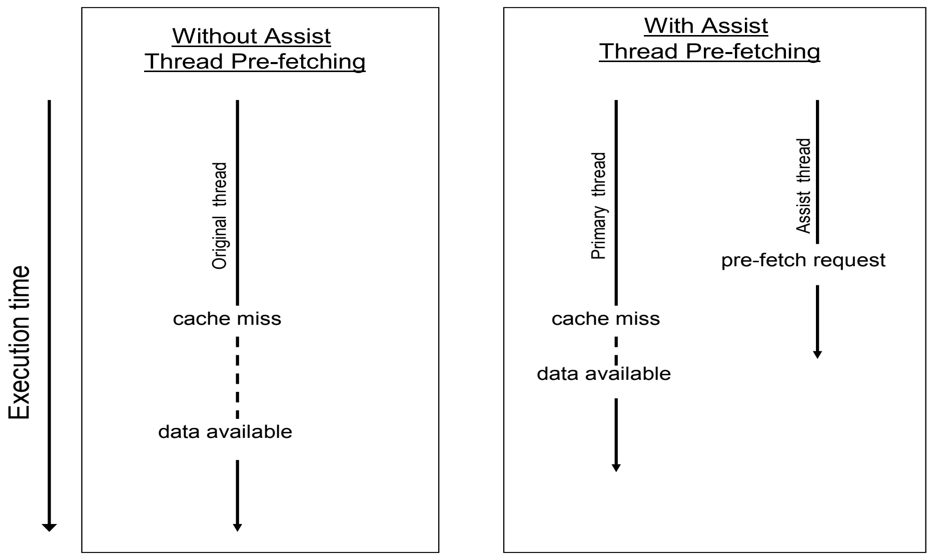
Data Pre-fetching Using Assist Threads

This work aims to develop compiler techniques for automatic data pre-fetching using assist threads. In the PERCS architecture, hardware can support multiple threads of execution on the same chip, either as SMT threads that share resources on a single core, or as CMP threads that execute on different cores on the same chip. When an application does not exhibit enough parallelism to effectively use all available threads, the extra threads can be used as assist threads to improve the performance of the primary thread(s). In this model, the primary threads perform all useful work in the application, while the assist threads work only to aid the execution of the primary threads, i.e. the work done in assist threads is not necessary for correct execution of the application. This model allows us to throttle the execution of assist threads when it is detrimental to primary thread execution.

This work targets using assist threads to pre-fetch into the cache data required by computation in the primary thread. When a cache miss occurs in the primary thread, it may cause execution of that thread to stall waiting for the memory subsystem to fetch the corresponding cache line. Performance loss due to memory stalls can be significant, and with increasing disparity between processor and memory speeds, it is expected to become even more significant. This performance loss can be mitigated if the use of a data item is anticipated ahead of time, and a dcbt instruction is issued to pre-fetch it into the cache if it is not already present there. This pre-fetching is done in the assist threads, which share the cache hierarchy with the primary thread. The compiler transforms the source code to automatically generate code for the assist threads, and synchronize their execution with respect to the primary thread. The following figure illustrates the effect of using assist threads for pre-fetching.



1. Basic Framework

To generate assist thread code, the compiler first needs to decide what memory accesses to pre-fetch into cache. The memory accesses targeted for pre-fetching are the delinquent loads, i.e. load instructions that cause the most cache misses during execution. It has been previously described how delinquent loads are determined, and we use the same results for pre-fetching using assist threads. The compiler then uses a back-slicing algorithm to determine the code sequence that will execute in the assist thread and will compute the memory addresses corresponding to the delinquent loads that are to be pre-fetched. The back-slicing algorithm operates on a region of code containing the delinquent load, and this region may comprise of the whole procedure, or the containing loop nest, or some level of inner loops within a nest. Command-line options exist to allow the user to specify the size of the region considered for slicing. Compiler heuristics are also being developed to automatically determine the region to use such that the resulting slice is optimal. This slice should pre-fetch a large number of data items ahead of their use in the primary thread, but it should do this without negatively impacting the cache usage of the primary thread by bringing in too much data ahead-of-time. Compiler heuristics to control the back-slicing region (and hence the size of the slice generated) are based on the ratio of computation needed for delinquent address calculation and the computation in the whole region. In some cases, the delinquent load may not be contained in any loop within its own procedure (say *foo*), though it is contained in a loop within another procedure (say *bar*) that invokes procedure *foo*. In this case, the compiler attempts to inline *foo* into *bar* before applying back-slicing, so as to enable better control over the granularity of the assist thread code generated. The basic algorithm for back-slicing makes use of the procedure control flow graph as well as the SSA graph, and it proceeds as follows:

