Software Caching and Computation Migration in Olden

Martin C. Carlisle* and Anne Rogers
Department of Computer Science
Princeton University
{mcc,amr}@cs.princeton.edu

TR-483-95

Abstract

The goal of the *Olden* project is to build a system that provides parallelism for general purpose C programs with minimal programmer annotations. We focus on programs using dynamic structures such as trees, lists, and DAGs. We demonstrate that providing both software caching and computation migration can improve the performance of these programs, and provide a compile-time heuristic that selects between them for each pointer dereference. We have implemented a prototype system on the Thinking Machines CM-5. We describe our implementation and report on experiments with ten benchmarks.

1 Introduction

Olden is a continuing project whose goals are to build a compiler and runtime system for C programs on distributed-memory SPMD machines, automatically detecting parallelism, and inserting communication as much as possible. We specifically focus on programs using recursive data structures. To date, little work has been done to address the problem of supporting these programs. Although work has been done on supporting SPMD execution of array-based programs, these techniques do not extend to programs using recursive structures, because they rely on the fact that arrays are statically defined and directly addressable. Recursive data structures, however, are dynamically defined and must be recursively traversed to be addressable.

In prior work [35], we described a new execution model for supporting programs that use pointer-based dynamic structures and described results on a preliminary implementation. That model used a simple mechanism for migrating the thread of control based on the layout of heap-allocated data and introduced parallelism using a technique based on futures and lazy task creation. Although we observed good performance on several benchmarks, we noticed that the model performed poorly in situations where the results of subcomputations on different processors were being merged. In these situations, the thread of computation tended to "ping-pong" between the two processors, and the resultant communication cost outweighed the available parallelism.

We, as have other researchers [21, 32], determined that both computation migration and caching need to be supported to provide good performance. Computation migration takes advantage of the locality of nearby objects in the data structure; while caching allows multiple objects on different processors to be accessed more efficiently. Unlike these other projects, where the selection between mechanisms must be made by the programmer, Olden provides a compile-time heuristic that for each pointer dereference selects the appropriate mechanism to be used. To assist the compiler in making this choice, we provide an extension to the language, *path-affinities*, that gives information about the expected layout of the data.

*Supported, in part, by a National Science Foundation Graduate Fellowship, the Fannie and John Hertz Foundation, and NSF FAW award MIP-9023542.

Since distributed-memory machines do not support caching directly, we implemented a software caching scheme. Providing a cache requires a mechanism for maintaining coherence. We use a local invalidation scheme where the cache is cleared at certain synchronization points. Our caching method relies on the insertion of pointer test code by the compiler and uses active messages [38] for communication between processors.

In Section 2, we give a brief overview of the Olden programming model, and in Section 3, describe Olden's computation migration and software caching mechanisms. Then, in Section 4, we present path-affinities and the heuristic used by the compiler to choose between computation migration and software caching. In Section 5, we report results for ten benchmarks using our implementation on the Thinking Machines CM-5 and then in Section 6, contrast Olden with related work. Finally, we give conclusions and suggestions for future work in Section 7.

2 Programming Model

The Olden model and its computation migration mechanism are described in detail in a prior paper [35]. In this paper, we will provide only a brief overview of the context of our work.

Olden uses an SPMD programming model. Each processor has an identical copy of the program, as well as a local stack that is used to store procedure arguments, local variables and return addresses. Additionally, there is a distributed heap where each processor owns one section. We view heap addresses as consisting of a pair of a processor name and a local address $\langle p, l \rangle$. This information is encoded in a single 32-bit word.

Olden takes as input a program written in a restricted subset of C, with some additional Olden-specific annotations. We require that programs do not take the address of stack-allocated objects, which ensures that all pointers point into the heap.[1] The major differences between our programming model and the standard sequential model are that the programmer explicitly chooses a particular strategy to map the dynamic data structures over the distributed heap, and annotates work that can be done in parallel using futures [20].

In general, the computation will tend to follow the data, so to get good performance, the programmer must place related pieces of data on the same processor explicitly. Mapping the data to the processors is achieved by including a processor number in each allocation request. Olden provides a library routine, `ALLOC`, that allocates memory on a specified processor, and returns a pointer that encodes both the processor name and the local address of the allocated memory.

Additionally, Olden uses futures to indicate opportunities for parallelism. The programmer may mark a procedure call as a *futurecall*, if it may be evaluated safely in parallel with its parent context. A *touch* must also be inserted by the programmer before the return value is used. Olden's implementation of futures is to save the futurecall's context (return continuation) on a work list, and evaluate the body directly. New threads are generated only if a migration occurs during the execution of the body of the future. In this case, the now idle processor will grab a continuation from the work list and start executing it; this is called *future stealing*. If no migration occurs, then the function will complete and we will avoid paying the overhead of creating a new thread.

The programmer-specified data layout heavily affects the performance of the futures. If large groups of related data are placed together, Olden will generate a small number of large granularity tasks, and thus be more efficient. A simple example of such a data layout for a balanced tree is to distribute sub-trees at some fixed depth equally among the processors. In this case, one thread will be generated for each subtree at that depth. These threads should have large granularity, and good load balancing.

Given a suitably restricted C program using `ALLOC` and futures as specified, the Olden compiler will generate an SPMD program that correctly handles references to global heap pointers and calls the Olden library routines as needed. These routines manage the threads, and perform the requisite communication between the processors.

[1] We do provide structure return values, which can be used to handle many of the cases where & is needed.

