Hierarchical Performance Modeling with Cache Effects: A Case Study of the DEC Alpha

Perry H. Wang

Advisor: Edward S. Davidson

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
email: phwang@eecs.umich.edu
phone: (313)763-6970

Ann Arbor, Michigan
March, 1995

Abstract

Although computers are becoming faster, today's high performance computer architectures require an efficient compiler in order to exploit the special features and avoid the special problems of that particular architecture. On the hardware side, a faster microprocessor is usually supported by faster cache, multiple instruction issuing, and efficient pipelining. To justify these features, the compiler must be able to schedule instructions well. Nowadays, most compilers support basic compiling techniques such as strength reduction, inlining, etc. Other techniques such as loop unrolling and cyclic scheduling to enhance and fine-grain scheduling to exploit instruction parallelism may also be used.

However, because microprocessors' speed is increasing faster than the speed of the memory system, there is a widening gap in speed between these two system components. This paper will discuss and demonstrate several aspects of why instruction scheduling for some kernel loops should be treated as a global problem, rather than in the traditional local greedy way of issuing each instruction as early as the microprocessor can handle it. In particular, instruction scheduling should consider the detailed implementation characteristics of the supporting cache. To support this idea, we have developed a performance bound model that includes cache effects for the DEC Alpha. By applying techniques such as loop unrolling and cyclic scheduling, new loop schedules substantially improve performance most of the time. In this research, seven out of the twelve LFK kernels running on the Alpha system have been improved through hand-scheduling that was guided by the performance bound model. The improved codes reach as high as 96% of the performance bound. The average clocks per floating-point operation (CPF) is 7.41 for the compiled LFK and 4.84 CPF through hand-scheduling, resulting in a speedup of 1.53 on average.

Table of Contents

List of Tables

List of Figures

1.0 Introduction

Scientific programs are usually dominated by floating-point intensive loops that iterate over sets of data. Our group has formulated upper bounds on performance for several machine architectures running application-specific tasks such as the Livermore Fortran Kernel (LFK) loops [6][8][11][12]. Formulating these performance bounds has focussed on several functional units which are typical bottlenecks. These units have been the issue unit, the floating-point unit, the memory unit, and a pseudo-unit for loop-carried dependence.

Caches are high speed memory systems that are transparent to the software and exploit the locality of memory references. For loop-based application code, there could be a significant speedup if the working set of a loop code can be kept smaller than the size of the cache. Our group's previous studies of the performance bound model did not include the effect of cache misses in formulating the performance bound models for two reasons:

1. Cache misses were considered to be unpredictable events, and

2. The size of the cache was typically large enough to contain the entire LFK working set.

But what if the working sets are larger than the size of the cache? One solution is to improve the memory reference patterns by dividing up the working set and iterating over the resulting smaller set of data. This approach is known as loop blocking for the cache. However, since it is not always possible to eliminate cache misses, this method does not give an accurate and fair measurement of the performance on a given machine architecture when running simple benchmarks that can easily be blocked. Another solution is to formulate the performance bound equations so as to include the effects of cache misses and allow the benchmarks to run unblocked. To incorporate cache miss events in a bound, they must be predictable and, in some sense, unavoidable. Cache misses can sometimes be predicted with a thorough understanding of the behavior of the cache and through well-structured fine-grained instruction scheduling of the loops.

This paper will focus on how to make cache misses predictable, and how to formulate the performance bound so as to include at least the leading-edge effects of cache misses. Cache misses are not always predictable for all loops, but will be accounted for when they are. This paper will also point out which types of loops still cannot achieve the bound and the reasons why they are unable to do so.

