Algorithms for parallel execution of programs with memory aliasing

Thomas S. Anantharaman
March 3, 1987
CMU-CS-87-109

This research is sponsored by the Defense Advanced Research Projects Agency, DoD, through ARPA Order 5167, and monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-0163. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the United States Government.

Table of Contents

List of Figures

Abstract

Vectorizing compilers that parallelize programs written in declarative languages like Fortran usually have difficulty dealing with memory aliasing which causes the parallelism of the program to depend on input data not available at compile time. Compilation algorithms which can handle memory aliasing involving single levels of indirection by using sophisticated synchronization primitives have appeared recently, but we show here that these algorithms produce speed up only for certain values of the input data, and actually asymptotically slow down execution speed for worst case input data. We introduce a new compilation algorithm based on another synchronization primitive which for the same forms of memory aliasing as before produces speedups which are always within a factor of 2 of ideal data flow execution with greedy scheduling, except that the maximum speedup will be limited by the underlying architecture. Extensions of the algorithm to more complex forms of memory aliasing including multiple levels of indirection and conditionals are also described.

1. Introduction

Vectorizing compilers can be used to translate sequential code written in ordinary declarative languages like Fortran, Pascal or C into parallel code suitable for vector processors as well as MIMD based processors [4, 6, 3]. Many of the underlying compilation algorithms determine instructions which can be executed in parallel by evaluating the dependencies between the instructions [4]. In the case of instructions referencing memory these dependencies will depend on the address of the memory location referenced. When these addresses depend on input data not known at compile time, it may not be possible to tell if two memory references refer to the same memory location or are unrelated. This problem is referred to as memory aliasing and most vectorizing compilers in such cases assume the worst forcing sequential execution of the memory references. In some of these cases the memory references are actually provably independent, but the proof can be quite complex [1]. Here, however, we are concerned with cases where the memory references cannot be proven independent. Consider for example the loop in Figure 1-1 (written in 'C' like code).

```
int A[N],B[N];
"read in array B[0..N-1]";
for(i=0;i<N;i++){
   A[B[i]] = ...;
}
```

Figure 1-1: An example of memory aliasing

If the values read into array B[] are all different, then the assignments to array A[] in the loop will be independent and can be executed in parallel. However if the values read into array B[] have repeated values, certain assignments in the loop will be to the same location and must be executed in sequence to produce the correct final values. This paper assumes that the compiler does not attempt to modify the code wholesale

based on higher level semantic analysis, and restricts itself to reordering instructions in such a way as to increase parallelism while producing the same final values in memory.

In this paper we examine compilation algorithms that allow parallel execution even in such cases. Such algorithms can also be applied when memory references are independent but where this independence is very hard to prove. Section 2 describes the model of the parallel architecture which forms the basis of the compilation algorithms. Section 3 describes a compilation algorithm proposed by Zhu and Yew [8, 9] which is capable of executing loops with memory aliasing in parallel, provided the aliased memory references are restricted to a single level of indirection (as in Figure 1-1). It makes use of a new and sophisticated synchronization primitive. It is shown why this algorithm only works well in the best case when few dependencies are encountered at run time, but produces an asymptotically worse execution time than the original sequential code in the worst case when few memory references are independent. Section 4 describes a new compilation algorithm which uses another new synchronization primitive to produce parallel code whose performance is in some sense optimal irrespective of the actual data encountered at run time, assuming as before that aliased memory references are restricted to a single level of indirections. The maximum parallelism is restricted by an architecture dependent quantity, and hardware cost grows as N^2 where N is the maximum parallelism. Section 5 describes a modification of the algorithm which allows the hardware cost to grow as N*log(N) but slightly sacrifices the optimality of the parallel execution. Section 6 describes an extension of the algorithm to aliased memory references with multiple levels of indirections. Section 7 outlines an efficient hardware implementation of the synchronization primitive used in these algorithms. Section 8 concludes with a summary and a description of some remaining problems in dealing with memory aliasing.

2. The architecture model

The architecture model which forms the basis of the vectorization/parallelization algorithms in this paper, is illustrated in Figure 2-1. A number of logically identical processors are connected to a (possibly different) number of memory modules, and each processor can access any memory module.

Memory references are assumed to be atomic and take a constant amount of time for a given architecture configuration, except when multiple references are to the same memory module, in which case the references are sequentialized in some arbitrary order. As long as the processor memory network does not suffer from the problem of hot spots (see for example [5]) this holds for most kinds of processor memory interconnection schemes.

