Coping with Memory Latency

Restructuring General-Purpose Programs to cope with Memory Latency

Interim Project Report

Dirk Coldewey

UCSC-CRL-97-06
March 20, 1997

Board of Studies in Computer and Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

The widening gap between microprocessor speeds and DRAM has spawned a number of approaches for tolerating the resulting latency of memory accesses. Four common approaches are software controlled prefetching, multi-threading, non-blocking loads, and relaxed consistency models. Most of these methods have been evaluated only on array based codes, although prefetching has also shown to be effective for instruction cache prefetching in operating system codes that have been hand-optimized for cache performance. Many of these methods suffer from considerable runtime overhead, scalability issues, or only work well for a small problem domain. This thesis proposes to show that a method of code transformation to support a combination of prefetching and coarse-grain multithreading can significantly reduce the effects of memory latency with less runtime overhead than currently proposed forms of prefetching, yet that performs well for both array-based codes as well as a broad class of pointer-based data structures found in commercial applications.

1. Problem Statement

While the performance of microprocessors has doubled roughly every three years, DRAM performance has not kept pace, growing at a rate of roughly 22% every three years [HP90]. Consequently the performance of memory is becoming a limiting factor in system performance. Faster processor cycle times can restrict the size of the primary cache, resulting in a trend towards putting both a small primary cache and a larger secondary cache directly on the processor chip of high speed processors such as the Alpha [Han93]. Shared-memory multiprocessor systems, now commonplace in the commercial marketplace, further exacerbate the problem by requiring processors to compete for the bus in order to satisfy requests to main memory, introducing additional latency due to bus contention and cache interference. Recent high end systems SMP systems can be configured with 12 to 20 or more processors. Distributed memory multiprocessor systems that provide a global memory abstraction, including non-uniform memory architectures (NUMA) such as DASH [LLG$^+$92] and systems based on Scalable Coherent Interconnect [Gus92], such as the Convex Exemplar suffer the additional overhead of a global communication network that can extend memory access times for remote memory to hundreds of clock cycles. Caches can eliminate much of the memory latency by placing the most frequently referenced instructions and data in fast SRAM memory, but even infrequent misses can have a significant impact on system performance.

Most of the published work on data cache performance has been focused on technical workloads, which tend to have poor data cache performance. Technical workloads usually consist of single user loop-intensive scientific and engineering applications that involve little operating system activity.

1.1 Technical Workloads

Scientific and engineering programs tend to exhibit considerable locality, with only a few loops generating the bulk of all memory references. Software controlled prefetching has proven effective in hiding a significant amount of latency in array based data structures where future address references are readily computed from the surrounding loop variables [Mow94] [KL91] [Por89]. Software controlled prefetching incurs the overhead of separate prefetch instructions for each prefetched cache line. Block prefetching loads multiple cache lines of contiguous memory with a single command. Chen and Baer showed that the instruction overhead could be significant, ranging from 0.9% to 8.6% of execution time for the mix of scientific and numerical applications that they evaluated, even when block prefetching was supported [CB94]. Part of the overhead comes from the prefetch instructions. Additionally, the index values of each prefetched array element must be computed twice: once when the array element is prefetched, and again when the array element is actually used. One goal of the proposed approach is to reduce the insruction overhead of software controlled prefetching when applied to array based codes.

1.2 Commercial Workloads

Recent measurements on commercial workloads have shown their system resource usage characteristics to be considerably different than those of the commonly used scientific and engineering benchmarks [MDO94]. Commercial applications such as the TPC benchmark suites distributed by the Transaction Processing Performance Council run in multi-user environments. Unlike technical workloads, they tend to consist of many user processes exhibiting short run-lengths because of frequent I/O operations. A substantial part of the application path-length tends to fall in the operating system kernel. The execution paths tend to have fewer loop iterations and more non-loop branches than technical codes.

Clark showed that cache performance is significantly lower than application only traces would indicate [Cla83]. Using traces of VAX memory references, Agarwal *et al* showed that the OS could be responsible for over 50% of the cache miss rate [AHH88]. Torrellas *et al* showed that the OS significantly affects cache performance, stalling the processors for 17% to 21% of their non-idle time for a mix of user-level and commercial data base workloads [TGH92]. When OS interference with the applications' cached working sets is factored in, this number reached up to 25% [1]. Chen and Bershad observed that OS data and instruction locality is considerably worse than that found in applications [CB93]. Maynard *et al* noted that the branching behavior of operating system codes tend to be significantly more random than that of applications, causing more cache lines to be touched. This results in a much larger footprint and higher cache miss rate [MDO94]. They also found that some commercial applications, and database applications in particular, exhibit caching behavior very similar to that of the OS. In summary, commercial codes exhibit considerably less data locality and instruction reuse, and both data and instruction cache miss rates are substantially higher.

Torrellas [TXD94] and Xia [Xia96] found that careful manual restructuring of the basic blocks of operating system codes could substantially reduce the instruction cache miss rate. Their approach requires a labor-intensive restructuring of the operating system for each generation of microprocessor and cache configuration. A second goal of this approach is to reduce the instruction cache miss rates of applications by increasing instruction cache reuse. The required program transformations are expected to involve less continual reengineering.

The multi-user commercial workloads investigated by Maynard *et al*, Torrellas et al, and others exhibit radically different data reference patterns than array based applications. While software controlled data prefetching has proven effective in hiding a significant amount of memory latency in array based applications, direct application of existing data prefetching schemes has not been shown to be effective for commercial applications. The third goal of our approach is to extend software controlled data prefetching to arbitrary pointer based data structures in commercial systems in general. The approach should lend itself particularly well to transaction processing and other client/server environments.

Software controlled prefetching has the potential to provide a cost-effective means of improving system performance by hiding memory latency. Software controlled data prefetching is currently applicable only to array-based codes, and incurs significant runtime overhead. We demonstrate that a combination of prefetching and coarse grain multithreading can hide memory read latency without introducing significant runtime overhead. The proposed approach allows pointer based data structures to be prefetched while simultaneously improving instruction cache behavior.

[1]Torrellas' evaluation environment consisted of a four-way SMP utilizing 33 MHz MIPS R3000 processors. This number can be expected to be considerably different for 12- to 20-way SMP systems running at up to 440 MHz today.

2. Related Work

A general-purpose approach to hiding latency has been the holy grail of large-scale computing since memory was identified as a performance bottleneck. This was addressed as one of the two fundamental issues in multipocessing by Arvind and Iannucci [AI87]. The widening gap between microprocessors and memory speeds is becoming an increasingly important factor in system performance in uniprocessors as well.

2.1 Multithreading

The Denelcor HEP and its successor, the Horizon were among the earliest architectures to employ multithreading to hide memory latency [Jor85]. The HEP consisted of a 16-element execution pipeline that performed a context switch at every clock cycle. Only a single instruction from any one thread was allowed to occupy the instruction pipeline at any given time, which required a large number o threads to effectively utilize the machine, and caused extremely poor single thread performance. Synchronization occurred through full/empty tags for each memory word and register. Parallelism had to be explicitly specified in HEP Fortran [Smi81].

Several incarnations of multithreading architectures have evolved since then, including April [ALKK90] and more recently an interleaved multiple context processor [LGH94]. April performs a context switch only when a memory request cannot be satisfied by the cache or local memory, or when explicitly requested, such as for a failed synchronization attempt. The expense of a context switch can be high because it requires the execution pipelines to complete processing. In contrast, the multiple context processor interleaves the execution of multiple threads at a per-instruction level, similar to the HEP. In contrast to the HEP, though, it employs local caches and execution unit pipelining. This results in efficient use of the processor even in the single thread case, as well as in fewer pipeline bubbles on a cache miss; only the thread that suffers the cache miss is actually stalled. Thus if the processor supports n simultaneous threads, only $1/n$ pipeline bubbles will occur for each thread that blocks on a cache miss.