Besides the leading-edge effect, i.e. the run time penalty for an isolated cache miss, there are generally also trailing-edge effects. The trailing-edge effects are caused by the effect on one reference by earlier references whose activity has not been fully completed even though the processor is free to issue other requests, e.g. when a cache line which was referenced earlier has not yet been completely brought into the cache when it is referenced again, causing additional penalty cycles for this second reference. To achieve the optimum performance bound, trailing-edge effects must be completely eliminated. As another example, write buffers often used to enhance the performance of store instructions and later references also contribute to the trailing edge cache effects. Therefore, performance modeling that includes these effects of the write buffer would create a tighter, less optimistic bound. The speed of the secondary cache is also important since how fast the secondary cache responds to a primary cache miss is closely related to the performance of the primary cache. This paper will talk about how both the write buffer and the secondary cache can impede performance and how to take these impedances into consideration in the performance bound model.

As an example of formulating the performance bound model with the effects of the cache system, the Alpha was chosen because it has a small on-chip cache. As a true 64-bit microprocessor, the Alpha architecture was designed to be a high performance system. Its implementation, the DECchip 21064, running at 200 MHz, has the fastest clock of any single microprocessor ever built. However, with fast single-chip processors like the DECchip, with small on-chip caches, the secondary cache becomes one of the important factors in evaluating performance. It is believed that a machine like the Cray T3D will sacrifice significant performance due to its elimination of the secondary cache. But even with its secondary cache, the memory system of an Alpha workstation is still unable to keep up with the fast clock speed of the processor.

In this study, we will look at the performance of the Alpha (21064 implementation) on the LFK loops. We show that memory effects, including the predictability of the primary cache misses, the behavior of the write buffer, and the speed of the secondary cache, can impede the performance greatly. We then show how we can schedule the instructions so that these effects can be reduced to the minimum achievable. Then the performance model, with the inclusion of cache effects, is applied to analyze the performance of the Alpha on the first twelve LFK loops as compiled, and with hand-coded improvements.

2.0 Related Work

This paper exploits and further develops previous work on performance bounds and uses cyclic scheduling techniques as described below.

2.1 Performance Bounds

Definition 1: Bottleneck Unit

A bottleneck unit is any machine unit that is fully utilized.

Our group's performance bound model is used to estimate the optimum performance of the loop-dominated applications on particular systems. The performance bound model that we employ was first introduced in Mangione-Smith's thesis [11], where the formulation of the performance bound model was thoroughly explained. After identifying some possible bottleneck units, the performance bound is simply the maximum of a set of terms where each term is formulated as the busy time of one possible bottleneck unit. The four commonly used bottleneck units are:

- t_f, for the floating-point unit, is the number of clocks required in the floating-point unit to process all the essential floating-point operations in a loop iteration, assuming they are independent and processed at maximum rate.
- t_m, for the memory unit, is the number of clocks required in the memory unit to process all the essential memory operations, such as loads and stores.
- t_i, for the instruction issue unit, is the minimum time required for the issue unit to dispatch all the instructions in one iteration.
- t_d, the loop-carried dependence unit, is a pseudo-unit that models the minimum time per iteration to execute a recursion. Recursion exists when a result of one iteration depends on the corresponding result of a previous iteration. If no recursion exists, then t_d is zero.

When the iteration time bound is achieved, the unit selected by the maximum function is kept continuously busy by its essential operations. The other units are presumed to operate fully

concurrently with the bottleneck unit, i.e. they never stall the bottleneck unit. When they do, the iteration bound is not achieved. The iteration time bound is thus

$$t_l = max(t_f, t_m, t_i, t_d) \quad (1)$$

Three examples of architectures, the DEC 3100, the IBM RS/6000 and the ZS-1, were considered in Mangione-Smith's thesis. The performance bound models of these three machines were justified and compared. Subsequent work on the performance bounds is reported in the following.