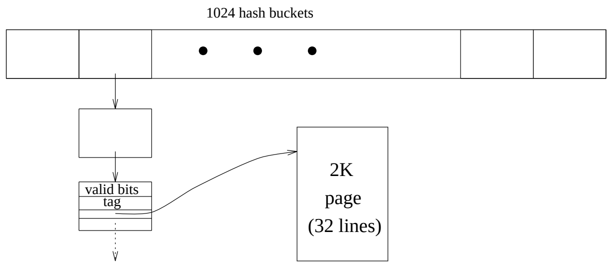

Figure 1: Olden's software cache

3 Remote Data Access

As previously mentioned, Olden provides two mechanisms for accessing data on a remote processor: computation migration and software caching.

3.1 Computation migration

The basic idea of computation migration is that when a thread executing on Processor P attempts to access a location residing on Processor Q, the thread is migrated from P to Q. To make this affordable, we send only the portion of the thread's state that is necessary for the current procedure to complete execution: the registers, program counter, and current stack frame. When it is time to return from the procedure, control is returned to Processor P. To accomplish this, Q sets up a special return stub to be used in place of the return to the caller. The stub migrates the thread back to P by sending a message containing the return address, return frame pointer, and the registers. Note that the stack frame is not returned, as it is no longer needed. Processor P can then complete the procedure return by restarting the thread at the return address.

Before each heap reference that uses computation migration, the Olden compiler inserts an explicit check to distinguish if it is local or remote and translate the address. The local versus remote check extracts the processor name from the heap address and checks it against the local processor's name. A remote reference is handled by a call to the migration routine from the Olden library.

3.2 Software Caching

The software caching mechanism is very similar to the caching scheme in Blizzard-S [37]. Each processor uses its local memory as a large, fully-associative, write-through cache. As in Blizzard-S, we perform allocation on the page level, and perform transfers at the line level.[2] The main difference between Olden's cache and that in Blizzard-S is that we cannot rely on virtual memory support. Consequently, rather than using a very large virtual table to translate addresses (pages of this table are allocated on demand by trapping page faults), we instead use a 1K hash table with a list of pages kept in each bucket. Figure 1 shows the translation table. Since entries are kept on a per-page basis, the chains in each bucket will tend to be quite short (in our experience, the average chain length is approximately one.)

The Olden compiler directly inserts software checks before each heap reference that uses software caching. The table lookup is very similar to that of Blizzard-S, with the addition of searching the lists stored in the hash table. In addition to checking the state bit for the line, as done in Blizzard-S, our lookup also returns a tag used to translate the address from a global to a local pointer (in Blizzard-S this translation is performed automatically by the virtual memory hardware). For both systems, in the event that the page is not allocated or the line is not valid, the appropriate allocation or transfer is performed by a library routine.

[2] In Olden, a page is 2K bytes, and a line 64 bytes.

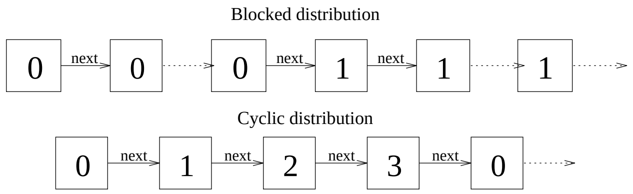

Figure 2: Two different list distributions. The numbers in the boxes are processor numbers. Dotted arrows represent a sequence of list items.

Once we introduce a local cache at each processor, we must provide a means to ensure that no processor sees stale data. The most intuitive coherence model is *sequential consistency* [27]. It may be summarized by stating that each processor performing a read on a location sees the most recently completed write to that location. Sequential consistency may be provided by the underlying system, or guaranteed by having invalidations in the code. A coherence mechanism that requires no communication is called a *local knowledge* scheme. Prior local knowledge schemes (e.g., [12, 15, 31]) have relied on the compiler specifically inserting code to invalidate the local cache. We implement a local knowledge scheme in Olden using the runtime system, by having each processor invalidate its entire cache upon receiving a migration. Since Olden's synchronization mechanisms require that concurrent threads do not interfere, a program running under this scheme provably will have the same semantics as the same program running on a sequentially consistent system (we sketch this proof in Appendix A). We have made a slight improvement to this scheme, by noticing that on returns we need only invalidate cached copies of lines from processors whose memories have been written by the returning thread. (Since the thread cannot read data written by other concurrently executing threads, it need only invalidate cache lines that it might have updated while on a remote processor.) This scheme will perform well when most shared data is written between migrations. Appendix A describes two other coherence mechanisms that we explored and compares their performance to our local knowledge scheme.

4 Selecting a Mechanism

Given these two mechanisms, the compiler must decide, for each pointer dereference, which it will choose for accessing remote data. Our goal is to minimize the total communication cost over the entire program. Consequently, although an individual thread migration is substantially more expensive than performing a single remote fetch (by a factor of about seven on the CM-5), it may still be desirable to pay the cost of the migration, if moving the thread will convert many subsequent references into local references. Consider a list of N elements, evenly divided among P processors (two possible configurations are given in Figure 2). First suppose the list items are distributed in a block fashion. A traversal of this list will require $N\frac{(P-1)}{P}$ remote accesses if software caching is used, but only $P-1$ migrations if the computation is allowed to follow the data. Hence it is better to use computation migration for such a data layout. Caching, however, performs better when the list items are distributed using a cyclic layout. In this case, using computation migration will require $N-1$ migrations, whereas caching requires $N\frac{(P-1)}{P}$ remote accesses.