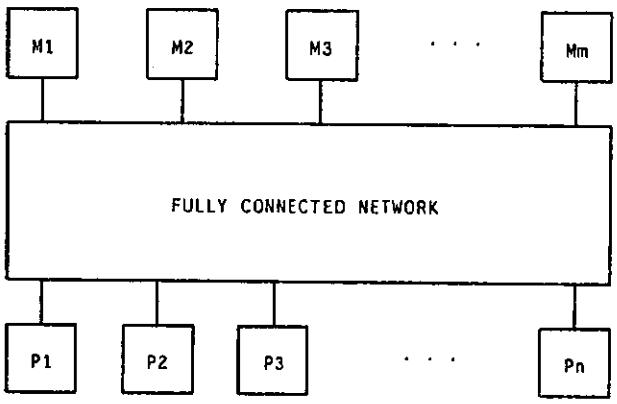


Figure 2-1: Model of the shared memory architecture

3. Previous work on run time resolution of memory aliasing

A new synchronization primitive for use in the Cedar multiprocessor was proposed by Zhu and Yew [8, 9], and the various examples of the use of this synchronization algorithm include an algorithm to parallelize loops with aliased memory references.

Cedar's synchronization primitive [8] uses a synchronization word per data item. The atomic synchronization operation is defined as follows :

```
if (test on key)* then
    operation on data item;
    operation on synchronization key;
end
```

The test on the synchronization key is a comparison with an integer (or no test). The operation on the key is a simple operation like increment, decrement, add, fetch, fetch&add, store or no action. The operation on the data item is a read or write. If the test fails either the process blocks until the test succeeds (indicated by a Kleene star on the test) or the operation on key and data are not executed and return a failure code. In the following it is assumed that the atomic synchronization operation takes a constant amount of time.

The most general case of memory aliasing considered in [9] is a (nested) loop which has memory references whose addresses are not known at compile time and contains if/then/else or case constructs. Only a single level of indirection is allowed for the aliased memory references, ie the address of each aliased memory reference is known at run time by the time the loop of interest is entered. This case is illustrated by the C program segment outlined in Figure 3-1, in which only the aliased memory references whose addresses cannot be evaluated at compile time are shown as references to the array A[]. The array indices $f_n(i)$ (n = 1..6) are input data dependent expressions that do not depend on array A[], such as B[i], C[i], where B[], C[], are arrays computed or input at run-time, before the loop is entered. The loop index i could be a constant size vector,

allowing for nested loops.

```
float A[N];
for(i=0;i<N;i++){
  ... = ... A[f1(i)];
  ... = ... A[f2(i)];
  if(...A[f3(i)]...){
    ... = ... A[f4(i)];
  } else {
    A[f5(i)] = ...;
  }
  A[f6(i)] = ...;
}
```

Figure 3-1: General loop with aliased memory references

If the array locations referenced by different iterations are all different the iterations could be run in parallel (instructions within each iteration would still be executed sequentially). Zhu and Yew's compilation algorithm attempts to optimise performance for this case. It generates synchronization instructions to go with each array access such that iterations can be scheduled in parallel. During an actual run the iterations will execute in parallel if and only if this is consistent with the data encountered at run time. The algorithm produces a modified program which may be described as follows :

Repeat the following steps until all iterations have executed.

1. Initialize all key values of aliased memory locations to infinity.

2. In parallel execute all aliased memory references for all iterations. However, instead of actually performing reads or writes to the aliased memory locations, the iteration index i is stored in the location's synchronization key whenever it is *less* than the value already present. This will result in each aliased memory location's key containing the *least* unexecuted iteration i which accesses it.

3. Execute in parallel all (unexecuted) iterations. Execute any particular iteration only if all aliased memory locations accessed have their synchronization key equal to i (ie this iteration is the least unexecuted iteration accessing the aliased memory locations in question).

The algorithm dynamically determines the reference pattern (RP) at each data dependent memory location, determines the first entry of this RP (during step 2), and postpones (during step 3) all references that do not occur in the same iteration as the first entry of the RP.

For the program segment in Figure 3-1 the modified code with synchronization is shown in Figure 3-2. Each location of array A[] now stores both data as well as a synchronization key. The atomic cedar synchronization operations are shown in the form {test:keyop,dataop}. The loop construct forall is like the loop construct for, except that all iterations may be executed in parallel. There is an implicit global

synchronization after every **forall**, to ensure that all parallel iterations have terminated.

```
struct {int key;
        float data;} A[N];

/* mark all iterations as unexecuted */
int D[N],done = 0;
forall(i=0;i<N;i++) D[i] = 0;

while(!done){
/* initialize keys */
    forall(i=0;i<N;i++) A[i].key = INFINITY;
/* locate least unexecuted iteration accessing each location of A[] */
    forall(i=0;i<N;i++)   if(!D[i]){
    {i<A[f1(i)].key : A[f1(i)].key = i; }
    {i<A[f2(i)].key : A[f2(i)].key = i; }
    {i<A[f3(i)].key : A[f3(i)].key = i; }
    {i<A[f4(i)].key : A[f4(i)].key = i; }
    {i<A[f5(i)].key : A[f5(i)].key = i; }
    {i<A[f6(i)].key : A[f6(i)].key = i; }
  }
/* execute iterations accessing only valid locations */
    done = 1;
    forall(i=0;i<N;i++)   if(!D[i]){
        if(A[f1(i].key == i &&
          A[f2(i].key == i &&
          A[f3(i].key == i &&
          A[f4(i].key == i &&
          A[f5(i].key == i &&
          A[f6(i].key == i){
             D[i] = 1;
             "Same loop body as in Figure 3-1 with A[n] -> A[n].data"
    } else done = 0;
  }
}
```

Figure 3-2: Cedar synchronization key applied to Figure 3-1

In the best case scenario the three forall loops will be executed once and the whole operation takes constant time (assuming a sufficient number of processors and memory banks).