Shih's directed study [13] was on the IBM RS/6000. His report singles out the specific causes of the performance gap between the bound and the delivered performance. The way in which the compiler had failed to optimize the code were identified, and code improvements to narrow the gap by removing these causes were individually applied resulting in 1.79 of the original performance and achieved performance that reached 93.701% of the bound averaged across the first 12 LFKs. This work, together with Mangione-Smith's results, are summarized in [12]

Boyd [8] developed the performance bound model into a hierarchical technique. The hierarchical MACS bound, introduced in this paper, can be applied to a Machine (M bound), a machine with a high-level coded Application of interest (MA bound), the Compiler-generated workload (MAC bound), and the actual Schedule of the workload generated by the compiler (MACS bound). Each bound level considers the named subset of M, A, C, and S and idealizes the others. The time bounds M, MA, MAC, MACS, and finally actual delivered performance, thus form a monotonically increasing series. The gaps between them can be associated with known causes and potential cures. The Convex C2 machine was used as a case study for this work.

Boyd [7] further studied the performance gaps between the levels of the MACS bound. One interesting point which relates to the current research lies in the gap P. Gap P is the performance difference between the actual measured performance and the MACS bound. This gap includes all effects ignored in the formulation of the MACS bound. "Cache effects" were listed as one of these unmodeled effects.

Azeem's master's thesis [6] was on the KSR-1. In this report, the performance bound for the KSR-1 was tightened up a little by including the effects of loop branches. They are affected by the number of times the loop is unrolled. However, these effects are small when the loop body is large.

Of all the reports and papers mentioned above, none of the work actually tries to include in the performance model the major unmodeled stalls in the P gap, the cache effects.

2.2 Cyclic Scheduling

Definition 2: Minimum Initiation Interval

The minimum initiation interval (MII) corresponds to the smallest number of cycles necessary to issue the instructions in a given loop iteration. MII must also be sustainable from iteration to iteration.

One of the scheduling techniques that will be widely used throughout this paper is cyclic scheduling, also known as software pipelining. Hsu's thesis [10] contains an excellent analysis of cyclic scheduling. In a cyclic schedule, all iterations have the same schedule, and instructions from later iterations are issued before all instructions of a previous iteration are issued. In other words, the instruction schedule for one iteration of a loop is expanded and several iterations are in execution concurrently. Successive iterations are started every MII clocks, and MII is typically

less than the time to issue all instructions of one iteration. Thus, optimum performance is achieved if at least one machine resource is fully utilized in steady-state by the essential instructions of the concurrent executions of several iterations. Cyclic scheduling is the most aggressive static scheduling technique known for producing optimum steady-state throughput.

Mangione-Smith [11] went further into the topic of cyclic scheduling by introducing the use of instruction templates with modulo reservation tables. This paper will also utilize instruction templates and modulo reservation tables to explain our scheduling methods.

3.0 Tightening up Performance Bounds

This section concerns the tightness of the performance bounds. As mentioned earlier, the previous work on the performance model ignored some performance loss factors and all unmodeled effects end up contributing to the P gap, the performance gap between the MACS bound and the actual measured performance. Modeling these effects within the bounds formulation will cause the MACS model to approach actual measured performance more closely, narrowing the P gap. A perfect model of actual performance would have a P gap of zero.

3.1 Essential Factors causing Performance Degradation

When the actual performance is far worse than the expected achievable performance, is the performance bound actually achievable? The performance model could have missed some vital machine-specific information which could lead to an unachievable performance bound. In this case, a tighter performance bound could be formulated to include the missing information. The first step is to find out the factors that cause performance degradation; the effects of these are visible as contributions to the performance gap P between the MACS bound and the delivered performance, as illustrated in Fig. 1(a). Then those contributing factors that cause an essential performance loss must be separated out. If the factors causing the performance loss are essential, the model should include such factors into the bound, as illustrated in Fig. 1(b). Nonessential factors

