dominating overhead turns out to be view creations, showing that the design choices made in the memory-mapping approach do indeed minimize overhead.

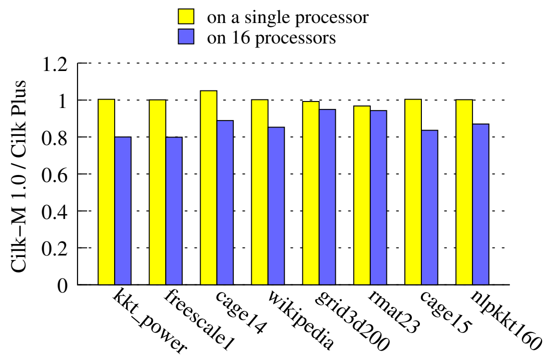
Figure 9 shows the speedups for `add-n` when executing on 1, 2, 4, 8, and 16 workers. To simplify the presentation, only five different n values are shown — 4, 16, 64, 256, and 1024. As can be seen from Figure 9, despite the reduce overhead during parallel executions, the scalability is not affected. All instances of `add-n` (with different n) have good speedups with `add-1024` having superlinear speedup. As mentioned earlier, the reduce overhead is correlated with the number of successful steals and can be amortized against steals. As long as the application has ample parallelism and that the number of reducers used is "reasonable,"[9] as in the case of `add-n`, scalability of the application will not be affected by the reduce overhead.

Performance evaluation using PBFS

We evaluated the two runtime systems on a parallel breath-first search application called PBFS [26]. Given an input graph $G(V,E)$ and a starting node v_0, the PBFS algorithm finds the shortest distance between v_0 and every other node in V. The algorithm explores the graph "layer-by-layer," alternating between two "bag" data structures for insertion. As the program explores the nodes stored in one bag, all of which belong to a common layer, it inserts newly discovered nodes from the next layer into another bag. The bags are declared to be reducers to allow parallel insertion.

Figure 10(a) shows the relative execution time between Cilk-M and Cilk Plus on a single processor and on 16 processors. Since the work and span of a PBFS computation depend on the input graph, we evaluated the relative performance with 8 input graphs whose characteristics are shown in Figure 10(b). These input graphs are the same ones used in [26] to evaluate the algorithm. For each data point, we measured the mean of 10 runs, which has a standard deviation of less than 1%. Figure 10(a) shows the mean for Cilk-M normalized by the mean for Cilk Plus.

[9]It is possible to write an application to use large number of reducers in such a way that the reduce overhead dominates the total work in the computation. In such case, the reduce overhead will affect scalability. This topic is investigated in more detail in further work by Lee [22, Ch. 5].



(a)

Name	$\|V\|$	$\|E\|$	D	*# of lookups*
`kkt_power`	2.05M	12.76M	31	1027
`freescale1`	3.43M	17.1M	128	1748
`cage14`	1.51M	27.1M	43	766
`wikipedia`	2.4M	41.9M	460	1631
`grid3d200`	8M	55.8M	598	4323
`rmat23`	2.3M	77.9M	8	71269
`cage15`	5.15M	99.2M	50	2547
`nlpkkt160`	8.35M	225.4M	163	4174

(b)

Figure 10: (a) The relative execution time of Cilk-M to that of Cilk Plus running PBFS on a single processor and on 16 processors. Each value is calculated by normalizing the execution time of the application on Cilk-M with the execution time on Cilk Plus. (b) The characteristics of the input graphs for parallel breath-first search. The vertex and edge counts listed correspond to the number of vertices and edges.

For single-processor executions, the two systems performed comparably, with Cilk-M being slightly slower. Since the number of lookups in PBFS is extremely small relative to the input size, the lookups constitute a tiny fraction of the overall work (measured by the size of the input graph). Thus, it is not surprising that the two systems perform comparably for serial executions. On the other hand, Cilk-M performs noticeably better during parallel executions, which is consistent with the results from the microbenchmarks. Since the reduce overhead in Cilk-M is much smaller than that in Cilk Plus, PBFS scales better.

9. RELATED WORK

Traditional virtual-memory mechanisms have been described in the literature to support various linguistic abstractions. This section summarizes this work and describes how each uses a virtual-memory mechanism to implement its respective linguistic abstraction.