2.2 Dataflow

The idea of multithreading was taken a step further in the MIT Tagged-Token Dataflow Architecture, where synchronization occurred at the instruction level [NA89]. Dataflow processors rely on low-cost context switching as a method of hiding memory latency and synchronization overhead [AI87]. Dataflow architectures restrict their application domain to dataflow graphs, which avoid all but data dependencies. This allows them to maximize the amount of parallelism exploitable by the hardware, right down to the instruction level.

Iannucci considered processor architectures to constitute a continuum, with von Neumann architectures occupying one end of the spectrum and dataflow processors at the other end [Ian88]. He recognized that it is possible to pick a point on this continuum that represented a much coarser granularity of parallelism than at the instruction level, and proposed a hybrid architecture between von Neumann and Dataflow. Large sequences of serial code can be identified in most dataflow graphs, and the most efficient execution of serial code occurs on a serial processor equipped with optimizations resulting from 30 years of engineering experience [Got91].

John Ellis pointed out that, in spite of the promises of scalability and ready access to program parallelism, no commercially viable general-purpose architecture at the dataflow end of the spectrum had been introduced by

1985 [Ell85]. In the meantime, dataflow as a hardware architecture appears to be nothing more than a historical footnote, perhaps because the economics and rapid evolution of microprocessors have made specialized hardware economically non-viable, or perhaps because the constraints of the single assignment languages on which dataflow represent too radical a departure from existing practices. In any case, they completely neglect the existing investment in software.

2.3 Prefetching

Prefetching decouples the initiation of memory read requests from their arrival. If enough work can be found between the time the memory read request was initiated and when it is satisfied, overlapping computation and communication has the potential to completely hide memory latency. If the memory system is pipelined, memory references can also be overlapped with other memory references.

One of the most common forms of prefetching is very commonplace in modern cache designs, where cache lines are larger than the memory word size. Some RISC processors will go so far as to reorder the memory requests so that the word that generated the miss will be satisfied first [IBM90]. In this instance, the hardware is essentially guessing that a temporally proximate future memory reference will fall within the same cache line. In multiprocessor systems, increased line sizes carry a concomitant potential for false sharing, where several processors that share no data invalidate each others lines because they each store to words that fall into the same cache line. Some modern RISC processors such as UltraSparc prefetch instructions for several cache lines beyond an instruction that generates a miss in the instruction cache.

Jouppi's proposed *multi-stream buffers* were a step in the evolution of prefetching sequential operands [Jou90]. Stream buffers automatically prefetch additional cache lines, starting at the initial miss target. His measurements indicate that instruction streams break their sequential access pattern by the time the sixth successive cache line is prefetched. Data reference accesses patterns are less regular, allowing only a 25% reduction in data cache misses to be achieved. This form of speculative prefetching can increase memory traffic by prefetching cache lines that will never be used, and thus may not be appropriate in a multiprocessor environment.

Porterfield first took advantage of compile time information to predict future references and prefetched all array references in inner loops a single iteration ahead [Por89]. He then refined his algorithm to take into account dependence information and to estimate the number of loop iterations before the loop began accessing data that would no longer fit into the cache. Klaiber and Levy extended Porterfield's work to fetch into a separate fetch buffer more than a single iteration ahead [KL91]. Mowry noted that, in addition to representing an inefficient use of chip area, their approach makes non-binding prefetches difficult [Mow94]. The non-binding property of prefetches is essential to ensuring program correctness in a multiprocessor environment. Lam *et al* proposed algorithms to predict cache misses in nested loops in order to optimize blocking algorithms for various cache geometries to maximize cache reuse [LRW91]. Wolf incorporated these ideas into a compiler that performed automatic parallelization and locality optimizations via loop slicing [Wol92]. Mowry extended Lam and Wolf's algorithms to incorporate prefetching in the form of software pipelining. The salient features of this approach are that Wolf's algorithms perform reasonably well at predicting misses. These algorithms are applicable to loop-intensive numerical codes, and do not address other data structures and algorithms. Software controlled prefetching is an important aspect of the proposed approach, and therefore a

```
  for ( i=0; i<256; i++ )
      X[i] = Y[i+1] + Y[I+2] - Z;
```

Figure 2.1: Original Loop.

```
/* prolog */
for ( i=0; i<4; i+=2 ) {
   PREFETCH(&X[i]);
   PREFETCH(&Y[i+1]);
}
/* steady state loop */
for ( i=0; i<252; i+=2 ) {
   PREFETCH(&X[i+4]);
   PREFETCH(&Y[i+5]);
   X[i] = Y[i+1] + Y[i+2] - Z;
   X[i+1] = Y[i+2] + Y[i+3] - Z;
}
/* epilogue */
for ( i=252; i<256; i+=2 ) {
   X[i] = Y[i+1] + Y[i+2] - Z;
   X[i+1] = Y[i+2] + Y[i+3] - Z;
}
```

Figure 2.2: Software pipelining applied to a simple loop

brief introduction is in order. Borrowing an example from Chen and Baer [CB94], consider the code transformation in figure 2.2.

The loop is split into a prolog, a steady state loop, and an epilogue. The prolog prefetches the data for the first for iterations of the steady state loop. The steady state loop issues prefetch instructions for the operands required 4 iterations later, while the epilogue executes the last four iterations without issuing any new prefetch operations. This example assumes that the cache line holds two array elements, so the loop is unrolled to avoid unnecessary prefetches, which can cause additional pressure on the registers [CB94]. In general, prefetches are scheduled $\lceil \frac{m}{e} \rceil$ loop iterations ahead, where e is the estimated execution time of the loop and m is the memory latency. Chen and Baer comment on the potential for prefetched data to be displaced before they can be used or to interfere with the working set, although Mowry tended to double the prefetch distance in his experiments without suffering significantly from interference effects [Mow94].

2.4 Non-blocking Loads

Non-blocking loads are a special form of prefetching. Non-blocking loads are register load instructions that allow processing to proceed upon a data cache miss until the target register is actually referenced. This approach has a number of severe limitations. Loads are *non-blocking* but not *non-binding*. A load is considered *binding* when the referenced data is no longer exposed to the coherency mechanisms of the memory hierarchy, in this case once it has been assigned to a register [1]. This means that the compiler must ensure that no stores occur to the address for which a load has previously been initiated [Mow94]. A further problem arises from the need to find sufficient work between register loads to keep the processor busy. This issue is universal to all approaches to hiding read latency. It is a greater problem for non-blocking loads because of the additional pressure on the registers, since the target register is unavailable for the entire duration between the time it is first referenced to the time that it is actually used. Farkas *al* used a trace scheduling compiler retargeted to an architecture supporting non-blocking loads to increase the distance between a load instruction and the first use of the corresponding target register [FJ94] [2]. The branching behavior of non-numerical codes makes it unlikely that

[1]Prefetches to specialized prefetch buffers also suffer from this problem [KL91] [Jou90].

[2]It is conceivable that the available distance may be further constrained by a trend towards multiple-issue and VLIW architectures. Adding registers to increase the available distance introduces a host of other problems, including the number of memory locations bound to registers and the concomitant reduction parallelism available across processor boundaries and between threads in multiple context environments.

even a VLIW compiler can find sufficient intra-thread parallelism to keep the processor busy without resorting to speculative execution [TXD94] [MDO94]. Given the poor instruction cache performance of commercial workloads, any benefits derived from speculative execution is likely to evaporate quickly with the increased instruction cache miss rate.