1. Initialize the set of instructions in the slice (say SliceSet) to be dcbt instructions whose operands are the address expressions in the delinquent load instructions.
2. Initialize the set of upward exposed reads (say UpwardReads) to be empty.
3. Repeat until there is no change to SliceSet:

   For each instruction in SliceSet:

   a) If the instruction is in a control-dependent basic block, then:

      For each predecessor basic block:

      i. Find the conditional branch instruction that leads into this instruction's basic block.
      ii. Add this conditional instruction to set SliceSet if it has not already been added, and if it is within the back-slicing region.

   b) For each read reference (memory or register) in the instruction:

      i. Determine the SSA definition (say *def*) for the reference.
      ii. If *def* has been processed earlier, then continue to the next reference.

iii. If *def* occurs outside the back-slicing region, add it to the set UpwardReads if it defines a procedure-local variable, and continue to the next reference.

iv. If *def* is not an SSA Phi node, then add it to the set SliceSet, and continue to the next reference.

v. *def* is a Phi node; add both its condition expression, as well as the definition expressions to SliceSet.

4. Remove from SliceSet all instructions that store to a variable that is not local to the procedure.

The sets SliceSet and UpwardReads determined by the back-slicing algorithm are used to generate a separate slice function that will execute as part of assist thread code. First, the original procedure is cloned to create a copy with its own local variables. Then, only instructions in SliceSet are retained in the cloned copy, and other instructions are discarded. Next, for each instruction in UpwardReads, two new assignment statements are generated: one in the original procedure, and one in the cloned copy. The assignment in the original procedure is placed directly before the entry to the back-slicing region, and it assigns the value of the upward exposed read variable to a shared memory location. The corresponding assignment statement in the cloned procedure is placed at the beginning of that procedure, and it assigns the value from the shared memory location to the procedure-local copy of the upward exposed read variable. The original procedure is executed in the primary thread, whereas the transformed cloned procedure is executed in the assist thread.

The original procedure is also responsible for signaling to the assist thread that it should start pre-fetching. The code for the original procedure is transformed so that after it has copied the values of the upward exposed read variables, it copies to a shared memory location (say *func_addr*) the address of the slice function for the succeeding back-slicing region. It also copies to another shared memory location (say *data_ptr*) the address of the memory area containing the values of the upward exposed reads for that slice function. Finally, it signals to the assist thread to start pre-fetching using the procedure given in *func_addr*, while the original procedure continues executing the original application code. The following pseudo-code illustrates the code transformation for slice function generation:

Original Code

```
// A and x are globals

y = func1();
…
i = func2();
// start of back-slice
while (i < condition) {
            …
            x = func3();
            …
            // delinquent load
            func4( A[i] );
            …
            i += y;
}
// end of back-slice
```

Primary Thread Code

```
y = func1();
…
i = func2();

// y and i are upward-exposed reads
char *tmp = malloc(sizeof(y)+sizeof(i));
*((int *)tmp) = y;
*((int *)(tmp+sizeof(y))) = i;

// start pre-fetching in assist thread
data_ptr = tmp;
func_addr = &slice_func_1;
signal assist thread;

// start of back-slice
while (i < condition) {
            …
            x = func3();
            …
            func4( A[i] );
            …
            i += y;
}
// end of back-slice
```

Assist Thread Slice Function

```
void slice_func_1(char *dptr) {

   int y = *((int *)dptr);
   int i = *((int *)(dptr+sizeof(y)));
   while (i < condition) {
            // pre-fetch request
            __dcbt( &A[i] );
            i += y;
   }
}
```