A typical execution is shown in Figure 3-3, assuming the loop iterations range from 1 .. 8 and the index of array A[] ranges from 0..9. For simplicity it is assumed that f1 = f2 = f3 = f4 = f5 so only two distinct input data dependent index sets (f1 and f6) are shown.

Three iterations of the outer loop are required in this example.

ITERATION i	INDEX DATA f1(i)	f6(i)	EXECUTION Pass1	Pass2	Pass3
1	0	6	X		
2	3	7	X		
3	3	4		X	
4	8	4			X
5	0	1		X	
6	9	5	X		
7	2	5		X	
8	2	1			X

INDEX n	REFERENCE PATTERN (n.f)	A[n].key Pass1	Pass2	Pass3
0	1.f1 6.f1	1	6	
1	5.f6 8.f6	5	6	8
2	7.f1 8.f1	7	7	8
3	2.f1 3.f1	2	3	
4	3.f6 4.f6	3	3	4
5	6.f6 7.f6	6	7	
6	1.f6	1		
7	2.f6	2	2	
8	4.f1	4	4	4
9	6.f1	6		

Figure 3-3: Execution of the program in Figure 3-2

However the worst case execution time can be $O(N^2)$, which is asymptotically worse than the sequential execution time O(N) even with unlimited number of processors and memory banks. If all references are to the same array location only one iteration will be executed each time the three forall loops above are executed. Moreover each such step will take (ignoring constant factors) O(N) time since during the first step all stores will be to the same location and get sequentialized by the memory system.

If the architecture model is modified to take into account the possibility of a recombining network [5], multiple references to the same location can get combined reducing execution time to $O(N\log N)$[1] . However, apart from the hardware cost of such a network, it precludes the use of low latency networks like the Hypercube or Crossbar as well as the use of dynamic routing algorithms which allow data to be routed around congested nodes. A software implementation of recombining networks as suggested in [7] will not work since the memory references that need recombining are not known at compile time.

Another potential problem with the above shared memory synchronization primitive is the excessive number of expensive shared memory accesses that can occur in implementing the indivisible synchronization operation. One solution suggested in [9] is to have a smart memory controller, which can recognize the synchronized memory requests and perform the indivisible comparison test and tag and data update without using up bandwidth between processor and shared memory. This still does not solve the problem of busy waiting. This case is supposed to be handled by the processor repeatedly requesting the synchronized access until it succeeds, which can increase the cost of synchronized memory access due to memory contention to

[1] Thanks to Prof. Allan Fisher for pointing this out.

several times the value of a normal access. An implementation which can handle this busy waiting with constant overhead is described later.

4. A new algorithm to resolve memory aliasing

This section describes a new synchronization primitive and compilation algorithm that allows run time resolution of aliased memory references. The technique differs from that described by Zhu and Yew (ZhuYew84) in that both the best case, worst case and intermediate case performance is within a factor of 2 of data flow execution with greedy scheduling, subject to the restriction that the maximum parallelism is limited by a constant which depends on architectural parameters including the number of bits in the tag.

Each data word in memory may be associated with a multibit synchronization tag. The synchronization primitive is defined by the following indivisible operation sequence:

1. Wait until tag bits specified by a mask are 0 at the memory location of interest.

2. Perform memory reference.

3. Modify tag bits by adding a specified signed number to the tag.

This is similar to the semantics of the synchronization primitive of Zhu and Yew [8], except that the tag test and tag modification are simpler and more restricted. During the period (if any) that the memory reference has to wait the process performing the memory reference may either block or receive an error code. In the following compilation algorithms it is assumed that the blocking mode of synchronization is being used. In the following examples synchronized memory references will be represented as follows : {mask : location, key increment}. Thus {0: A[3],4}, means that the read/write of location A[3] proceeds unconditionally, and is followed by incrementing the synchronization key of the same location by 4. It is assumed that all memory references are indivisible and atomic ie each processor waits until all previous references are acknowledged, before initiating the next request, and the changes to memory locations are effective simultaneously wrt all processors.

The first synchronization algorithm will be illustrated using the example used previously in Figure 3-1. As before the addresses of the memory references are assumed not to be modified after the loop is entered. The algorithm produces a modified program which may be described as follows (initial and final synchronization key values are always zero, no initialization is normally required). :

Synchronization algorithm 1

Repeat until all loop iterations are executed :

1. Perform a data flow analysis of all aliased memory references in the next k (usually 4-16) iterations. This is done in parallel by unconditionally incrementing the synchronization key of all aliased memory locations involved. The value of k and the increment is determined at compile time as explained later. In the case of conditionals and case statements all branches are executed.