Abadi et al. [1] describes how one can efficiently support "strong atomicity" in a software transactional memory (STM) [32] system using a traditional virtual-memory mechanism supported by standard hardware. An STM system implements ***strong atomicity*** [8] if the system detects conflicts between transactions as well as conflicts between a transaction and a ***normal*** memory access performed outside of a transaction. Supporting strong atomicity may incur significant overhead, however, if the system must also keep track of every normal memory access. To avoid such overhead, Abadi et al. propose an alternative approach to leverage the page protection mechanism provided by the virtual-memory hardware, mapping the heap space twice, one for normal accesses and one for transactional accesses. When a page is being read or written to by a transaction, the system revokes certain access permissions on its corresponding mapping allocated for normal accesses, thereby detecting potential conflicts at the page granularity.

Berger et al. [6] propose Grace, a runtime system that eliminates concurrency errors and guarantees deterministic executions for multithreaded computations based on fork-join parallelism. Grace employs a ***threads-as-processes*** paradigm, where a thread seen by a user program running on Grace is in fact implemented as a process. Since processes do not share virtual address space, this paradigm enables different processes in Grace to map the shared regions (i.e., global variables and heap) with different access permissions, thereby detecting potential currency errors such as races, deadlocks, and atomicity violations at the page granularity. Updates to the shared regions are buffered (via copy-on-write mappings) and committed at "logical thread boundaries" in a deterministic order only when the updates do not cause a "conflict" with existing values. Upon a successful commit, updates are reflected in shared mappings and become globally visible.

Liu et al. [27] present Dthreads, which has the same goal as Grace, to prevent concurrency errors and enforce deterministic executions on multithreaded computations. The Dthreads runtime system adopts the same threads-as-processes paradigm as Grace and leverages the virtual memory mechanism similarly. Dthreads differs from Grace in three ways, however. First, Dthreads supports most general-purpose multithreaded programs, whereas Grace supports only fork-join parallelism. Second, Dthreads supports the full synchronization primitives implemented by POSIX threads [16], whereas Grace does not. Finally, Dthreads resolves "conflicts" among threads deterministically using a last-writer-wins protocol, whereas Grace executes parallel branches speculatively and must roll back upon conflict detection. Consequently, Dthreads enables better performance than Grace.

Finally, Pyla and Varadarajan [30] describe Sammati, a language-independent runtime system that provides automatic deadlock detection and recovery. Like the aforementioned works, Sammati employs the same threads-as-processes paradigm and leverages the virtual-memory hardware to allow processes to employ different accesses permissions on address space allocated for shared memory. Unlike the other work, Sammati focuses on automatic deadlock detection and recovery but not on deterministic executions. Thus, Sammati does not enforce a deterministic ordering in when updates are committed. Moreover, the system assumes that the program is written correctly — even though Sammati can detect races involving two writes to the same location, it cannot detect races involving a read and a write.

Like the Cilk-M research described here, each of these three studies describes how virtual-memory hardware can support a particular linguistic mechanism. One distinct difference between these studies and Cilk-M's memory-mapped reducers is that they employ traditional virtual-memory mechanisms supported by existing operating systems, whereas Cilk-M utilizes thread-local memory mapping (TLMM), which enables each thread to map part of the virtual address range independently while preserves sharing in the rest of the address space.

10. CONCLUSION

Recently, concurrency platforms have begun to offer high-level "memory abstractions" to support common patterns of parallel-programming. A ***memory abstraction*** [22] is an abstraction layer between the program execution and the memory that provides a different "view" of a memory location depending on the execution context in which the memory access is made. For instance, ***transactional memory*** [15] is a type of memory abstraction — memory accesses dynamically enclosed by an `atomic` block appear to occur atomically. Arguably, the Grace [6] and Dthreads [27] systems described in Section 9 are also examples of memory abstractions — every memory access is buffered and eventually committed (i.e., becomes globally visible) in some deterministic order. The cactus-stack mechanism implemented in Cilk-M [22, 23] provides another example of a memory abstraction. Reducer hyperobjects are yet another memory abstraction for dynamic multithreading.