Olden uses a three-step process to select a mechanism for each program point. First, the programmer specifies *path-affinities*, which give hints to the compiler regarding the layout of the data. Second, a data flow analysis is used to find pointers that are traversing the data structure in a regular manner. In each loop (either iterative or recursive), at most one such variable is selected for computation migration. Finally, interactions between loops are considered, and additional variables are marked for caching if it is determined that using computation migration for them may cause a bottleneck.

```
while (s) {
  s = s->left;
  t = t->right->left;
  u = s->right;
}
```

(Assume path-affinity of left is 90 and right is 70).

Update Matrix (after \ before)	s	t	u
s	90		
t		63	
u	70		

Figure 3: A simple loop with induction variables

4.1 Path-affinity hints

Since the communication cost of a particular mechanism for a particular program fragment is highly dependent on the layout of the data, we allow the programmer to provide a quantified hint to the compiler regarding the layout of a recursive data structure. Each pointer field of a structure may be marked with a path-affinity that represents the probability that a path (i.e., a traversable sequence of pointers in the data structure) along that field will be local. For example, if a field, `F`, has a path-affinity of 70%, that would indicate that a path along field `F` in the data structure would be expected to cross a processor boundary 30% of the time. Consider again the examples given in Figure 2. In the blocked case, the path-affinity of the `next` field is $1 - \frac{P-1}{N-1}$ (of the $N-1$ `next` pointers, $P-1$ of them point to an object on a different processor). In the cyclic case, the `next` field has a path-affinity of zero (each `next` pointer is to an object on a different processor). This example also illustrates the intuition behind our heuristic's use of path-affinities. In general, computation migration is preferable for high affinity paths, and software caching for low affinity paths.

Note that the path-affinities supplied by the programmer are merely hints, and may be omitted (in which case a default value is used), approximated, or even wrong without affecting program correctness.

4.2 Update matrices

We want to be able to estimate how the program will, in general, traverse its recursively-defined structures. To accomplish this, we examine the loops and recursive calls (hereafter referred to as *control loops*) checking how pointers are updated in each iteration. We say that `s` is updated by `t` along field `F` in a given loop, if the value of `s` at the end of an iteration is the value of `t` from the beginning of the iteration dereferenced through field `F` (i.e., `s'=t->F`). This notion extends directly to paths of fields. Intuitively, variables that are updated by themselves in a control loop will traverse the data structure in a regular fashion. We call such variables *induction variables*. In Figure 3, `s` and `t` are induction variables (since `s'=s->left` and `t'=t->right->left`), whereas `u` is not (since its value cannot be written as a path from its value in the previous iteration).

We summarize information on possible induction variables in an *update matrix*. The entry at location `(s,t)` of the matrix is the path-affinity of the update, if `s` is updated by `t`, and is blank otherwise. In Figure 3, since `s` is updated by itself along the field `left`, the entry `(s,s)` in the update matrix is 90 (the affinity of the `left` field). Induction variables are then simply those pointers with entries along the diagonal (i.e., they have been updated by themselves). In our example, the induction variables `s` and `t` have entries on the diagonal. We will consider only these for possible computation migration, as they traverse the structure in a regular manner.

The update matrices may be computed using standard data-flow methods. (Note again that exact or conservative information is not needed, as errors in the update matrices will not affect program correctness.) The only complications are that variables may have multiple updates or update paths of length greater than one. There are three cases. The first is a join point in the flow graph (e.g., at the end of an `if-then` statement). Here we simply merge the two updates from each branch by taking the average of their affinities. This corresponds to assuming each branch is taken about half of the time, and could be improved with better branch prediction information. If the update does not appear in both branches, then rather than averaging the update, we omit it. We do this because we wish to consider only those updates that occur in every iteration of the loop, thus guaranteeing

```
int TreeAdd (tree *t) {
  if (t == NULL) return 0;
  else
    return TreeAdd(t->left)+
           TreeAdd(t->right)+
           t->val;
}
```

(Assume path-affinity of left is 90 and right is 70).

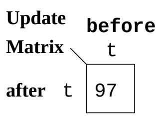

Figure 4: TreeAdd

that the updated variable is actually traversing the structure.

Second, we must have a rule for multiple updates via recursion. Consider the simple recursive program in Figure 4. Note that `t` has two updates, one corresponding to each recursive call. The two recursive calls form a control loop. This control loop does not include the join at the end of the `if-then-else` statement, as the recursive calls occur before the end of the `else` branch. Therefore, the merging rule described above does not apply for the control loop, and the update of `t` is not omitted. In this case, we define the path-affinity of the update as the probability that either of the updates will be along a local path (since both are going to be executed). If we assume the path-affinity of `left` is 90% and `right` is 70%, then the probability that both are remote is 3% (assuming independence). Consequently, the path-affinity of the update of `t` because of the recursive calls is 97% (the probability that at least one will be local). We will further motivate this choice in Section 4.3.

The final possiblity is an update path of length greater than one (e.g., `t=t->left->left`). The path-affinity is then simply the product of the path-affinities of each field along the path.