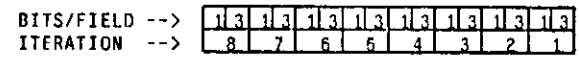
2. Execute the k iterations in parallel (ie schedule them ignoring the dependencies due to aliased memory references). Each aliased memory reference will be replaced by a synchronized reference whose mask value is chosen so that the reference blocks until all preceding reads & writes to the same location have executed, and decrements the synchronization key by the same value used as an increment during step 1 above. In the case of conditionals and case statements aliased memory references in the branches not taken must also be executed, but in such a way that only the synchronization key is modified.

The value of the increment/decrement used for a particular synchronized memory reference and the value of the mask used in step 2 above is determined (at compile time or at loop entry) as follows.

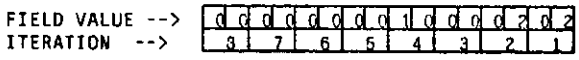
First, a worst case reference pattern (RP) for a single iteration is obtained which is simply the sequence of aliased reads and writes that would occur during sequential execution. In the case of conditionals or case statements the sequence is modified as if *all* branches had been executed in some (arbitrary) order. In Fig 3-1 this RP is $R_1R_2R_3R_4W_5W_6$ where the subscripts refer to the corresponding index set subscript (eg R_3 corresponds to A[f3(i)]). The combined RP of a number of consecutive iterations is obtained by concatenating the RP's of the individual iterations. Bits are now allocated to each write in the RP from left to write. Each write is allocated a bit field which can keep a count of the preceding consecutive reads and the write, which requires $[\log_2(\text{No-of-reads} + 2)]$ bits. In the above example there would be a 3 bit field and a 1 bit field allocated for each iteration. The number of bits in the synchronization key determines the number of iterations (the value of k in step 1 above) that can be parallelized. In this case with a 32 bit key $k = 32/4 = 8$. The allocation of bits in the synchronization key is illustrated in Figure 4-1 (a).

The increment in step 1 (= decrement in step 2) for any aliased memory reference simply corresponds to incrementing the corresponding bit field by 1. Thus R_1,R_2,R_3,R_4 and W_5 of the 3rd iteration would be allocated bits 8-10 (counting bits from 0), and the increment for each of these references would be 256 (2^8).

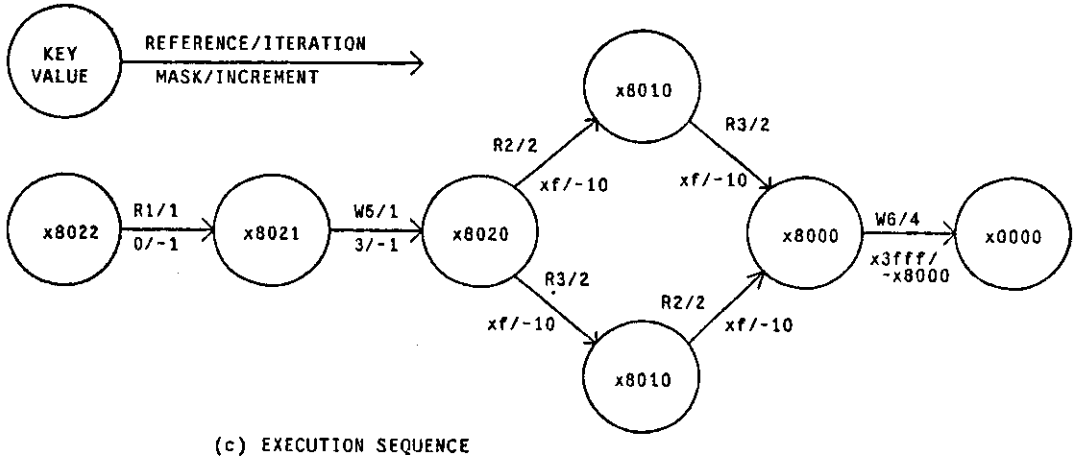
The synchronization key at an array location encodes the actual RP encountered at run time at that array location. If the RP of iterations 1 to 8 for a particular array location at run time was R_1 W_5 in iteration 1, R_2 R_3 in iteration 3, and W_6 in iteration 4, the synchronization key after step 1 would be as shown in Figure 4-1 (b).



(a) ALLOCATION OF SYNCHRONIZATION KEY BITS



(b) SYNCHRONIZATION KEY VALUE = x8022



(c) EXECUTION SEQUENCE

Figure 4-1: Sync. key for algorithm 1 and example in Figure 3-1

The mask value for reads in step 2, simply enables the bit fields of all previous writes. Thus the mask for reads R_1 to R_4 of the 3rd iterations would be bits 0-7 or 255.

The mask value for writes enables the same bits as reads plus all but the LSB of the write's own bit field. Thus the mask for W_5 of the 3rd iterations would be bits 0-9 or 1023.