2.5 Relaxed Consistency Models

Memory consistency models have a significant impact on performance in multiprocessor designs. *Sequential consistency* guarantees that the execution of instructions behaves as it would on a uniprocessor that does not employ any load or store buffers. This ensures program correctness in instances where multiple threads of control read and write the same data – mutual exclusion. In a uniprocessor, this problem is solved by ensuring that the results of store operations are visible to the coherence mechanisms of the memory hierarchy, either by making the write buffer (which is usually small) fully set associative as in victim caches [Jou90], or by using an write-allocate cache policy that ensures that the caches always reflect the values in the store buffers. In a multiprocessor, sequential consistency can be maintained by ensuring that all store operations stall until the main memory is updated. It has the unfortunate side effect of disallowing buffering of write operations, thus introducing considerable latency into the system.

Relaxed consistency permits a more arbitrary interleaving of read and write requests, based on the observation that it possible to get away with it almost all of the time [Pfi95]. At times when sequential consistency is essential, specialized operations ensure that the state of memory is consistent throughout the memory hierarchy. *Release consistency* is the most relaxed memory consistency model, requiring that synchronization occur via specialized *acquire* and *release* operations to acquire and release synchronization variables [GLL+90]. Mowry studied the impact of employing release consistency instead of sequential consistency on the Stanford DASH Multiprocessor in conjunction with software-controlled prefetching, and found that it could have a dramatic impact on performance [Mow94].

3. Proposed Solution

Software controlled prefetching has proven effective at hiding memory latency in many loop intensive codes. We would like to investigate means of hiding latency for a more general class of programs. Given the non-loop branching behavior of operating system and data base codes, it is unlikely that software controlled prefetching or other previous means of hardware prefetching will prove effective. Another technique must be found to hide latency. For prefetching to be effective, the prefetch address must be identifiable far enough in advance so that it can be prefetched into near memory by the time it is first referenced. There may be an insufficient number of instructions between the time a variable is bound to an address and when it is first used to permit the data to be prefetched in time. A common approach is to perform a *context switch* before the first use of the remote data. The problem with this scheme is that the number of context switches is a function of the number of remote references. Since each context switch carries overhead, the context switch time may quickly overwhelm any benefit derived from hiding the latency of the remote reference. If a sufficient number of remote references can be prefetched for an entire block of code at once, then only a single context switch is required. Based on this observation, we propose a method of combining coarse-grain multithreading with prefetching to hide memory latency. The general approach is described in section 3.1. Section 3.2 describes how this approach can be extended to allow software controlled prefetching to be applied to pointer based data structures such as binary trees. Finally, section 3.3 proposes hardware support for this method that substantially reduces the instruction overhead of prefetching.

3.1 Coarse Grain Multithreading

We initially constrain our problem domain to codes that can be partitioned into blocks in which all the data that will be accessed in the block is predictable between the time the block is entered and the first data reference. The purpose of this restriction is to enable the programmer or compiler to place prefetch instructions for the long latency memory references that occur within the block at the beginning of the block. In practical terms, this means that blocks must have a small number of entry and exit points. Xia showed that many of the frequently executed blocks of operating system codes meet this criterion [Xia96]. For the time being we also impose a second restriction, requiring that all instructions that perform long latency data references within the block must have a high probability of being executed. This restriction makes it possible to avoid issuing unnecessary prefetches. Figure 3.1 illustrates a block of code that does not exhibit this quality; it cannot necessarily be discerned by compile time whether `Y` will be referenced. This restriction does preclude many important data structures, such as binary trees. Strategies for prefetching such data structures are discussed in section 3.2.

Blocks that experience the highest instruction and data cache miss rates are identified using cache profiling [SP95], or simply by means of educated guesses on the part of the programmer. The generic schema for tranforming these code blocks to perform prefetching is as follows:

1. Determine the data that will be referenced within the block and place *prefetch* commands that prefetch the required data into cache at the beginning of the block. This information might be gathered through profiling or some other means of gathering cache statistics [SP95] [MDO94]. Compilers can

```
/* start of block */
...
foo(X);
if ( funky() )
        foo(Y);
foo(Z);
...
/* end of block */
```

Figure 3.1: Reference to Y is difficult to predict at the beginning of the block.

identify references that are likely to miss in the data cache for many array based codes [Mow94].

2. Provide the address of the instruction at which execution can continue once (sufficient) data has arrived to the context switch mechanism. This address is referred to as the *continuation. continuation*, in accordance with Graf [GHD+91].
3. Perform a context switch to some other piece of code for which data should be ready, if necessary.

This scheme presupposes a *runtime system* that coordinates the context switches. *Context* is defined as the state of the program required to let execution proceed at the specified continuation. Context switches can occur among multiple blocks that are allowed to execute in parallel. It is convenient to think of the blocks among which context switching occurs as *threads*. Threads are loosely defined as sets of instructions that are executed until they voluntarily yield control to the runtime system. Threads share a common address space with other threads running within the same system-level process. A thread has associated with it a certain amount of state – at the very least the program counter and register values. To reduce the amount of overhead required to save the context of a thread prior to performing a context switch, some architectures provide multiple contexts in hardware [ALKK90] [LGH94].

```
entry point:
    S(X,Y);
finished;
```

Figure 3.2: Orignal Code Block

```
entry point:
   prefetch data(&X[0],count);
   prefetch data(&Y[0],count);
   prefetch instructions(cont,fini);
   context_switch;
cont:
   S(X,Y);
fini:
   ...
```

Figure 3.3: Multithreaded Prefetching Code Block Schema.

The original code block of figure 3.2 is instrumented with prefetch instructions for data and possibly instructions, as illustrated in figure 3.3. The details of the context switch are dependent on the amount of state that must be saved at the point of the context switch, and how much context can be supported in hardware. A generic schema for context switching in the absence of hardware support is provided in figure 3.4. The context switch schema minimizes the overhead of a context switch in instances when the runtime system determines that the current thread is to continue executing, since no context is saved unless a different target is determined. The call to `next_thread` causes the runtime system to return the thread identifier of the next thread scheduled to run. If the thread identifier matches that the current thread, then the current thread continues executing. Otherwise, the context required to permit the current thread to run, once it is rescheduled, is saved. The entry point of the current thread, identified with the `context_restore` label in figure 3.4, is then passed to the runtime system in order to provide the reentry point of the current thread.

There are several problems with this scheme. First, the overhead of a context

```
context_switch:
   target = next_thread();
   if ( target != this_thread ) {
      thread_save(this_thread);
      thread_switch(this_thread,restore);
   restore:
      thread_restore(this_thread)
   }
continuation: ...
```

Figure 3.4: Context switch schema.

switch may severely reduce any performance benefits derived from having the prefetched data in the cache unless context switches can be performed without significant overhead. The simple `context_switch` statement hides the fact that, at the very least, the continuation address must be communicated to the thread control software. The overhead of saving thread context may be addressed by providing multiple contexts in hardware [ALKK90] [LGH94]. Second, there must be another block of code that is ready to run in order to keep the processor busy while the data for the current block is being prefetched. The other block should not suffer significant cache conflicts with the data being prefetched. Evaluating which blocks of code are ready to run can also be expensive, depending upon the complexity of the underlying mechanism. For software pipelining, scheduling is a natural result of the looping mechanism, but our target codes are not constrained to loop intensive codes. Mowry showed that a victim cache can be effective in hiding some of the spurious cache evictions due to conflicts [Mow94], but clearly the currently executing context should not be a loop intensive blocked algorithm that effectively utilizes most of the cache before yielding to another thread. A specialized hardware *context unit* to support low overhead scheduling of code blocks and control prefetching is proposed in section 3.3.