So far, we have only discussed computing update matrices intraprocedurally. A full interprocedural implementation would need to be able to compute paths generated by the return values of functions, and handle control loops that span more than one procedure (e.g., a mutual recursion). Our preliminary implementation performs only a limited amount of interprocedural analysis. In particular, we do not consider return values, or analyze loops that span multiple procedures. Although this preliminary implementation is sufficient for all of our benchmarks, we plan to extend it to a full interprocedural analysis using access path matrices [22].

4.3 The heuristic

Once the update matrices have been computed, the heuristic uses a two-pass process to select between computation migration and software caching. First, each control loop is considered in isolation. Then, in the second phase, we consider the interactions between nested control loops, and possibly decide to do additional caching. In addition to having the update matrix for each control loop, we also assume each control loop has a notation indicating if it is parallelizable. The Olden compiler checks for the presence of futures to determine if a control loop may be parallelized.

In the first pass, for each control loop, we select the induction variable whose update has the strongest path-affinity. If a control loop has no induction variable, then it will select computation migration for the same variable as its parent (the smallest control loop enclosing this one). If the path-affinity of the selected variable's update exceeds a certain threshhold, or the control loop is parallelizable, then computation migration is chosen for this variable; otherwise, dereferences of this variable are cached. Dereferences of all other pointer variables are cached. We select computation migration for parallelizable loops with path-affinities below the threshhold because this mechanism allows us to generate new threads. (As described in Section 2, in Olden, new threads are only generated following a migration).

The threshhold, and the default path-affinity have been set to 90%[3] and 70%, respectively. These values were chosen so that, by default, list traversals will use caching, tree traversals will use computation migration, and tree searches will use caching. The averaging method for recursive calls was also designed to obtain this behavior. In our experience, these decisions provide the best

[3] Since the cost of a migration is about seven times that of a cache miss, the break-even path-affinity is about 86%.

```
Traverse(tree *t) {
  if (t==NULL) return;
  else {
    Traverse(t->left);
    Traverse(t->right);
  }
}

WalkAndTraverse(list *l, tree *t) {
  for each body, b, in l do in parallel {
    Traverse(t);
  }
}
```

```
Walk(list *l) {
  while (l) {
    visit(l);
    l=l->next;
  }
}

TraverseAndWalk(tree *t) {
  if (t==NULL) return;
  else {
    do in parallel {
      TraverseAndWalk(t->left);
      TraverseAndWalk(t->right);
      }
  Walk(t->list);
  }
}
```

Figure 5: Code examples with and without a bottleneck

performance most of the time. In those cases where the defaults are not appropriate, the programmer can specify path-affinity hints. (We do not allow the programmer to modify the threshhold, but the same effect can be obtained by modifying the path-affinities). We explicitly specified path-affinities in only three of the ten benchmarks (TSP, Perimeter, and MST).

Considering control loops in isolation does not yield the best performance. Inside a parallel loop, it is possible to generate a bottleneck by using computation migration. Consider the codes in Figure 5. `WalkAndTraverse` is a procedure that for each list item traverses the tree. If computation migration were chosen for the traversal of the tree, the parallel threads for each item in the list would be forced to serialize on their accesses to the root of the tree, which becomes a bottleneck. In `TraverseAndWalk`, for each node in the tree, we walk the list stored at that node. Since there is a different list at each node of the tree, the parallel threads at different tree nodes are not be forced to serialize, and there is no bottleneck. In general, a bottleneck occurs whenever the initial value of the induction variable of the inner loop is the same over a large number of iterations of the outer loop. Returning to the examples, in `WalkAndTraverse`, `t` has the same value for each iteration of the parallel `for` loop, while in `TraverseAndWalk`, `t->list` has a different value in each iteration, as `t->list` has a different value at each node in the tree. Athough in general this is a difficult aliasing problem, we do not need exact or conservative information. If incorrect information is used, the program will run correctly, but possibly more slowly than if more precise information were available. Our current approximation tests to see if the induction variable for the inner loop is updated in the parent loop. If so, we assume no bottleneck will occur; otherwise, we use caching in the inner loop to avoid the possibility of a bottleneck. Once the heuristic has analyzed the interactions between loops, the selection process is complete.

5 Experimental Results

We have implemented Olden on a Thinking Machines CM-5. The input to Olden is a C program annotated with futures, touches, calls to Olden's allocation routine, and data structure affinity information. Our system consists of a compiler for the annotated C code and a runtime system. The compiler is an adaptation of **lcc** [17], an ANSI C compiler, that generates code for testing for pointer locality and for handling futures and touches. It also includes our heuristic for choosing between thread migration and software caching. The runtime system is written in a combination of C and SPARC assembly code.

This section summarizes preliminary results from using Olden on a suite of ten benchmarks. Table 1 briefly describes each benchmark. Table 2 lists the running time of a sequential implementation plus speed-up numbers for up to 32 processors for each benchmark. The numbers reported represent averages over three runs done in dedicated mode using the local knowledge cache coherence scheme. We report whole program times (W) for three benchmarks, Power,[4] Barnes, and Health, to allow

[4] The speedups for Power exceed those reported in prior work [35] due to an improved implementation of migration.