The compiler inserts code in the program entry point procedure (i.e. in the main procedure) to create and run the assist thread. This thread is bound to a specific processor, which is either the same processor as the one executing the primary thread in case of SMT, or a processor on the same chip in case of CMP. When it starts up, the assist thread executes a continuous loop, in each iteration waiting for the primary thread to signal start of pre-fetching, and then invoking the corresponding slice function.

```
while (1) {
        wait for primary thread to signal start of pre-fetching;
        // Global shared variable func_addr gives assist thread slice function to invoke.
        // Global shared variable data_ptr gives address where values of upward exposed
        //        read variables have been stored by the primary thread.
        (*func_addr)(data_ptr);
}
```

2. Optimizing Code Generated for the Slice Function

The assist thread execution should not affect the results of primary thread execution in any way. For this reason, back-slicing excludes all stores to shared variables when generating assist thread code. Due to this, the semantics of the code executed in an assist thread may deviate from the original application, and this may result in the execution of unexpected code paths or the use of unexpected data, which may trigger exceptions that would never have been triggered in the application. Thus, the assist thread has to be set up such that all exceptions generated by assist thread code are locally contained, and do not propagate any effects to the primary thread. Depending on the type of exception and the context in which it occurs, the exception may be ignored in the assist thread, or it may cause the assist thread to skip the execution of certain code regions.

One way to increase the efficiency of the assist thread code is to have it use the same data values for shared variables and follow the same control flow paths as the primary thread code, thus computing the same addresses for delinquent loads that the primary thread will use. In some cases, this can be achieved by creating a new copy of the shared variable that is private to the assist thread, and initializing the value of this copy before entry into the back-slicing region. Such privatization may help when the value of the shared variable is modified within the back-slicing region before being used, either for delinquent address computation or for conditional control flow. Privatization may be worthwhile only for small-sized data items (scalars and small arrays or aggregates), since it incurs extra memory overhead as well as the overhead of initialization.

Another approach to increasing assist thread efficiency is to reduce the slice function code size. When the slice function code size is reduced, the assist thread takes up fewer hardware resources, and has opportunities to run-ahead and issue more pre-fetch requests ahead of their use in the primary thread. This is especially important in the case of SMT threads where the primary thread and assist thread share resources on the same core. The code size can be reduced by selectively eliding some branching control flow code contained in the slice function. This is done when the code for computing both branches of the control flow is simpler than the code for computing the branch conditional. Doing this is safe because assist thread exceptions have to be specially handled, and assist thread execution is designed not to impact the results of the original program, even when it executes code paths that would not execute in the original code. The code transformation to remove some branching control flow may result in the assist thread code issuing pre-fetch requests for data that is never used in the primary thread. Depending on data placement and cache usage characteristics of the application, the unnecessary pre-fetch requests may or may not have a detrimental impact on performance. Compiler heuristics have to be used to apply this transformation only when it is likely to be useful, and avoid it in cases when the cache utilization is expected to be high, or when there are null pointer checks in the address computation, or when the size of the slice function is already small relative to the back-slicing region.

Note that the code transformation for privatization is at odds with the transformation for branching control flow elision, since it introduces stores to privatized variables and increases slice function size. Also, control flow elision may result in code paths different from the original code, which goes directly against the purpose of privatization. The compiler needs to carefully apply each of these transformations based on analysis of code in the back-slicing region.

3. Synchronizing Assist Threads

If the assist thread runs too far ahead of the primary thread, it may pre-fetch data into cache long before it is used in the primary thread. This will put pressure on the cache utilization, and may cause other useful data to be evicted from cache, or may even result in the pre-fetched data being evicted from cache before it is actually used in the primary thread. On the other hand, if the assist thread lags far behind the primary thread, the pre-fetch requests that it issues will occur too late to be of any use, and may unnecessarily bring data into cache if this data had earlier been used and then evicted due to contention. Thus, it is important that pre-fetch requests occur in a timely manner, and this requires the assist thread execution to be synchronized with the primary thread execution. Without proper synchronization, the assist threads may not provide any benefit, and may even lead to performance degradation in the worst case.