3.2 Strategies for arbitrary Data Structures

Pointer-based data structures do not lend themselves particularly well to prefetching. Consider a search through a linked list of structures, where the a pointer to the head of the next structure is the only means of determining the address of the next element. Prefetching clearly takes on an entirely different meaning than when the address of the next element can be determined as a function of the base address of a data structure and the induction variables. Traversal of a linked list is described in section 3.2.1. The branching behavior of many applications such as tree structures precludes the use of software controlled prefetching because the traversal through the data structure essentially follows a random path. Consider the traversal of a binary tree. If both the left and right node of a tree are always prefetched, then one prefetch target will usually have been prefetched in vain. To make matters worse, software pipelining is scheduled $D = \lceil \frac{l}{s} \rceil$ loop iterations ahead in order to hide latency, where s is the execution time of the shortest path through the loop and l is the prefetch latency. It is obviously not desirable to prefetch up to 2^D nodes when only D are required for the comparisons that control the traversal through the tree.

While a single traversal of the tree does not provide sufficient opportunity to exploit software pipelining, many traversals performed concurrently can provide sufficient parallelism to allow software pipelining to be exploited. Conceivably, the traversal of common data structures in many codes is not initiated in tight loops, but is temporally distributed. In general, some codes that have runtime profiles similar to that illustrated in figure 3.5 can be restructured so that traversal of the data structures is performed in parallel. Software pipelining can then be applied. In an online transaction processing environment, for instance, multiple temporally proximate transactions can conceivably

```
    ...
traverse( DataStructure, key1 );
    ...
other_work_1();
    ...
traverse( DataStructure, key2 );
other_work_2();
    ...
traverse( DataStructure, key3 );
    ...
```

Figure 3.5: Code exhibiting potential for temporal restructuring.

be grouped for simultaneous traversal of the data structure.

The premise behind the approach is that a single unit of work performed on a given data structure may not provide sufficient opportunity to hide the latency via software pipelining, work is allowed to accumulate until a threshold is reached or a request for immediate resumption forces work to proceed. Traversal of a binary search tree is described in section 3.2.2, followed by a recursive version in section 3.2.3.

3.2.1 Searching an Unordered Linked List

Neither non-blocking loads nor traditional software controlled prefetching can help hide the latency encountered in traversing a linked list, although both techniques may become applicable after program transformation. The list header points only to the first element on the linked list, and subsequent elements cannot generally be known in advance. Recall that, in order to hide the memory latency of references in loop-based codes, prefetches may have to occur multiple iterations in advance. This is problematic for several reasons. Some pointers may generate memory exceptions. This problem is solved by making prefetches non-excepting [Mow94]. A larger problem is that the address of the nth element requires that the addresses of the pointers to the `next` field of each preceding element in the linked list be resolved. While Mowry's compiler algorithm is able to prefetch a single level of indirection in a dense array, a linked list represents an arbitrary level of indirection.

The snippet of code in figure 3.7 illustrates a search through a linked list using software pipelined prefetching[1]. The overhead consists of the prefetch instructions and the prolog. The prolog, as coded, is unlikely to yield any direct benefit because the first five prefetches are likely to miss; it does allow us to prefetch far enough in advance during the steady state to hide most of the memory latency. The variable `PipeDepth` of this example represents the *prefetch distance*, i.e. the number of iterations that the code is executed in advance in order to hide the memory latency. If the memory latency is 100 cycles and the prefetch distance is 10, this could impose a 1000 cycle stall time.

Alternatively, the list header can maintain a separate array whose elements always point to the first `PipeDepth` elements instead of just the first element. Additional overhead is incurred in the extra time required for insertion and deletion of the first `PipeDepth` elements for the benefit of completely avoiding the initial misses on a search[2]. There is also the overhead associated with maintaining an array of `PipeDepth` elements, which is expected to have a neglible impact on the data footprint. If `list->head` is now assumed to be an array of `PipeDepth` elements, the prolog code is illustrated in figure 3.8

If the desired element is located at the front of the list, then any extra prefetch operations represent pure overhead and unnecessary additional memory traffic. The same

[1]Remember that this example is for the purpose of illustrating how a pointer chain may be prefetched. If a linked list is long enough and searched frequently enough to warrant prefetching, a more efficient data structure would presumably be selected.

[2]If the list is unordered, as assumed here, then insertion is only more expensive for the first `PipeDepth` elements, since the list header can include a pointer to element `PipeDepth+1`.

```
traverse( DataStructure, key )
{
   if ( accumulated keys < threshold ) {
      add key to set
      perform context switch
   }
   Software Pipeline traversals
   enable stalled threads
}
```

Figure 3.6: Accumulation of sufficient traversal requests to permit effective software pipelining.

```
list_element_p
search_list( list_p list, int key )
{
  int i, j, c, s;
  list_element_t *p = list->head, *l;

  /* prolog */
  s = min( PipeDepth, list->count );
  for ( i=0; i<s && p!=NULL; i++ ) {
    PREFETCH( &p->key, sizeof(int) );
    PREFETCH( &p->next, sizeof(list_element_p) );
    p = p->next;
  }

  /* steady state */
  for ( c=0, l=list->head; p!=NULL; p=p->next, l=l->next, c++ ) {
    PREFETCH( &p->key, sizeof(int) );
    PREFETCH( &p->next, sizeof(list_element_p) );
    if ( l->key == key ) {
      printf("found key %d at position %d\n", key, c );
      return l;
    }
  }

  /* */
  for ( ; l!=NULL; l=l->next, c++ )
    if ( l->key == key ) {
      printf("found key %d at position %d (postamble)\n", key, c );
      return l;
    }
  return NULL;
}
```

Figure 3.7: Software Pipelined prefetching of a linked list.

```
for ( i=0; i<s && p[i]!=NULL; i++ ) {
  PREFETCH( &p[i]->key, sizeof(int) );
  PREFETCH( &p[i]->next, sizeof(list_element_p) );
}
```

Figure 3.8: Prefetch Prolog of Linked List Search.

is true for any prefetches issued for elements in the linked list located after the desired element. Since we are primarily concerned with aggregate throughput, however, it is reasonable to expect that the cost of these few cases can be amortized in the course of many iterations over a sufficiently long list. Naturally, a more time efficient data stucture might be a better choice than the linked list selected as an illustrative example here; a means of applying prefetching to the search of a binary tree is discussed in the next section.

3.2.2 Binary Tree Search

The following example illustrates the implementation of a multithreaded tree search. The original version in figure 3.10 includes the definitions of data structures and globals used in both this and the recursive version described in 3.2.3. For the sake of simplicity, the example assumes lightweight threads that perform blocking search operations on a static binary tree, i.e. one which does not change between invocations. There is no strict requirement that the application be implemented using lightweight threads – it need only support software pipelining. Provided that there is some other synchronization mechanism, the search request can be implemeted as a non-blocking call. In the example provided in figure 3.13, threads that submit requests to search the tree are added to an array of requestors until the search request can be satisfied, although the thread could be replaced with a callback function, or simply a pointer back to the data structure that creates the correspondence between each search request and the corresponding answer. The goal is to accumulate enough search requests to allow us to pipeline the searches for software controlled prefetching; how this is accomplished is dependent on the application – a multi-threaded client/server application will behave differently than a compiler.

As each request is received, the root node of the tree and the search key are added to the *search node set*. Static or dynamic analysis of the ideal prefetch distance determines a threshold below which prefetching is not considered effective, represented by the global variable `barrier` in the sample code of figure 3.11. If the number of reqests for a search falls below this threshold, the search is postponed until a sufficient number of requests have been submitted to allow processing to proceed. This threshold is thus also the maximum number of keys for which searches are conducted in the course of a single invocation of the simultaneous search function `tree_gang_search()`. Since a binary search tree is not necessarily balanced, some keys will be matched earlier than others. The number of keys being sought will consequently diminish as the search progresses through the tree, and the loop over the search keys becomes progressively tighter. Eventually the number of remaining keys will be too small to effectively hide the memory latency.