Table 1: Benchmark Descriptions

Benchmarks	Description	Problem Size
TreeAdd	Adds the values in a tree	1024K nodes
Power	Solves the Power System Optimization problem [30]	10,000 customers
TSP	Computes an estimate of the best hamiltonian circuit for the Traveling-salesmen problem[24]	32K cities
MST	Computes the minimum spanning tree of a graph[6]	1K nodes
Bisort	Sort by creating two disjoint bitonic sequences and then merging them[8]	128K integers
Voronoi	Computes the Voronoi Diagram of a set of points[19]	64K points
EM3D	Simulates the propagation electro-magnetic waves in a 3D object[14]	2K nodes
Barnes-Hut	Solves the N-body problem using hierarchical methods [5]	8K bodies
Perimeter	Computes the perimeter of a set of quad-tree encoded raster images [36]	4K x 4K image
Health	Simulates the Columbian health care system [29]	1365 villages

for comparison with published results. We report kernel times only for the rest to avoid having their data structure building phases, which show excellent speed-up, skew the results. We use a true sequential implementation compiled with our compiler for computing speedups. The speed-up numbers for the one-processor version give a measure of Olden's overhead. These experiments were performed using CM-5s at two National Supercomputing Centers: NCSA at the University of Illinois and NPAC at Syracuse University.

The benchmarks fall into two categories: those that use only migration (M) and those that use both migration and caching (M+C). TreeAdd, Power, TSP, and MST use tree-based algorithms that have simple data access patterns. Consequently, the Olden compiler chooses to use migration alone to satisfy remote references in these programs. TreeAdd and Power both have good performance.[5] They do not show perfect speedup, because of unavoidable overhead from testing pointers, handling futures and touches, and managing the stack. Also, tree-based algorithms naturally show less parallelism near the top of the tree. TSP performs well, but not quite as well as TreeAdd and Power. It uses uses a divide-and-conquer algorithm as they do, but unlike them its merge phase is non-trivial. Each merge is sequential and walks through the subtrees, which requires a migration for each participating processor. Using software caching in place of migration would increase rather than decrease the cost of communication for this application, because a large amount of data is accessed on each processor during the subtree walk. The performance for MST is poor and degrades sharply as the number of processors increases, because the number of migrations is $O(NP)$. Caching would not reduce the communication costs for this program, because these migrations serve mostly as a mechanism for synchronization.

The remaining six benchmarks use a combination of migration and software caching. Bisort performs two sorts, one forward and one backward, on a randomly generated set of integers. The data is stored in a binary tree. The algorithm creates a bitonic sequence in each subtree and then merges them to obtain the sorted result. The merge phase swaps subtrees to create two disjoint bitonic sequences and then performs two recursive calls. Swapping the trees rather than pointers to the trees is expensive, but helps maintain locality, which is crucial to the performance of the second sort and subsequent uses of the data. A pair of pointers is used to search the subtrees during the merge phase. The migration heuristic is designed to use software caching for tree searches, so dereferences to these pointers use caching. The tree swaps, on the other hand, use migration, because a large amount of data is touched on each processor between migrations.

Voronoi is a classic geometric divide-and-conquer algorithm. The points are stored in a binary tree sorted by x-coordinate. The algorithm computes the Voronoi diagram of the two subtrees and then merges them. The merge phase walks along the convex hull of the two sub-diagrams, and adds edges to knit them together to form the Voronoi diagram for the whole set. Walking along the

[5] Our efficiency on 64 processors is about 80%, compared to 75% by Lumetta et al. [30].

Table 2: Results

Benchmarks	Heuristic choice	Sequential time (sec.)	Speed-up by number of processors 1	2	4	8	16	32	Migrate-only Speedup (32)
TreeAdd	M	4.49	0.73	1.47	2.93	5.90	11.81	23.4	
Power^W	M	286.59	0.96	1.94	3.81	6.92	14.85	27.5	
TSP	M	43.35	0.95	1.92	3.70	6.70	10.08	15.8	
MST	M	9.81	0.96	1.36	2.20	3.43	4.56	5.14	
Bisort	M+C	31.41	0.73	1.35	2.29	3.52	4.92	6.33	6.13
Voronoi	M+C	49.73	0.75	1.38	2.41	4.23	6.88	8.76	0.47
EM3D	M+C	1.21	0.86	1.51	2.69	4.48	6.72	12.0	0.05
Barnes-Hut^W	M+C	555.79	0.74	1.42	3.00	5.29	8.13	11.2	$<$0.01
Perimeter	M+C	2.47	0.86	1.70	3.37	6.09	9.86	14.1	2.96
Health^W	M+C	34.19	0.73	1.47	2.93	5.72	11.09	16.42	16.52

^{W} – Whole program times

convex hull of a single subresult is best done with migration, but the merge phase walks along two subresults, alternating between them in an irregular fashion. As a result, the heuristic chooses to pin the computation on the processor that owns the root of one of the subresults and use software caching to bring remote subresults to the computation. This version performs dramatically better than an early version that used only migration [35]. The heuristic does not make the optimal choice in this situation; it would be better to traverse one of the subresults while caching the other (such a version has a speed-up of over 12 on 32 processors). We are exploring ways to improve the heuristic to handle this case better.

EM3D models the propagation of electromagnetic waves in a 3D object, which is represented as a bipartite graph containing **E** nodes and **H** nodes. At each time step, new values for the **E** nodes are computed from a weighted sum of the neighboring **H** nodes, and then the same is done for the **H** nodes. The main computation loop consists of walking down a list of nodes, reading the values from the neighbors and using them to update the current node. The heuristic chooses to use migration for the nodes, because they have high locality, and to use software caching for the edges, because they have low locality. Our implementation performs comparably to the ghost node implementation of Culler et al. [14], yet does not require substantial modification to the sequential code.