3.1 Throttling Assist Thread Execution

An assist thread can be throttled using a dynamic runtime check to keep it within a certain threshold distance of the primary thread. This check is easy to perform when the back-slicing region comprises of a loop level. In such cases, the code is transformed to include an explicit loop iteration counter variable in both the original function code and the assist thread slice function. These loop iteration counters are shared variables that are initialized outside the loops, and incremented at the end of each iteration. Also, code is inserted at the beginning of the loop body in the slice function, to check if the loop counter in the slice function is greater than the primary thread loop counter by more than a threshold number. If it is, then the transformed code causes the assist thread to wait until this condition is no longer true, and the primary thread execution is within the threshold number of iterations of the point of execution that the assist thread is waiting at. The threshold number is a tunable parameter that can be specified on the command-line by the user when compiling the application. The compiler can also estimate a threshold value by evaluating the difference between the code comprising the original back-slicing region and the code generated for the slice function, and by inspecting the cache usage characteristics of the back-slicing region. Based on its analysis, the compiler may decide not to perform any code transformations for assist thread throttling.

3.2 Accelerating Assist Thread Execution

When there is little difference between the back-slicing region code and the slice function code, the assist thread may not run sufficiently ahead of the primary thread, or it may even lag behind the primary thread since it has lower priority at runtime. In such cases, assist thread execution has to be accelerated to be of any use. One technique to do this is to increase the size of the back-slicing region. This allows greater scope for decreasing the amount of code executed in the assist thread relative to the code executed in the primary thread. As a result, the assist thread pre-fetching can start at an earlier point in the primary thread's execution, thus giving the assist thread more of a head-start.

Another technique to accelerate assist thread execution is to use multiple assist threads for one primary thread. Instead of starting a single assist thread in the main procedure, the code is transformed to start multiple threads. This technique can be easily applied when the back-slicing region is a loop, and the pre-fetch address computation is a function of the loop iteration count. The slice function code is generated with an extra parameter that gives the number of assist threads in use. This parameter is used to determine the loop iteration counter increment for the loop within the slice function. Also, each assist thread knows the offset at which to start its loop iterations based on an identifier made available to the thread when it is started. Thus, loop iterations are distributed evenly among the multiple assist threads in use. For example, if three assist threads are in use, then for any three consecutive iterations of a back-slicing region loop, each of the three iterations will execute in a different assist thread.

Yet another technique to accelerate assist thread execution is to have the assist thread simply skip iterations periodically. Once again, this technique is easily applied when the back-slicing region is a loop, and the pre-fetch address computation is a function of the loop iteration count. The code for the slice function can be modified to include a runtime check similar to the one used for throttling assist thread execution. However, in this case, when the loop iteration count of the slice function is found to be less than or within a threshold distance of the loop iteration count in the primary thread, the slice function loop iteration variable is simply incremented by a certain number so as to have the assist thread skip several iterations ahead of the primary thread.

4. Using SMT Assist Thread or CMP Assist Thread

The assist thread may be started as an SMT thread that executes on the same core as the primary thread and shares hardware resources with it, or it may be started as a CMP thread that executes on a separate processor core. Using SMT threads has the advantage that it has less overhead in terms of hardware resources used for the application execution. Also, an SMT assist thread shares the complete cache hierarchy with the primary thread, and can pre-fetch data into the closest cache level (L1 cache). However, since an SMT assist thread and the primary thread execute on the same processor core, there is potential for resource contention between the two threads. It is important to run the assist thread at a lower priority than the primary thread, so as to minimize performance degradation due to such resource contention. Also, if an SMT assist thread pre-fetches into a smaller L1 cache while a CMP assist thread pre-fetches into a larger L2 cache, then in the case of heavy cache utilization the SMT thread is more likely to pollute the cache contents with pre-fetches that cause eviction of currently useful data.

When an SMT assist thread is not performing useful work (either it is waiting for a signal from the primary thread telling it when to start executing a slice function, or it is being throttled to prevent it from running far ahead of the primary thread), it should be temporarily removed from the ready queue so as to avoid using up processor issue slots just for doing a busy-wait. Fine control over thread execution states is important for performance, particularly in the case of SMT threads. For this reason, it is preferable to use a kernel thread interface for assist threads that provides a greater level of user control, rather than going through the abstraction of a software library such as pthreads.