The above rules for generating masks and increments ensure that in step 2 a read is blocked until all possible preceding writes (and consecutive reads preceding them) have actually executed, and that a write is blocked until all possible preceding reads and writes have executed. If one of these preceding reads/writes corresponds to a different address, the synchronization key in question will not be incremented during step 1, and so will appear to have already been executed in step 2. Each increment in step 1, will be matched by a corresponding decrement in step 2, so that the key values are restored to zero after all references have executed.

The sequence of mask and sync values assumed at the memory location in Figure 4-1 (b), during step 2 of the algorithm is shown in Figure 4-1 (c).

The result of applying the algorithm to the loop in Figure 3-1 is shown in Figure 4-2. The keyword *forall* again refers to a parallel *for* loop, with implicit synchronization on its termination. The values of mask and increments are represented compactly by macros mask(n) and incr(n) defined as :

- mask(n) = mask with n LSB's enabled.
- incr(n) = 2^n.

```
float A[N];
for(i=0;i<N;i+=8){
  forall(j=0;j<8 && i+j<N;j++){/* step 1 */
    {0:A[f1(i+j)],incr(j*4)};
    {0:A[f2(i+j)],incr(j*4)};
    {0:A[f3(i+j)],incr(j*4)};
    {0:A[f4(i+j)],incr(j*4)};
    {0:A[f5(i+j)],incr(j*4)};
    {0:A[f6(i+j)],incr(j*4+3)};
  }
  forall(j=0;j<8 && i+j<N;j++){/* step 2 */
    ... = ... {mask(j*4):A[f1(i+j)],-incr(j*4)};
    ... = ... {mask(j*4):A[f2(i+j)],-incr(j*4)};
    if(...{mask(j*4):A[f3(i+j)],-incr(j*4)}...){
      ... = ... {mask(j*4):A[f4(i+j)],-incr(j*4)};
      {mask(j*4+2):A[f5(i+j)],-incr(j*4)}
    } else {
          {mask(j*4):A[f4(i+j)],-incr(j*4)};
      {mask(j*4+2):A[f5(i+j)],-incr(j*4)} = ...;
    }
    {mask(j*4+3):A[f6(i+j)],-incr(j*4+3)} = ...;
  }
}
```

Figure 4-2: Algorithm 1 applied to example in Figure 3-1

Note that instructions within an iteration may themselves be scheduled in parallel, ignoring the dependencies between aliased memory references, since these are enforced by the synchronization mask in the same way within an iteration as between iterations. Thus each iteration may be scheduled on more than one processor, or on a pipelined processor with overlapped instruction execution.

The performance of this compilation algorithm will be compared to that of data flow execution with a greedy scheduling algorithm. The performance of the latter is obtained for any input data set, by doing a post-mortem analysis of the execution, and obtaining the actual dependency graph for the instructions. The instructions are then scheduled to be executed as soon as all previous dependent instructions have executed (the greedy scheduling algorithm). The execution time (eg on a data flow machine) of this schedule given infinite hardware will be optimal, except that when multiple reads to the same location are scheduled at the

same time, the underlying hardware will perform the reads in some arbitrary order which may not be the optimal order, ie the most time critical read may not be performed first. In such cases the performance can range from the optimal value to some sub-optimal value. The following lemma shows that the performance of the compilation algorithm described in this section, will lie somewhere within this range, if a constant factor of 2 is ignored, and subject to the restriction that the maximum parallelism is restricted by the number of tag bits which determines the maximum number of iterations k that can be executed in parallel.

Lemma 1: The execution time of each outer iteration produced by algorithm 1 is within a factor of 2 of the performance range of data flow execution with a greedy scheduling algorithm, if there are no conditionals or case statements in the loop. Memory bank collisions are ignored (ie assume infinite number of memory banks), and simultaneous references to the same memory locations are assumed to be sequentialized in some arbitrary order by the memory.

Proof: The execution of memory references in step 2 of the transformed program corresponds precisely to data flow execution : writes wait for preceding reads and writes to complete, and reads wait for preceding writes to complete. The fact that reads also wait for some preceding reads to complete does not effect the possible performance range, since memory references to the same location are sequentialized by the memory anyway. The execution time of step 2 therefore lies in the specified range. The execution of step 1 contains the same number of synchronized memory references as step 2, and can be scheduled in parallel, so that its execution time cannot exceed the upper limit of range for step 2. Hence the total time is within a factor of 2 of the specified performance range.

In the worst case all references are to the same location, so all the bit fields of that location's tag are set to the number of R/W operations in each R^*W sequence. In the first execution cycle all but the reads of the first R^*W sequence will block, while these reads will get sequentialized in some arbitrary order. After that the write of the same sequence will unblock followed by the reads of the next sequence and so on. Sequential execution will result. Since step 1 (setting up the keys) will also get sequentialized, the total time taken will be within twice the sequential execution time.