Barnes-Hut simulates the evolution of bodies in a gravitational system. This computation is broken into three pieces: building the tree used to represent the particles, calculating new accelerations for the particles by walking the tree, and then computing the new positions of the particles. In our implementation, the tree building phase is sequential and starts to represent a substantial fraction of the computation as the number of processors increases. Factoring out this cost, we achieve a speed-up of over 19 on 32 processors. The migration heuristic chooses a combination of migration and caching for the first two phases: migration to send computation to the processor that owns the particle; caching to bring "distant" tree nodes to the computation as needed. Migration is chosen for the particles, because they have high locality. Software caching is chosen for tree even though it has high locality to avoid causing a bottleneck at its root. Migration alone suffices for computing the new positions of the particles, again because they have high locality. Falsafi et al. [16] give results for six different implementations of this benchmark; our results using their parameters (approximately 36 secs/iter) fall near the middle of their range (from 15 to 80 secs/iter).

Perimeter uses a quad-tree encoding of the raster image. The algorithm superficially looks similar to TreeAdd, but traverses the tree in a very different way when computing the contribution of neighboring quadrants. The heuristic chooses to use caching when determining the neighbors of a quadrant, because they may be far away in the tree.

Health simulates the Columbian health care system [29] using a four-way tree. Each node of the tree represents a hospital, and at each node there is a list of patients. At each timestep, the tree is traversed, and patients, once assessed, are either treated or passed up the tree to the parent. The heuristic, according to its design, chooses migration for the tree traversal, and caching to access remote items in the lists. Although Health uses the same synchronization as MST, we obtain a better speedup as there is more work done in each iteration. Since the number of patients at each node that arrive from a remote processor is small (less than two percent), the additional overhead of maintaining the cache outweighs the benefit of caching.

6 Related Work

Our work in Olden spans two different areas of parallel computing: providing support for programming and maintaining multiprocessor cache coherence. In this section we describe how our work relates to that of other groups in each of these areas. Both are very active areas of research; out of necessity we restrict our discussion to papers that seem most relevant.

6.1 Programming models

Linda [9] provides a tuple-space mechanism for distributed processors to work on a shared linked structures. This model provides a global shared address space, but no control over the actual assignment of data to processors.

Emerald [23] and Amber [11] are object-oriented languages that employ thread and object migration mechanisms to improve locality. These languages provide primitives for object location and mobility, and constructs to allow the programmer to indicate whether the thread or the object(s) should move to satisfy an invocation that references a remote object.

Prelude [21], is an explicitly parallel language that provides a computation model based on threads and objects. Annotations are added to a Prelude program to specify which of several mechanisms — remote procedure call, object migration, and computation migration — should be used to implement an object or thread.

Orca[4] also provides an explicitly parallel programming model based on threads and objects. Orca hides the distribution of the data from the programmer, but is designed to allow the compiler and runtime system to implement shared objects efficiently. The Orca compiler produces a summary of how shared objects are accessed that is used by its runtime system to decide if a shared object should be replicated, and if not, where it should be stored. Operations on replicated and local objects are processed locally; operations on remote objects are handled using a remote procedure call to the processor that owns the object.

Split-C [14] is a parallel extension of C that provides a global address space and maintains a clear concept of locality by providing both local and global pointers. Split-C provides a variety of primitives to manipulate global pointers efficiently. In a related piece of work, Lumetta et al. [30] describe a global object space abstraction that provides a way to decouple the description of an algorithm from the description of an optimized layout of its data structures.

The Concert system [13] provides compiler and runtime support for efficient execution of fine-grained concurrent object-oriented programs. Concert provides a globally shared object space, common programming idioms (such as RPC and tail forwarding), inheritance, and some concurrency control. Objects are single threaded and communicate asynchronously through message passing (invocations).

Cid [32], a recently proposed extension to C, supports a threads and locks model of parallelism. Cid threads are lightweight and the thread creation mechanism allows the programmer to name a specific processing element on which the thread should be run. Unlike Olden, Cid threads cannot migrate once they have begun execution. This makes it awkward to take advantage of data locality while traversing a structure iteratively. Cid also provides a global object mechanism that is based on global pointers. The programmer explicitly requests access to a global object using one of several sharing modes (for example, readonly) and is given a pointer to a local copy in return. Cid's global objects use implicit locking and the runtime system maintains consistency.

6.2 Cache coherence

Early multiprocessors tended to have caches separated from a set of memories by a bus. An invalidation protocol is a common way to maintain coherence. A write to a shared line causes an invalidation signal to be sent on the bus. The processors snoop the bus looking for such transactions and invalidate lines in their local caches as necessary. More recently proposed multiprocessors are built from processor-memory pairs interconnected through a network and use directories instead of snooping [3].

Li and Hudak [28] implemented *shared virtual memory*, a system that supports sequentially consistent transparent shared memory with locks in software. Their system, Ivy, uses a directory based scheme to manage coherence on the page level. Later systems, such as Munin [10], seek to reduce communication by using relaxed consistency models, such as *release consistency* [18]. Petersen and Li [33] and Konthothanassis and Scott [25] implement release consistency using the operating system's virtual memory mechanisms.

Most papers that use the term "software coherence" have referred to the insertion of invalidation instructions by a compiler. Darnell and Kennedy [15], Cheong and Veidenbaum [12] and Min and Baer [31] propose a variety of local coherence mechanisms for FORTRAN. A comparison of software and hardware schemes was done by Adve et al. [1].