There are two complimentary means of ensuring that a sufficient number of searches are in the software pipeline. First, a lower bound on the number of simultaneous searches is established. If the number of keys remaining in the search set falls below the lower bound, then further searching is postponed and the function returns, allowing other work in the system to proceed. The stragglers remain in the set of search nodes until the next call to the routine. As long as simultaneous searches continue to be conducted, the search for each key is guaranteed to complete. Since this is a binary search tree, insertion of additional tree nodes will not adversely affect this scheme, although any deletions from the tree between instantiations of the searches will invalidate the search. Second, the initial threshold that triggered the search is adjusted so that most of the latency can be hidden most of the time without introducing significant self-interference or violating other system requirements such as service response time. Recall that the first call to the simultaneous search function initially has only pointers to the root occupying the search node set. While the MSHRs ensure that only one memory request is outstanding to the same cache line at any given time, the initial iterations of the search do not provide enough prefetch targets to hide much latency. A positive side effect of postponing searches for the stragglers is that a diverse set of nodes begins populating the set of requested nodes in the course of a few instantiations of the simultaneous search. A drawback is that the algorithm becomes sensitive to deletions. The algorithm can be made robust with respect to deletions by adding a state field to the tree header and marking the tree as "dirty". requiring remaining searches to start at the root again and then be forced to completion to guarantee termination[3].

The results for various threshold values are displayed in figure 3.9. The speedup varied from 28.7% to 32.3%, where the completion threshold corresponds to the pipeline depth, and the startup threshold values refer the the number of requests that were accumulated before computation would commence. Since the difference between the startup and the completion thresholds represents the number of requests actually worked on, the relatively low performance improvement of the runs with both low completion and low startup threshold not surprising.

3.2.3 Recursion

Recursion poses a special challenge to the approach. In order to search a tree recursively, several nested sets of subroutine calls could be kept in progress simultaneously, with context switches between them via calls to *longjmp()*. This adds significant overhead in form of saving registers, as well as adding the potential for a significant amount of stack growth. Although the sum of the stack sizes is no worse than it would be if the searches were performed sequentially, there is potential for interference among the different threads due to the resulting increase in the data footprint.

These issues can be effectively side-stepped by performing the search in a manner similar to the loop-based tree search of section 3.2.2. The code in figure 3.14 illustrates a recursive version of the search described in section 3.2.2. The request is submitted via the blocking routine of figure 3.15, and the recursive search is illustrated in figure 3.14. The recursion is unravelled if the available amount of work does not meet the lower bound criterion on the software pipeline depth. For this particular application this is of no consequence, but for applications which rely on maintaining the state of the stack variables from prior procedure invocations, this could prove more of a problem. In these cases the search can be allowed to complete without regard for the lower bound, at the expense of more memory stalls.

3.2.4 Applicability

The approach requires a considerable transformation of the underlying application, which now must be able to schedule work in a decoupled manner. From the perspective of an object oriented system, a message is sent to an object that employs accumulation

[3]There are presumably better solutions, but this simple one suffices in order to show robustness.

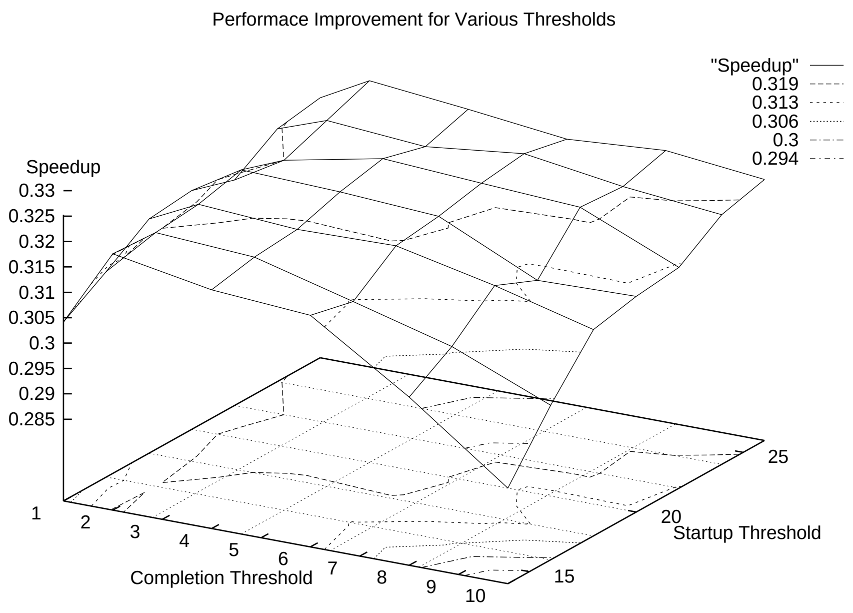

Figure 3.9: Threshold and Pipeline Depth Values for Prefetching.

requires the caller to handle postponement of the result. This means that the caller essentially performs a non-blocking request to the server. One approach might be to treat access to data structures that are expensive as one might any other long-latency operation – by blockking the calling thread. I suspect that, in practice, this would involve too much overhead. A better approach would be to parallelize the subsystem so that featherweight threads can be utilized.

If the application that is being optimized is a compiler, this might correspond to initiating syntax analysis of different functions within the program simultaneously, and then processing the results for each function separately.

For operating system codes and other systems with real-time constraints, a microtimer can be provided with a small amount of additional hardware. Short duration timers can be expensive to implement efficiently in software for a significant number of threads. These timers can be used to enforce a scheduling discipline that prevents time critical activities from expiring if an insufficient number of events have accumulated to trigger processing.

Because commercial applications exhibit significant instruction cache miss rates [TXD94] [MDO94] [Xia96], this approach benefits the implementation even in the absence of prefetching hardware because it guarantees, at the very least, instruction cache reuse by forcing temporal locality of instruction references. Conceivably data locality benefits as well, although it is not clear to what extent this can make a difference except where non-blocking loads can exploit the additional parallelism.

3.3 Context Unit

Ideally, we would like to be able to dynamically schedule units of work when the data they operate upon is available in near memory. The operating system does this for I/O requests, for for instance, but the

```
typedef struct node_s {
  int key;
  int depth;
  struct node_s *left, *right;
} node_t, *node_p;

typedef struct node_set_s {
  int count;
  int key[MAX_THRESHOLD];
  node_p node[MAX_THRESHOLD];
  thread_t *thread[MAX_THRESHOLD];
} node_set_t, *node_set_p;

typedef struct tree_s {
  int count;
  node_p root;
} tree_t, *tree_p;

int PipeDepth = 5, Threshold = 20;

node_p tree_search( tree_p tree, int key )
{
  node_p node = tree->root;

  while ( node != NULL ) {
    if ( key == node->key )
      return node;
    if ( abs(key) < abs(node->key) )
      node = node->left;
    else
      node = node->right;
  }
  return NULL;
}
```

Figure 3.10: Simple search of a binary search.

delays due to disk latency are of sufficient magnitude that operating system overhead is neglible. Since we're interested in hiding memory latency, which is measured in scores of cycles rather than hundreds of thousands of cycles, more attention must be paid to overhead. One way of reducing instruction overhead is to increase the amount of useful data prefetched with each prefetch instruction. Block prefetching loads multiple cache lines of contiguous memory with a single command. Chen and Baer showed that, even when block prefetching is used, the instruction overhead imposed by software pipelining can be significant, ranging from 0.9% to 8.6% for the mix of scientific and numerical applications that they evaluated [CB94]. The number of prefetch commands is not the only overhead associated with software controlled pipelining. Another form of overhead comes from having to recalculate loop index values multiple times – once for the prefetch, and again when the data is used. If software controlled pipelining can hide most of the latency of an array based application, reducing prefetch instruction overhead could have a significant impact on the final execution time. The proposed combination of

```
#define PREFETCH_NODE( i ) PREFETCH( node[i], sizeof(node_t) )

static int barrier = 0;
static node_set_t node_set = { 0, {0}, {NULL} };
static node_p *node = node_set.node, tmp;
static node_set_p tree_gang_search(void);

node_set_p
tree_search_accumulate( thread_p new_thread, tree_p tree, int new_key )
{
  thread_t *thread = node_set.thread, *tmp_thread;
  int *key = node_set.key, tmp_key;
  int prolog, i;

  if ( tree ) {
    thread[barrier] = new_thread;
    node[barrier] = tree->root;
    key[barrier] = new_key;
    barrier++;

    if ( barrier < Threshold )
      return NULL;
  }

  if ( !barrier )
    return NULL;

  return tree_gang_search();
}
```

Figure 3.11: Code to accumulate search requests.

prefetching and multithreading imposes additional overhead in managing the context switches.