This algorithm works correctly even in the presence of conditionals and case statements, but now the execution time is within a factor of 2 of optimal only in the best case (when all data dependent memory references map to different locations). This is because all branches of the conditionals must be executed in step 2 and step 1, and unless the data dependent memory references in *false* branches all map to locations that are different from all other data dependent locations, their execution cannot be overlapped with that of the *true* branch actually taken and other iterations. If the test of the branch or case statement does not depend on an aliased memory reference (though the bodies of the branches contain aliased memory references), the algorithm can be modified, so that only the *true* branches are executed, both in step 1, and step 2. In this case the above lemma still holds.

5. Increasing the maximum parallelism

A major drawback of algorithm 1 is that only a small number of iterations (in the example above k = 8) can be executed in parallel. This puts an upper bound on the parallelism that can be exploited. To increase this upper bound the number of tag bits per data word and (of course) the number of processors must be increased proportionately, so that the hardware cost is $O(N^2)$ where N is the desired maximum parallelism. This is generally unacceptable for large N.

Algorithm 1 can be modified by overlapping the execution of step 1 and step 2. If step 2 takes more time than step 1 to execute (which is likely since the code is unlikely to consist entirely of aliased memory references) step 1 can execute several outer iterations ahead of the earliest outer iteration of step 2. In general if step 2 takes K times the time to execute as step 1, the upper bound on the parallelism increases from k iterations to (K+1)*k iterations. This is illustrated in the time diagram of Figure 5-1. Since all instruction in step 1 can be scheduled in parallel, K is simply the number of sequential machine instructions in one iteration.

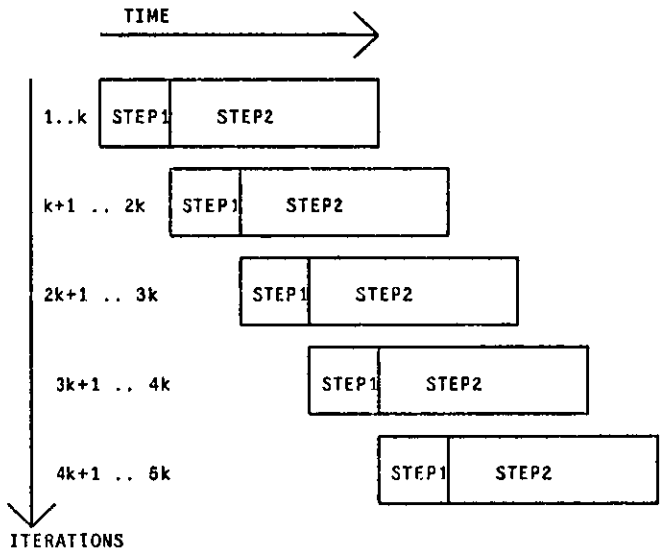


Figure 5-1: Time diagram of algorithm 2

To ensure that the synchronization keys from overlapped sets of iterations do not get mixed up, step 1 of any particular set of k iterations only begins executing after step 1 of the previous set has finished executing (as indicated in the figure). Furthermore during step 1, a synchronization key should only be modified if one of the following holds:

1. The synchronization key is zero. This guarantees that there is no outstanding memory references to this location from any previous set of iterations.

2. The synchronization key is not zero, but it is known that its current value was set by the current set of k iterations. This guarantees that some previous memory reference from this set of k

iterations has already verified (recursively) that there are no outstanding memory references to this location from any previous set of iterations.

For this to work the synchronization key must itself be tagged by the set of iterations which last modified it. Since the number of sets of iterations simultaneously active is usually a small number, only a few bits are required for this tag. Thus 6 bits would be sufficient to uniquely identify upto 64 sets of overlapped iterations.

The modified algorithm (algorithm 2) can thus be described as follows.

algorithm 2

Break down the iterations of the loop into successive sets of k iterations, where k is determined as in algorithm 1. Attempt to execute K of these sets in parallel (K is determined at compile time by the number of processors available : there must be enough to execute K*k iterations in parallel), by using a $[\log_2 K]$ bit tag on each synchronization key, and executing each set T in the range 1 .. N/k^2 as follows :

1. Wait for step 1 of set (T-1) as well as step 2 of set (T-K) to complete execution by using some additional explicit synchronization. Then execute step 1 of set T as in the case of algorithm 1, but add a test to each synchronized memory reference which tests that the synchronization key is either zero, or that the synchronization key's tag is equal to T. Also set the key's tag to T after each reference.

2. Proceed with step 2 as in algorithm 1.

In general the size of the synchronization key will grow as O(logN) where N is the maximum parallelism, resulting in a total hardware cost of O(NlogN). This bound holds in the range $k < N <= k*(K+1)$. (For larger values of N hardware cost again grows as $O(N^2)$). The main drawback of this technique is that successive sets of iterations can only be overlapped if none of their aliased memory references are dependent. This is not as bad as it seems since whenever two sets of iterations have memory references that are dependent the corresponding references must be executed sequentially anyway. However the above lemma about performance optimality may no longer be valid in general. Note that it is still valid for both the best case and the worst case (and for a number of intermediate cases).

