Summary:

The paper proposes an approach called *Decoupled Software Pipeling* (DSP), which tries to optimize the execution of *recursive data structure* (RDS) traversal loops.
During the traversal of a recursive data structure, in each iteration of the loop the next traversed element must be loaded from a pointer in the current element. Due to the recursive nature of the data structure, the next element is most likely not stored in close location to the current one, since these data structures are primarily used whenever there is a need to dynamically add new or remove existing elements. Therefore during traversal, there is a high possibility that a memory access would miss the cache and hence cause an increased delay.
Modern compilers and processors try to alleviate such issues caused by high latency instructions (such as memory access) by allowing some kind of *instruction-level parallelisim* (ILP). This allows to hide the latency by scheduling independent instructions while waiting for the high latency instruction to complete.
However, the latency is for the compiler statically hard to determine and often there aren't enough fully independent instructions which could be scheduled. Furthermore even the instruction reordering done by the processor isn't in most cases not sufficient, since the processor has no full knowledge of the program code and can accordingly only perform local decisions.
This condition also arises during the traversal of a RDS, since the determination of the next traversed element is most likely a memory load, which could easily miss the cache, after the computation on the current element has been completed. Therefore it could be better to enumerate the next element, before the actual computation would be done. This optimization, however, is most likely not applied by the processor or the compiler with existing techniques.
At this point the DSP approach gains its momentum. The approach proposes the split of the former sequential, single-threaded execution into separate *consumer* and *producer* threads. Here the producer performs the enumeration of all elements, which will be traversed by the loop. The consumer now "consumes" the elements, i.e. it performs the computation on each element as it would be done by the sequential loop.
This has the advantage, that now the cache "pollution" and any latency caused by the consumer won't affect the producer and vis versa. Both threads are *decoupled* from each other.
Now the question remains where the producer writes and the consumer retrieves the elements from? From Memory? In this case, the approach would encounter the same issues again. By using a limited queue e.g. as array, locality could be improved, but still memory with its latency would remain.
The approach therefore introduces an hardware extension called *Synchronization Array*, which provides an non-memory possibility for processor cores to directly share elements through a queue-like interface.

Issues / Open Questions:
I think the linked-list traversal in figure 1 is a bad suited example for the latency which would be involved on the critical path by any instruction in the body of the loop, since there is still locality.
As far I recall, the processor can only obtain cache-line sized chunks from the memory, which would then stored in the cache. Since the cache-line size in the experiments is 64 bytes and apparently a linked list entry here contains the `next` pointer and a int-sized `value`, chances are high that upon access to the `value` the whole entry would be stored in the cache.
Therefore the access to the `next` pointer would be very fast. Hence it is legal to first access the current element, before switching to the next one, because the latency seems to be identical.

Decoupled Software Pipelining with the Synchronization Array

1 Summary

This paper discusses a technique used to parallelize recursive data structure traversals which are considered difficult to do so due to pointer chasing involved. Two new ideas, decoupled software pipelining(DSWP) and synchronization array are used in order to achieve that. DSWP divides the sequential loops into parallel fragments, while synchronization array helps in lightweight communication between the threads.

Authors discuss about identifying critical path in the program for good cache performance. Critical path in a RDS traversal algorithm is calculation of next pointer which is used for further computations inside the loop. This calculation of the next pointer and further computations can be done independently of each other. So, problem is transformed into an instance of producer-consumer where producer only calculates the next pointers and consumer uses these pointers for computations. Using operating system provided primitives for synchronization between producer and consumer is costly, and decreases the performance gain obtained by parallelization.

A structure called synchronization array(SA) is used for inter-thread communication. This is a virtual queue, and is accessed by producer and consumer threads. Each entry in the SA contains metadata in addition to actual data to identify the corresponding producer and consumer. This metadata is also used to check if queue is empty or full to block consumer or producer, if necessary. SA rename logic takes care of reclaiming the slots in the array after consumption. SA will block the producer when the queue is full and wakes up the producer when an item is consumed. Similarly, when queue is empty consumer is blocked until an item is produced. SA will be part of process control block of the process.

Method to identify RDS traversal loops to carry out DSWP is discussed next. This is done by identifying induction pointer loads(IPL) in the program. Initialization of pointers in the IPL forms the producer function. Rest of the computation that uses this pointer will form the consumer.

2 Questions

1. How does synchronization array rename logic works exactly?

Decoupled Software Pipelining with the Synchronization Array

Summary

The paper presents a technique to parallelize so-called RDS loops, i.e. loops traversing recursive data structures. Since those loops are told to suffer significantly from cache miss delays during iteration, which cannot be handled by current out-of-order execution approaches satisfyingly, the paper suggests parallelization of the induction parts of the loops that cause the high latencies with those who would suffer from that. The mean to deal with that issue is *decoupled software pipelining*, which is basically slicing the loop into an induction part that is the code fragment determining the next object/pointer on which the loop is executed, and a computational part that computes the side effect of the iteration on the current item. Because they assume that a task synchronization between the threads executing the two slices of the loop via shared memory is too expensive, the use of a dedicated non-memory-based address space, the so-called synchronization array (SA), is propagated.