A specialized hardware *context unit* to support low overhead scheduling of code blocks and control prefetching might consist of the following components:

- A table of prefetch schemes, where a prefetch scheme is defined by stride, element size, initial value, and the termination condition. Any number of prefetch schemes may be associated with a continuation, allowing an arbitrary number of prefetch targets to be pending for each continuation. This table acts as a cache that is indexed by a unique identifier for each continuation, such as the address of the first instruction in a block.
- A table of base addresses with pointers to the associated prefetch schemes.
- A state machine to compute prefetch addresses and continuation values. Prefetch addresses are immediately submitted to the memory hierarchy to be prefetched.
- *Synchronization ports* from which continuation values can be read. Reading a synchronization port may *trigger* the prefetch of a set of prefetch addresses associated with that port, with one port associated with each continuation.
- A queue of enabled continuations. A continuation is considered *enabled* if prefetching has been initiated and sufficient operands have arrived to drive the computation forward without encountering read stalls. In software pipelin-

```
node_set_p
tree_gang_search()
{
  node_set.count = barrier;
  prolog = min( PipeDepth, barrier );

  for ( i=0; i<prolog; i++ )
    PREFETCH_NODE( i );

  while ( barrier ) {
    /* If the threshold has been reached, postpone further searching.
     * The condition tree == NULL means we're "forcing" the search.
     */
    if ( barrier < PipeDepth && tree != NULL )
      break;

    /* The MSHRs check for duplicate prefetches.
     */
    for ( i=0; i<barrier; ) {
      if ( node[i] == NULL || (key[i] == node[i]->key) ) {
        barrier--;
        SWAP( node[i], node[barrier], tmp );
        SWAP( key[i], key[barrier], tmp_key );
        SWAP( thread[i], thread[barrier], tmp_thread );
        continue;
      }
      if ( key[i]) < node[i]->key )
        node[i] = node[i]->left;
      else
        node[i] = node[i]->right;

      PREFETCH_NODE(i);
      i++;
    }
  }
  return &node_set;
}
```

Figure 3.12: Loop-based multithreaded search of a binary tree.

ing, this condition is met once the epilog has completed.

The continuation values can represent branch target addresses, index values, or data addresses. The number of simultaneous outstanding prefetches should be kept small, while the number of available prefetch schemes may be arbitrarily large.

3.3.1 Synchronization

Prefetch operations must be synchronized between the processor and the context unit in order to support loop intensive codes. Once the context unit has the information of what to prefetch and how much to prefetch for each loop iteration, the CPU needs to signal communicate completion of each unit of work so that the context unit can initiate a new prefetch operation. This occurs when the CPU loads a continuation value from the

```
void
blocking_search( thread_t *thread, tree_t *tree, key_t key )
{
  node_set_t *answers;
  thread_t   *sleeper;
  if ( (answers = tree_search_accumulate( thread, tree, key ) ) == NULL ) {
    thread_sleep();
    return;
  }
  for ( sleeper = answers->thread; sleeper != NULL; sleeper++ )
    thread_awaken( sleeper );
}
```

Figure 3.13: Multithreaded search request.

```
int
tree_gang_search_recurse( tree_t *tree, node_set_t *node_set, int count )
{
  node_p *node = node_set.node, tmp;
  for ( i=0; i<count; ) {
    if ( node[i] == NULL || (key[i] == node[i]->key) ) {
      count--;
      SWAP( node[i], node[count], tmp );
      SWAP( key[i], key[count], tmp_key );
      SWAP( thread[i], thread[count], tmp_thread );
      continue;
    }
    if ( key[i]) < node[i]->key )
      node[i] = node[i]->left;
    else
      node[i] = node[i]->right;

    PREFETCH_NODE(i);
    i++;
  }
  if ( count >= PipeDepth && tree != NULL )
    return tree_gang_search_recurse( node_set, count );

  return count;
}
```

Figure 3.14: Recursive multithreaded search of binary a tree.

context unit.

```
write context id to context unit
if ( fetch method required )
     write context fetch methods
write any runtime context information
read continuation value
```

The first read of the synchronization port for a given context results in two sets of prefetches being enqueued; one for the first set of prefetch operands and one for subsequent sets. Consider the following example, which is representative of a loop declaration for a blocked algorithm:

```
for ( i=0; i<1000; i+=50 )
      loop( i, X );
```

The loop parameters, 0, 1000, 50 are parameters of the context fetch method and are communicated to the context unit via

```
node_set_p
tree_gang_search( tree_p tree, int new_key )
{
  static int barrier = 0;
  static node_set_t node_set = { 0, {0}, {NULL} };
  static node_p *node = node_set.node, tmp;
  thread_t *thread = node_set.thread, *tmp_thread;
  int *key = node_set.key, tmp_key;
  int prolog, i;

  if ( tree ) {
    node[barrier] = tree->root;
    key[barrier] = new_key;
    barrier++;

    if ( barrier < Threshold )
      return NULL;
  }

  if ( !barrier )
    return NULL;

  node_set.count = barrier;
  prolog = min( PipeDepth, barrier );

  for ( i=0; i<prolog; i++ )
    PREFETCH_NODE( i );

  barrier = tree_gang_search_recurse( node_set, barrier );
  return &node_set;
}
```

Figure 3.15: Mutithreaded binary search tree request, recursive version.

the call to `context_method` in the following code. The base addresses of the two long latency operands is written to the context unit via `context_base`. The transformed code is then:

```
context_method( X, 0, 10000, 50 );
for ( i=next i; i<1000; i=next i )
    loop( i, X );
```

The first instance of `next i` initiates the first prefetch set and enqueues subsequent prefetches. Thus the initial `next i` corresponds to the prolog of software pipelined prefetching. By replacing the original assignment `i=0` and the increment operation `i+=50` with `next i`, the recurring run time overhead of the synchronization has been reduced to a single load instruction per loop iteration[4]. If the cost of a the `next` instruction is the same as a register ALU operation, i.e. when the context unit is on-chip, then the total overhead of prefetching is the initial cost of preparing the context unit with the prefetch method and base address.

3.3.2 Macropipelining

Software pipelining causes at least one prefetch command to be issued for each cache line that is to be prefetched. It is possible to reduce some of this overhead by extending the prefetch command to fetch mul-

[4]More accurate would be the difference between a load instruction and a register ALU operation to generate the next index value.

tiple blocks at each iteration, but this only mitigates the problem and does not generalize beyond contiguous blocks of memory. *Macropipelining* is an extension of software pipelining that allows all of the operands for a loop to be prefetched without significant runtime overhead.