6. Memory aliasing with multiple indirection

So far it has been assumed that the addresses of all aliased memory references were known when the loop was entered. The algorithm can be generalized to handle less restrictive cases in which the address of a memory reference is modified during the execution of the loop. It is possible to define the indirection level of an aliased memory reference as (one more than) the longest sequence of indirection which may be involved

---

[2] This range need not be known at compile time

from loop entry to determine the final address of the aliased memory reference. Thus the following loop has a level of indirection of 2 assuming that the arrays F1[] and F2[] are already defined on loop entry:

```
int A[N],B[N],F1[N],F2[N];
...
for(i=0;i<N;i++){
  B[F1[i]] = ...;    /* first indirection */
  A[B[F2[i]] = ...;  /* second indirection if F2[i] = F1[j] for any j<=i */
}
```

The maximum level of indirection need not be defined at compile time as illustrated by the following loop:

```
int A[N],F[N];
...
for(i=0;i<N;i++){
    A[A[F[i]] = ...;
}
```

In this case the maximum level of indirection is N which may not be known at compile time. The following extension to algorithm 1 (which can also be applied to algorithm 2) requires that the maximum level of indirection (L) be known at compile time:

algorithm 3

The algorithm generates the two basic steps of algorithm 1, but repeated for each level of indirection, for a total of 2*L steps. In the first pair of steps only aliased memory references whose level of indirection is 1 are executed, and in the second pair of steps only aliased memory references whose level of indirection is 2 are executed, and so on. By the time aliased memory references of any particular level of indirection are tackled, their addresses have already been determined in the previous step. Instructions that do not involve memory aliasing, are executed in the earliest possible instance of step 2, as determined by their dependence on other instructions that do involve aliased memory references.

Note that as before each step generated is applied in parallel to a set of iterations determined by the synchronization key size. Thus the best case performance is O(L) which is optimal. The worst case performance is within a factor of 2 of the sequential execution time since each aliased memory reference is replaced by two synchronized references. It is not known at this time if the performance is also optimal for intermediate cases as in the case of algorithm 1.

7. Hardware implementation

It is possible to implement the above synchronization primitive with negligible run time overhead, by having the memory controller perform the several operations that make up the indivisible synchronization primitive. The latency of synchronized memory references will be larger since the memory controller has to make two access to memory, one for the date and one for the key but the increase is no more than a factor of 2. The load on the processor memory connection is slightly higher for synchronized references than for normal references since the synchronization mask and increment values must be provided in addition to the address. However because of the simple format of the mask and increment values, they can both be encoded as the bit offsets involved, which for 32 bit keys takes $5+5 = 10$ bits. Even with an additional 6 bit tag (required in algorithm 2) 16 bits should suffice.

The real problem is implementing the semantics of the blocked synchronized memory references efficiently. Blocking of synchronized memory references can be implemented by using spin locking. This is the approach assumed in the case of the Cedar synchronization key [8]. However spin locking can be expensive in terms of processor to memory bandwidth consumed, slowing down the response of other memory references, and can invalidate the assumption of constant cost of memory references. For the kinds of synchronization algorithms described in [9], as well as the new synchronization algorithms described above, close to worst case performance can degrade dramatically, since in that case large number of synchronized references are blocked simultaneously, and get unblocked sequentially.

Implementations of blocking (also known as busy waiting) without spin locking have been suggested in [2] as part of a cache coherence scheme. When a processor tries to access a locked memory location its snooping cache is employed to track all memory traffic until the location is unlocked by some other processor. This technique however does not work without a broadcast bus interconnection between processors and memory and therefore cannot be applied to a scalable multiprocessor architecture.

The following scheme does not require a broadcast bus, and is based on the memory controller keeping track of blocked processors and memory traffic that might unblock processors. This scheme will work if the blocking condition is always a function of memory locations controlled by the same memory controller. This is the case for the synchronization primitives introduced here, since the blocking condition is only a function of the synchronization key (plus tag), which can easily be arranged to always be in the same memory block as the corresponding data item. Each memory controller has a small associative table (Figure 7-1). The associative table has one entry for each possible blocked memory reference. The total table size should equal the number of processors. Whenever the controller determines that a synchronized memory reference should cause a processor to block it indefinitely withholds the acknowledge from that processor (which is effectively blocked since is cannot distinguish this case from network congestion or other network delay) and stores the

address of the location accessed as well as the synchronization condition in the associative table. Every time a successful synchronized memory reference causes a synchronization key to be modified the memory controller does an associative lookup using the reference's address and new key, to see if any blocked processor's synchronization condition is true, and if so executes the delayed memory reference and finally acknowledges it. The simple form of the test on the synchronization key in the new synchronization primitive (test if a particular field is 0, or a small tag field matches a given value for algorithm 2) makes the associative table easy to implement using custom VLSI.