The synchronization array is basically a fast synchronized buffer for task communication between threads. The thread producing a task calls the produce() function to push the task (represented for exampple by a single pointer to a RDS node) to the SA, while the consumer thread pulls those work items via consume() and executes them. Additionally, the entries in the SA are augmented with synchronization flags and tags identifying the thread belonging and execution order information of individual tasks. Thus, we get an efficient (because on hardware level implemented) buffer for communicating jobs between master and slave threads, thereby also achieving the chronological decoupling between the threads.

The parallelization of the loop itself is done during compilation. A dependence analysis is performed to identify the induction code of the loop. That code is sliced out of the loop and augmented by the produce() function to be executed on the first thread, thereby isolating the *critical path* of the loop. The rest of the loop code (which does not contain any loop-carried dependencies) plus an initial consume() function call per iteration forms the code executed by a second slave thread independently. The result system are two threads, one computing all nodes of a RDS to process and one manipulating those nodes in parallel to that.

The paper further states that the principle of *decoupled software pipelining* can be applied to other applications as well as extended to multiple threads, but that those applications (obviously) depend on the power of software analysis applied at compilation time.

Open Questions

- The paper states that common dynamic techniques of out-of-order synchronization are not able to hide the delays of critical path instruction. The described approach deals with those latencies by parallelization. Wouldn't it be more desirable to fill the latency gaps instead of distributing them over several cores by improving present techniques?

Summary Decoupled Software Pipelining

The paper introduces the concept of decopuled software pipelining; compared to the other papers we read, speed ups are achieved at a much finer granularity. Namely, it works on the instruction level.

The authors argue that on current hardware archtitectures, load instructions have a latency so high that even Out–of–Order systems are stalled by it. They claim that this is especially prevalent when working on recursive data structures, that is, data structures which point to further instances of it.They argue that this is caused by processors not prefetching the next load instruction, but the instructions working on the data. However, if the load instruction would be prefetched, its latency could be hidden by executing the "work" instructions in the meantime.

Their proposed solution to this is decoupled software pipelining, which separates the traversal from the computation. It transforms the program to give it a producer / consumer stucture. With it, the loading can progress faster, and by buffering the loaded data, a speedup can be achieved.

The authors claim that this transformation can be done by a compiler, arguing that the traversal code could be identified by searching for load instructions which depend on previous values loaded by the same instruction. Then, progarm slicing seems to be used to separate the traversal from the remaining instructions.

One issue with DSWP is that the producer/consumer pattern normally has a high synchronization overhead. To overcome this issue, the authors propose a hardware extension, the synchronization array. The synchronization array is able to buffer data of the consumer and to return it, once the cosumer demands it. If the consumer however has not produced the data prior to a consumer's request, it will add a tag. This tag allows the system to provide the data to the correct instruction later when it is available. As there is no real hardware with a synchronization array, the authors run some simulations. With those they show that they can achieve speedups on both out-of-order as well as on in-order cores; they additionally use further simulations to show that their speedups are not solely induced by prefetching.

Open Questions

- Can decoupled software pipelining be applied to non-linear recursive data structure? In trees, one has normally multiple pointers which could be followed, and the chose one depends on the value of the data.
- How exactly is the out-of-order support for the synchronization array implemented?
- How does the compiler ensure the "one-to-one correspondence between produce and consume instructions"?

Decoupled Software Pipelining with the Synchronization Array

1 Summary

The paper proposes a technique to increase the performance of recursive data structures such as lists trees etc. by parallelizing the traversal loops of said structures. They call their technique decoupled software pipelining (DSWP) and implement it as a compiler optimization. DSWP will search for loops that employ loads of recursive structures and try to slice the loops in two parts: The traversal code which controls the access of all nodes traversed and the computation code that performs an operation on each node. Since these code fragments are mostly only dependent in one direction they can be executed in parallel using a buffer queue for synchronization. For obvious reasons this synchronization will introduce an immense overhead. To reduce this overhead the paper proposes to replace the synch-queue by a hardware based synchronization scheme for which they describe an abstraction they call the synchronization array. This synchronization array will mimic the access scheme of a queue by providing synchronized access and bookkeeping for certain registers. The two threads that compute a RDS traversal loop will access the SA by using a unique number to identify a pair of read write accesses. Since this technique for synchronization is not based on memory but on registers it will not interfere with cache locality and its overhead will be very small.

2 Questions

- How does the synchronization array behave when register space is nearing exhaustion? Is the buffer size dynamic? Can it be controlled by software?