Olden's coherence scheme is an adaptation of these ideas to our programming model. We obtain relaxed consistency by performing coherence events only at migrations, and as in Blizzard [37], provide coherence at the cache-line level. The local knowledge scheme is related to compiler invalidation.

7 Conclusions

We have presented a new mechanism for automatically selecting between computation migration and caching for explicitly parallel programs that use recursive dynamic data structures on message-passing machines. We performed experiments on ten benchmarks using our prototype implementation on the CM-5. Our results indicate that the heuristic makes good selections with minimal programmer input, and that by combining computation migration with software caching, we can obtain significant improvements in performance.

The performance improvements gained by combining computation migration and software caching are not limited to distributed memory machines. Programs run on both networks of workstations and recently proposed hybrid shared-memory/message-passing machines, such as Alewife[2], FLASH[26], and Typhoon/Tempest[34], could benefit from the combination. Implementations of Olden for such machines would use different thresholds for choosing between computation migration and caching. The threshold for a network of workstations would favor computation migration, because of its high communication latency, whereas the threshold for machines with extensive hardware support would favor caching.

We are in the process of porting Olden to the Tempest interface. Our first Tempest implementation will run on top of Blizzard on the CM-5. This will allow us to take advantage of fine-grain access control to make pointer tests cheaper and examine the tradeoff of this overhead against the cost of processing interrupts on misses.

References

[1] S. Adve, V. Adve, M. Hill, and M. Vernon. Comparison of hardware and software cache coherence schemes. In *ISCA*, pages 298–308, 1991.

[2] A. Agarwal et al. The MIT Alewife machine: A large scale distributed-memory multiprocessor. Technical Report MIT/LCS TM-454, MIT, 1991.

[3] A. Agarwal, R. Simoni, M. Horowitz, and J. Hennessy. An evaluation of directory schemes for cache coherence. In *ISCA*, pages 280–289, 1988.

[4] H. Bal, M. F. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. on Software Engineering*, 18(3):190–205, March 1992.

[5] J. Barnes and P. Hut. A hierarchical $(o)(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[6] J. Bentley. A parallel algorithm for constructing minimum spanning trees. *J. of Algorithms*, 1:51–59, 1980.

[7] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *IEEE Computer Society Intl. Conference*, pages 524–533, February 1993.

[8] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM J. Comput.*, 18(2):216–228, 1989.

[9] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in Linda. In *PoPL*, pages 236–242, January 1986.

[10] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *SOSP*, pages 152–164, 1991.

[11] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *SOSP*, pages 147–158, December 1989.

[12] H. Cheong and A. Veidenbaum. A cache coherence scheme with fast selective invalidation. In *ISCA*, pages 299–307, 1988.

[13] A. Chien, V. Karamcheti, and J. Plevyak. The Concert system– compiler and runtime support for efficient, fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-1815, Dept. of Comp. Sci., Univ. of Illinois at Urbana-Champaign, June 1993.

[14] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273, 1993.

[15] E. Darnell and K. Kennedy. Cache coherence using local knowledge. In *Supercomputing*, pages 720–729, 1993.

[16] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing*, 1994.

[17] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, Redwood City, CA, 1995. ISBN 0-8053-1670-1.

[18] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA*, pages 15–26, May 1990.

[19] L. Guibas and J. Stolfi. General subdivisions and voronoi diagrams. *ACM Trans. on Graphics*, 4(2):74–123, 1985.

[20] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501–538, October 1985.

[21] W. Hsieh, P. Wang, and W. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *PPoPP*, pages 239–248, 1993.

[22] J. Hummel, L. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *PLDI*, pages 218–229, June 1994.

[23] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems*, 6(1):109–133, 1988.

[24] R. Karp. Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Mathematics of Operations Research*, 2(3):209–224, August 1977.

[25] L. Kontothanassis and M. Scott. High performance software coherence for current and future architectures. Technical report, Dept. of Comp. Sci., Univ. of Rochester, September 1994.

[26] J. Kuskin et al. The stanford FLASH multiprocessor. In *ISCA*, pages 302–313, April 1994.

[27] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9), September 1979.

[28] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[29] G. Lomow, J. Cleary, B. Unger, and D. West. A performance study of Time Warp. In *SCS Multiconference on Distributed Simulation*, pages 50–55, February 1988.

[30] S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing: The development of a parallel program. In *Supercomputing*, pages 243–249, 1993.

[31] S. Min and J. Baer. Design and analysis of a scalable cache coherence scheme based on clocks and timestamps. *IEEE Trans. on Parallel and Distributed Systems*, 3(1):25–44, January 1992.

[32] Rishiyur Nikhil. Cid: A parallel, "shared-memory" C for distributed-memory machines. In *Workshop on Languages and Compilers for Parallel Computing*, 1994.

[33] K. Petersen and K. Li. Cache coherence for shared memory multiprocessors based on virtual memory support. In *Intl. Parallel Processing Symp.*, April 1993.

[34] S. Reinhardt, J. Larus, and D. Wood. Tempest and typhoon: User-level shared memory. In *ISCA*, 1994.

[35] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Trans. on Programming Languages and Systems*, 1995. Also available as Princeton CS-TR-447-94 via anonymous ftp.

[36] H. Samet. Computing perimeters of regions in images represented by quadtrees. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-3(6), November 1981.

[37] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain access control for distributed shared memory. In *ASPLOS*, October 1994.

[38] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *ISCA*, May 1992.