This paper has laid out a new way of implementing reducers, namely, through use of the TLMM mechanism. As demonstrated in Section 8, experimental results show that the memory-mapping approach admits an efficient implementation. Interestingly, it appears that TLMM can be used to implement Grace and Dthreads with lower runtime overhead, which suggests that the TLMM mechanism may provide a general way for building memory abstractions.

With the proliferation of multicore architectures, the computing field must move from writing serially executing software to parallel software in order to unlock the computational power provided by modern hardware. Writing parallel programs, however, gives rise to a new set of challenges in how programs interact with memory, such as how to properly synchronize concurrent accesses to shared memory. We believe that investigating memory abstractions is a fruitful path. Memory abstractions ease the task of parallel programming, directly by mitigating the complexity of synchronization, and indirectly by enabling concurrency platforms that utilize resources more efficiently.

Acknowledgments

Thanks to Matteo Frigo for tips and insights on the reducers implementation in Cilk++. Thanks to Pablo Halpern of Intel, one of the original designers of reducers and a Cilk Plus developer, for helpful discussions on the implementation of reducers in Cilk Plus. Thanks to the Cilk team at Intel and the Supertech Research Group at MIT CSAIL for their support.

11. REFERENCES

[1] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 185–196, Raleigh, NC, USA, 2009. ACM.

[2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification Version 1.0*. Sun Microsystems, Inc., March 2008.

[3] Balaji Ayer. Intel® Cilk^{TM} Plus is now available in open-source and for GCC 4.7! `http://www.cilkplus.org`, 2011. The source code for the compiler and its associated runtime is available at `http://gcc.gnu.org/svn/gcc/branches/cilkplus`.

[4] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The habanero multicore software research project. In *Proceeding of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, OOPSLA '09, pages 735–736, Orlando, Florida, USA, 2009. ACM.

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-LX)*, pages 117–128, Cambridge, MA, November 2000.

[6] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented*

Programming Systems Languages and Applications, OOPSLA '09, pages 81–96, Orlando, Florida, USA, 2009. ACM.

[7] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin, 1999.

[8] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[9] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habenero-Java: the new adventures of old X10. In *PPPJ*. ACM, 2011.

[10] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005.

[11] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.

[12] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-First Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, Calgary, Canada, August 2009. Won Best Paper award.

[13] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[14] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl*, 13:333–356, 1992.

[15] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Conference on Computer Architecture. (Also published as ACM SIGARCH Computer Architecture News, Volume 21, Issue 2, May 1993.)*, pages 289–300, San Diego, California, 1993.

[16] Institute of Electrical and Electronic Engineers. Information technology — Portable Operating System Interface (POSIX) — Part 1: System application program interface (API) [C language]. IEEE Standard 1003.1, 1996 Edition.

[17] Intel Corporation. *Intel® Cilk$^{\text{TM}}$ Plus Application Binary Interface Specification*, 2010. Revision: 0.9.

[18] Intel Corporation. *C++ and C interfaces for Cilk reducer hyperobjects*. Intel Corporation, 2011. Intel® C++ Compiler 12.0: `reducer.h` Header File.

[19] Intel Corporation. *Intel® Cilk$^{\text{TM}}$ Plus Language Specification*, 2011. Revision: 1.1.

[20] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10:265–310, November 1992.

[21] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 36–43. ACM, 2000.

[22] I-Ting Angelina Lee. *Memory Abstractions for Parallel Programming*. PhD thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, June 2012. To be submitted in February 2012.

[23] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 411–420, Vienna, Austria, September 2010. ACM.

[24] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2007. Available from `http://msdn.microsoft.com/magazine/`.

[25] Charles E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, March 2010.

[26] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 303–314, June 2010.

[27] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, Cascais, Portugal, 2011. ACM.

[28] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[29] OpenMP application program interface, version 3.0. OpenMP specification, May 2008.

[30] Hari K. Pyla and Srinidhi Varadarajan. Avoiding deadlock avoidance. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 75–86, Vienna, Austria, 2010. ACM.

[31] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[32] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, Ottowa, Ontario, Canada, August 1995.

[33] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX '92*, pages 1–9, 1992.