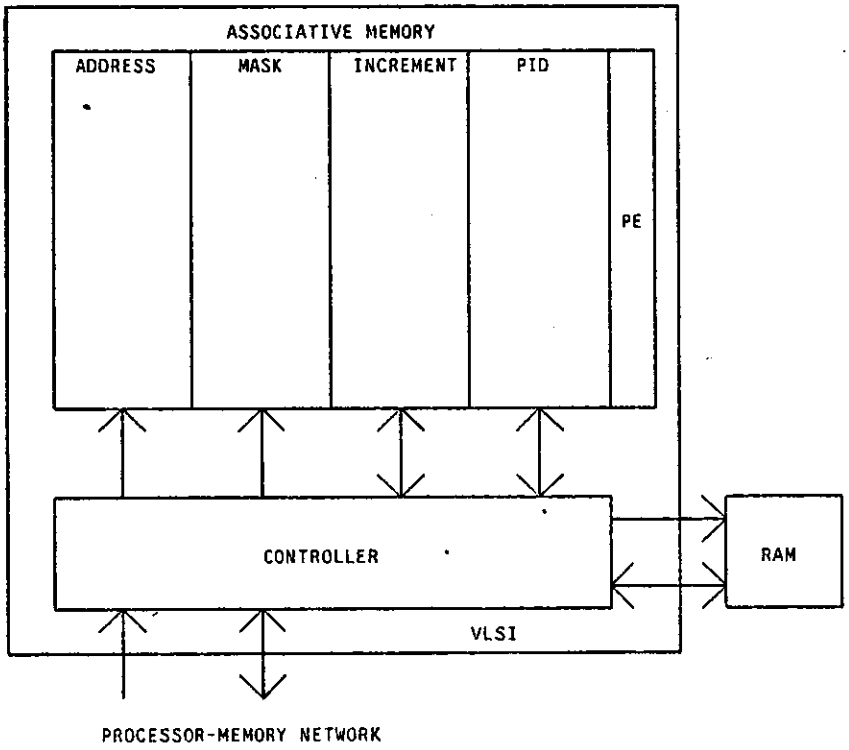


Figure 7-1: A memory controller handling busy wait

One drawback of this scheme is that is does not scale well in the asymptotic sense : The total hardware cost per memory module is proportional to the number of processors. Since the number of memory modules should also be proportional to the number of processors, the total cost grows as the square of the number of processors. However in practice the size of the associative memory should not dominate the cost of the memory module, unless the number of processors starts exceeding about 1k, so this is a realistic concern only if there are actually programs that have parallelism in excess of 1k and this parallelism all depends on aliased memory references.

8. Conclusion

This paper has described a number of compilation algorithms which are able to parallelize loops that contain aliased memory references, in such a way that performance is within a factor of 2 of optimum both for best case and worst case input data. The algorithms can handle aliased memory references in conditional statements though if the conditional test itself depends on an aliased memory reference, the worst case performance deteriorates somewhat. Multiple levels of indirection in aliased memory references can be handled provided the number of levels of indirections is a compile time constant. The synchronization primitive required was shown to have a hardware implementation that satisfied the timing assumptions made in the performance analysis.

This paper has not addressed the issue of instruction scheduling which becomes an important problem when the parallelism in the architecture is less than the potential parallelism encountered in the program. The problem is similar to the scheduling problems encountered in data flow machines under similar conditions. Since many actual processor memory networks are pipelined, it is also necessary to consider how the synchronization algorithms can be modified to accomodate multiple outstanding memory references from the same processor.

References

1. U. Banerjee. Data dependence in Ordinary Programs. Report 76-837, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, November, 1976.

2. Philip Bitar and Alvin M. Despain. Multiprocessor Cache Synchronization. The 13th annual international Symposium on Computer Architecture, June, 1986, pp. 424-433.

3. Joseph A.Fisher,John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction, 1984, pp. 37-46.

4. David J. Kuck. "A Survey of Parallel Machine Organization and Programming". *ACM Computing Surveys 9*, 1 (March 1977), 29-59.

5. Gyungho Lee, Clyde P. Kruskal and David J. Kuck. The Effectiveness of Combining in Shared Memory Parallel Computers in the presence of 'Hot Spots'. Int'l Conf. on Parallel Processing, 1986, pp. 35-50.

6. David A. Padua, David J. Kuck and Duncan H. Lawrie. "High-Speed Multiprocessors and Compilation Techniques". *IEEE Transactions on Computers C-29*, 9 (September 1980), 763-776.

7. Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. Int'l Conf. on Parallel Processing, 1986, pp. 51-58.

8. Chuan-Qi Zhu and Pen-Chung Yew. A-Synchronization Scheme And Its Applications For Large Multiprocessor Systems. Proc. of the 1984 Int'l Conf. on Distributed Computing Systems, May, 1984, pp. 486-493.

9. Chuan-Qi Zhu and Pen-Chung Yew. A Scheme to enforce data dependence on large multiprocessor systems. CSRD report No. 24 24, University of Illinois at Urbana-Champaign, May, 1986.