[39] M. Zekauskas, W. Sawdon, and B. Bershad. Software write detection for a distributed shared memory. In *USENIX Symp. on Operating Systems Design and Implementation*, pages 87–100, 1994.

A Maintaining Cache Coherence

Due to space constraints, we described only one cache coherence scheme in the main text of the paper. This appendix discusses the issue of coherence in more detail, presenting two alternative coherence mechanisms and a quantitative analysis of their relative performance.

Because it is quite expensive to implement sequential consistency, many relaxed models have been proposed [7, 10, 18]. These models rely on having sufficient synchronization in the program to provide the illusion of sequential consistency. In Olden, we have implemented two other relaxed coherence models in addition to the local knowledge scheme discussed earlier. We demonstrate the correctness of each of these by relating them to a common relaxed model, *release consistency* [18].

In a release-consistent program, the programmer inserts acquires and releases for different locks. The system guarantees that after a lock is acquired, the processor performing the acquire sees all of the writes that occurred before the previous release of that lock. Release consistency maps nicely onto Olden, as each migration sent may be viewed as releasing a lock, and each migration received as acquiring one. Since the semantics of Olden's futures requires that no thread can read a data item while another thread is writing it, the "virtual locks" may be perceived as locking all of the data that is written by the thread. An Olden program running with any coherence protocol that provides release consistency with respect to these virtual locks will have the same semantics as the same program running on a sequentially consistent system.

Olden's local knowledge scheme invalidates its entire cache on receiving a migration (i.e., on each acquire). Consequently, the processor performing the acquire will see all of the writes that occurred before the previous release, and thus have the same semantics as on a sequentially consistent system.

Our second coherence protocol is an adaptation of *eager release consistency* [10]. In an eager release-consistent program, at each release all of the information regarding updates is forwarded immediately to other processors, and acknowledgements are collected before the release is allowed to complete. We have implemented eager release consistency in Olden by having the compiler insert code to track each write into the heap. These writes are tracked at the line level, by keeping a vector of *dirty bits* for each shared page. Additionally, the runtime system tracks sharers when it receives cache requests. Just as we performed allocations at the page level, we also track sharers at the page level, to reduce the amount of state information. At each migration, the runtime system sends invalidations to each sharer of a page, indicating which lines have been written on that page. This does not introduce false sharing, since writes are tracked at the line level, but could cause some spurious invalidation messages to be sent (i.e., an invalidation of a line might be sent to a processor that does not share that line). This scheme may be referred to as a *global knowledge* scheme, as invalidations are performed on the basis of knowledge obtained from other processors.

Finally, we implemented a *bilateral scheme*, that combines both local and global knowledge. Again, the compiler inserts code to track writes; however, sharers are not tracked. Instead, a timestamp is kept for each page. This timestamp is incremented when a migration leaves a processor, if the page has been written. On receiving a migration, a processor marks all of its pages, so that they miss on the first access. (This is similar to the use of epoch bits, as proposed by Darnell et al. [15].) On this miss, the handler sends the page's timestamp to its home, and is informed which lines need to be invalidated. The bilateral scheme greatly reduces the amount of invalidation traffic; however, it still must pay the cost of tracking writes. Consequently, the bilateral scheme should perform best on codes where there is a large amount of data that is read-only across several migrations, so that the savings in misses will outweigh the extra cost of tracking writes.

Table 3: Caching Statistics on 32 processors

Benchmarks	Cachable Writes # (1000s)	% Remote	Cacheable Reads # (1000s)	% Remote	% of Remote references that miss: local	global	bilateral	Total Pages Cached
Bitonic Sort	8,208	0.045	32,617	0.054	28.6	24.9	29.2	1604
Voronoi	9,825	1.57	42,359	1.26	5.89	5.89	5.89	2982
EM3D	0	0	839	19.4	6.18	6.18	6.18	1995
Barnes-Hut	2,707	18.3	73,601	55.6	0.815	0.563	0.792	21749
Perimeter	0	0	1,018	2.02	8.80	8.63	8.80	502
Health	8,861	0.063	33,405	0.019	87.0	10.3	87.0	163

Table 3 gives statistics on the behavior of the three coherence schemes. For most of our benchmarks, the three schemes behave basically identically, as almost all shared data is written between migrations. Barnes-Hut has a slightly smaller number of misses for the global and bilateral schemes. Bitonic Sort and Perimeter both have a slightly smaller number of misses only for the global scheme. The percentage of misses in Health decreases dramatically by adding global knowledge, but since the total number of misses is small the local knowledge scheme has the lowest running time. Overall, since the number of unnecessary misses generated by the overzealous invalidation of the local scheme is quite small (i.e., less than two percent of the total number of cacheable references), and the overhead of tracking writes is substantial (seven instructions for non-shared pages, and twenty-three instructions for shared pages)[6] the local knowledge scheme has the best running times for our benchmark suite.

Although it may seem surprising that Olden programs run faster with such a coarse coherence scheme, it follows logically from the way Olden uses data. To get good performance, the programmer must lay out the data so that related data is stored on the same processor. Also, since threads are guaranteed not to interfere, we are able to postpone the invalidations significantly. Given these considerations, and the fact that the benchmarks do not have much long-term read-only data, it is not surprising that the local knowledge scheme outperfoms the others.

[6] This could possibly be reduced using techniques similar to those by Zekauskas et al. [39]; however, the extra overhead would still exceed the caching performance gain for our benchmarks.