The context unit prefetches `Context[0]` and `Context[1]` using the methods provided, which are downloaded to the context unit. Their actual implementation is dependent on the capabilities of the context unit, but semantically they can be though of as remote procedure calls that generate loop index values, add them to the provided offsets, prefetch the appropriate cache lines, and write the resultant loop index value to a table. When the appropriate `next` command is issued by the processor, the index is matched against the supplied value via the `next i` instruction, and proceeds.

Figure 3.16 illustrates how macropipelining can be combined with *blocking* optimizations to increase cache reuse while minimizing the number of instructions required to prefetch the required cache lines. The `next` primitive is used to synchronize with the context unit, notifying it that work on one set of operands has completed and the prefetch of the next set may proceed. The value returned from the `next i` command is the continuation value associated with the next set of available data. If there are no constraints on the order in which blocks are accessed, then the described form of synchronization allows blocks to be computed out of order, depending on what can be accomodated in the cache. The described form of prefetching does not require the entire set of prefetch operands to be fetched before execution can proceed – once the first set of prefetch schemes have been executed, execution can proceed.

3.4 Summary

This study investigated a number of schemes to tolerate latency in commercial applications, beginning with a generic scheme that combines prefetching with coarse grain multithreading. An alternative scheme allows application of software pipelining to generic pointer chains, which are augmented with prefetch structures to reduce the misses during the prolog phase. Data structures in which the traversal path is not predictable, such as binary search trees, are parallelized by accumulating a sufficient number of traversal requests to enable software pipelining across multiple traversals. Where it is possible to *accumulate* traversal requests for a data structure in software systems that exhibit poor instruction cache behavior, between requests, the instruction cache hit rate should improve as well. Finally, a hardware mechanism is proposed that can substantially reduce the instruction overhead of prefetching when *macropipelining*, a generalized form of software pipelining, is applied to array-based and commercial codes.

3.5 Future Work

A significant potential benefit of the calculated procrastination strategy is that it should also improve instruction cache performance. Our simulation infrastructure does not currently measure instruction cache behavior at all. This needs to be investigated. The integration of the prefetch hardware simulator remains to be done. And finally, a broader set of applications need to be investigated in a more complex memory hierarchy than assumed for this study.

```
context_t Context[2];
Context[0].base = X;
Context[0].method = X_prefetch;
Context[1].base = Y;
Context[1].method = Y_prefetch;
size = Context[0].size;
PREFETCH( Context, 2 );

for ( i=next i; i<size(X); i=next i )
  for ( j=next j; j<size(Y); j=next j )
    for ( ii=i; ii<blocksize(X); ii++ ) {
      for ( jj=j; jj<blocksize(Y); jj++ )
        F(X[ii][jj],Y[jj]);
    }
```

Figure 3.16: Reducing prefetch overhead via Macropipelining.

4. Approach

There are several components to this research. The first demonstrates the effictiveness of combining prefetching and coarse grain multithreading on standard scientific, engineering, and numerical codes. A sufficiently representative set of standard scientific, engineering, and numerical benchmarks can be hand-coded to perform prefetching. Second, the restructuring of non-numeric codes is evaluated. This requires a creative approach, as discussed in section 4.1. Finally, the proposed hardware support is simulated at the behavioral level. It has already been implemented and tested but remains to be incorporated into the simulation environment.

Prefetching on its own does not achieve its full potential even in scientific applications, which can be attributed to several factors. In addition to introducing overhead to generate the addresses and execute the prefetch operations themselves, prefetching can have destructive side effects. A prefetch operation may evict a line from the cache that contains data that is still needed before the prefetched data is referenced. If the evicted block of memory is referenced again before the prefetched cache line is used, the prefetch operation has the effect of replacing a single miss with two misses and a prefetch operation. Similarly, a prefetch that arrives too early may be evicted by a cache miss before it can be used. Prefetches may evict the results of other useful prefetch operations before they are ever referenced. Thus the proposed solutions are evaluated for a number of cache configurations.

4.1 Evaluation

Validation of performance on scientific codes is relatively straightforward using the evaluation infrastructure currently in place. The benchmarks for SPLASH, NAS, and other numeric applications are widely available. Because of the amount of hand restructuring required, I will select an interesting subset of these benchmarks for evaluation.

A potentially challenging component of this thesis is evaluating the transformations on commercial codes. These tend to be very large software systems that are the result of hundreds of man-years of effort. It is unrealistic to think that a single person could be hope to complete the restructuring of a major application within the time-span reasonable for a doctoral dissertation, and is well beyond the scope of the proposed research. As a result, I plan to show that the advocated approach is applicable to a selected group of data structures and algorithms that can be identified by means of profiling or published results from the research of others. This information will include the following:

- the data structure and algorithm employed.
- the average amount of cache pollution between invocations.
- the throughput.

The data structures and and algorithms will be optimized, if possible, using the techniques proposed. If it is not possible, the reasons will be identified. If the throughput of the system is measured in transactions per second, for instance, we should be able to estimate the improvement as a result of our program transformations.

The statistics on cache pollution are used as a parameter to generate cache interference between successive invocations of optimized code, as outlined in figure 4.1. This is contrasted with unoptimized code in a manner described in figure 4.2. Cache statistics are only gathered for the sections of code within the loop that are being evaluated, and disabled for the remaining sections.

This makes it possible to estimate performance improvement on this section of code

```
/* optimized code */
while ( testing ) {
  perturb_cache( %dirty )
  turn on cache statistics gathering
  count = gang_schedule(work,results)
  if result != NULL {
    for ( i=0; i<count; i++ ) {
      result = results[i]
      turn off cache stats gathering
      process_result(result)
      turn on cache stats gathering
    }
  }
  turn off cache stats gathering
}
```

Figure 4.1: Evaluation of optimized code.

```
/* unoptimized code */
while ( testing ) {
   perturb_cache( %dirty )
   turn on cache stats
   result = schedule(work)
   turn off cache stats
   process_result(result)
}
```

Figure 4.2: Evaluation of unoptimized code.

over the required number of iterations. Caution must be exercised to avoid adversely affecting throughput for the selected system or application in other ways. For instance, for the tree example described above, this approach works well, but if I now require multiple trees to occupy memory at the same time, I have potentially adversely affected the memory and cache footprints of the application. In general, I'm operating under the assumption that memory is cheap and plentiful, just slow. Given both Tandem and Oracle's approach of throwing memory at any problem, this is a reasonable assumption – adding 32MB of memory for even a 10% performance improvement is a very good return on investment at todays memory prices.

Because the simulation environment is fairly complete, we can generate results over a broad range of cache pollution values. This shotgun approach allows us to show the level of performance improvement under a broad range of cache pollution scenarios.

4.2 Infrastructure

I have constructed an evaluation environment built around the MINT MIPS R4000 simulator front end [VF93]. MINT executes code compiled to run on a MIPS R4000 processor. System calls are passed on to the underlying operating system. The back end has been modified to generate and read address traces instead of relying on simulation to generate cache usage statistics. The simulator currently supports two types of traces: those generated by the back end of MINT itself, and PatchWrx traces [SP95]. It provides support for a multilevel cache hierarchy and a pipelined memory subsystem based on the RS/6000 [IBM90], but can be extended to model arbitrary pipelined memory subsystems. Parameters include bus width, memory interleave factor, number of outstanding requests on the bus, and the minimum distance between them. The memory subsystem simulator also allows for modeling of memory refresh events and static column DRAM setup and access time.

The back end includes a programmable access processor to support decoupled access/execute (DAE) architectures [JRH$^+$94] [SWP86] [Wul92] in combination with featherweight multithreading [Col94].

The integrated memory and cache system make it possible to collect numerous statistics on cache behavior, both with and without prefetching. Besides hit and miss rates, the cache subsystem counts the number of prefetch operations executed for operands not already in the cache, prefetches of lines already in the cache to measure unnecessary prefetches, misses to prefetched operands, and the number of prefetched lines that were evicted before they could be referenced at least once. The number times prefetches are evicted by other prefetches and by normal cache access conflicts is also tracked, allowing us to evaluate the efficacy of manual and compiler inserted prefetch operations.

Each cache line has been augmented with a *last reference register* that always contains the time of the last reference to that cache line. In the absence of a prefetching compiler this provides a means of identifying candidates for prefetching, based on which lines exhibit the greatest miss frequency. The time stamp indicates the earliest point in time at which a prefetch can be inserted without displacing a cache line that will be needed prior to the reference to the prefetched cache line. If the number of cycles between the time that a cache line becomes available and the time that the prefetch candidate is required is too short, then the prefetch is dropped and no prefetch trace event is generated. This approach enables us to construct an Oracle. An initial execution or trace is used to generate a prefetch event trace based on the earliest time that a prefetch can be inserted into the instruction stream without evicting a useful line. The prefetch event trace is then sorted by timestamp and merged with the original memory reference trace. The resulting execution time gives an upper bound on the improvement in overall execution time that can be achieved by prefetching.

Finally, the number of data cache misses caused by each instruction can be tracked to create a profile of the instructions that tend to cause the most data cache misses. This information provides an indicator of the amount of effort and the number of separate prefetch instructions required to achieve a given coverage factor. For each of the instructions selected to be preceded by a prefetch, it is possible to trace the addresses that missed, allowing evaluation of off-line and on-line prediction algorithms [KV94] [VK91].

4.2.1 Remaining Infrastructure Work

The model currently makes several assumptions: while it does model finite prefetch issue queues, it currently assumes infinite write queues. The modeling of prefetch issue queues and write queues is essential to accurately measure the stall times. There are some tradeoffs to consider in this model as well, since there is a potential for contention between the prefetch issue queue and the write buffer.

While arbitrary levels of caches are supported for non-prefetching, the model currently only supports a primary cache and memory subsystem with prefetching. Since most modern microprocessor based systems incorporate large second level or even third level caches, the system should be enhanced to support multiple levels of cache.

References

[AHH88] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[AI87] Arvind and Robert A. Iannucci. Two Fundamental Issues in Multi-processing. Technical Report MIT Computation Structures Group Memo 226-6, MIT, May 1987. Conference on Parallel Processing in Science and Engineering.

[ALKK90] A. Agarwal, B-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *17th Int. Symp. on Computer Architecture*, pages 104–114. IEEE, 1990.

[CB93] J. Bradley Chen and Brian N. Bershad. The Impact of Operating System Structure on Memory System Performance. *Operating Systems Review*, 27(5), December 1993.

[CB94] Tien-Fu Chen and Jean-Loup Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium of Computer Architecture*, pages 223–232, 1994.

[Cla83] D. Clark. Cache Performance in the VAX-11/780. *ACM Transactions on Computer Systems*, 1(1):24–37, February 1983.

[Col94] Dirk Coldewey. A Data Flow Approach to Thread Caching. Technical report, UCSC, Santa Cruz, CA, 1994.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, MIT, Boston, MA, 1985.

[FJ94] Keith I. Farkas and Normann P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *IEEE Computer Architecture*, pages 211–222, 1994.

[GHD+91] V.G. Graf, J.E. Hoch, G.S. Davidson, V.P. Holmes, D.M. Davenport, and K.M. Steele. The Epsilon Project. In *Advanced Topics in Data-Flow Computing.* Prentice Hall, 1991.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in scalable shared-memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–25, June 1990.

[Got91] Israel Gottlieb. Work Distribution of in the DSDF Architecture. In *Advanced Topics in Data-Flow Computing*, pages 381–382. Prentic Hall, Englewood Cliffs, New Jersey 07632, 1991.

[Gus92] David Gustafson. The Scalable Coherent Interface. *IEEE Micro*, 12(1):10–12, February 1992.

[Han93] Jim Handy. *The Cache Memory Book.* Academic Press, Inc., San Diego, CA 92101, 1993.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[Ian88] Robert A. Iannucci. Toward a Dataflow/Von Neuman Hybrid Architecture. In *Proc. 15th Annual Symp. on Computer Architecture*, volume 16, pages 131–140. IEEE, 1988.

[IBM90] IBM. *POWER Processor Architecture.* IBM Corporation, Advanced Workstation Division, Austin, Texas, 1990.

[Jor85] Harry F. Jordan. HEP: Achitecture, Programming and Performance. In *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, pages 1–40. MIT Press, Cambridge, Massachusetts, 1985.

[Jou90] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. 17th Annual Symp. on Computer Architecture*, volume 18, pages 364–373. IEEE, 1990.

[JRH$^+$94] Lizy Kurian John, Vinod Reddy, Paul T. Hulina, , and Lee D. Coraor. Program Balance and its Impact on High Performance RISC Architectures. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pages 370–379, January 1994.

[KL91] A.C. Klaiber and H.M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th Interrnational Symposium on Computer Architecture*, pages 43–53, 1991.

[KV94] P. Krishnan and Jeffrey Scott Vitter. Optimal Prediction for Prefetching in the Worst Case. In *SODA 94*, 1994.

[LGH94] James Laudon, Anoop Gupta, and Mark Horowitz. Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors. In *Multithreaded Computer Architecture*, pages 167–200. Kluwer Academic Publishers, Norwell, Massachusetts 02061, 1994.

[LLG$^+$92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH Multiprocessor. *Computer*, pages 63–79, March 1992.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[MDO94] Ann Marie Grizzaffi Manyard, Colette M. Donnelly, , and Bret R. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *ASPLOS-VI Proceedings*, pages 145–155. ACM Press, November 1994.

[Mow94] Todd C. Mowry. *Tolerating Latency through Software-Controlled Data Prefetching.* PhD thesis, Stanford University, 1994.

[NA89] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proc. 16th International Symposium on Computer Architecture*, pages 262–272, 1989.

[Pfi95] Gregory E. Pfister. *In Search of Clusters.* Prentice Hall, Inc., Upper Saddle River, New Jersey, 1995.

[Por89] A. K. Porterfield. Software Methods for Improvement of Cache Performance on Supercomputer Applications. Technical Report COMP TR 89-93, Rice University, May 1989. Cited in Klai91.

[Smi81] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Tutorial on Supercomputers*, page 425. IEEE Press, 1981.

[SP95] Richard L. Sites and Sharon E. Perl. PatchWrx – A Dynamic Execution Tracing Tool. Technical Report Systems Research Center, Digital Equipment Corporation, October 1995. Submitted for Publication.

[SWP86] J.E. Smith, S. Weiss, and N.Y. Pang. A Simulation Study of Decoupled Architecture Computers. *IEEE Transactions on Computers*, C-35(8), August 1986. Cited in John94.

[TGH92] Josep Torrellas, Anoop Gupta, and John Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *ASPLOS-V Proceedings*, pages 162–174, September 1992.

[TXD94] Josep Torrellas, Chun Xia, and Russell Daigle. Optimizing Instruction Cache Performance for Operating System Intensive Loads. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pages 360–369, January 1994.

[VF93] Jack E. Veenstra and Robert J. Fowler. MINT Tutorial and User Manual. Technical Report Technical Report 452, University of Rochester, June 1993.

[VK91] Jeffrey Scott Vitter and P. Krishnan. Optimal Prefetching via Dat Compression. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, October 1991.

[Wol92] Michael Edward Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, 1992.

[Wul92] W.A. Wulf. Evaluation of the Wm Architecture. In *Proceedings of the 19th Annual International Symposium of Computer Architecture*, pages 382–390, May 1992. Cited in John94.

[Xia96] Chun Xia. *Exploiting Multiprocessor Memory Hierarchies for Operating Systems*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1996.