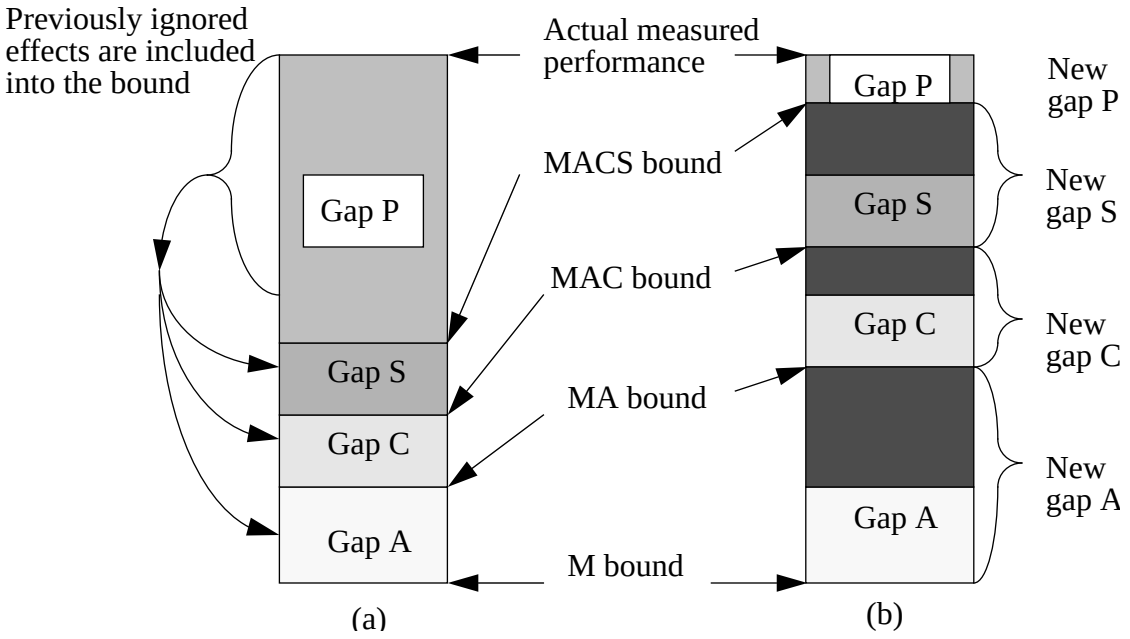

FIGURE 1. Performance Bound: Old (a) vs. New (b)

which can be avoided at some bound level should not be included in the bound at that level. For example, a factor that can be eliminated by rescheduling may be included in the MACS bound, for which the schedule is fixed, but not the MAC or other bounds, for which the schedule is idealized.

3.1.1 Nonessential Performance Degradation

Nonessential performance degradation can be removed through code optimizations. From the perspective of the MACS bound hierarchy, the M bound is the machine's peak floating-point performance, the A gap is caused by a non-ideal set of floating-point operations and by essential instructions of other types, the C gap consists of nonessential instructions, and the S gap consists of poor instruction scheduling. These gaps, particularly the C and S gaps, can often be reduced by developing and using better compilers. Therefore, the factors that cause the C and S gaps are often avoidable. Previous studies have tackled gap reduction well by hand-coding techniques that could be included in future compilers.

3.1.2 Essential Performance Degradation

Essential factors of performance degradation may be caused by inadequate hardware support, such as a small size of the cache, which may increase essential cache misses, or too few registers, which may increase register spilling. Essential causes of performance degradation cannot be avoided regardless of how well a compiler that uses today's best known techniques can optimize the code. If even more aggressive optimization techniques are developed in the future, it may be possible to reduce the effects of operations deemed "essential" today. From the perspective of the MACS bound, some factors in the P gap are essential in contributing to performance degradation and account for a portion of the stalls in this "unmodeled" gap. These factors were simply ignored in the previous work on the MACS model because those stalls were assumed to be unpredictable or insignificant. This paper deals with the cache effects as one of the main essential factors in performance degradation.

3.2 Steady-State Performance Predictability

A loop is said to be in steady state if all future iterations of an infinite loop would have same number of cycles. Steady-state can be achieved if the effects of all the instructions in the loop body are *predictable.* On the other hand, unpredictable events cause the run time of loop iterations to vary and thus cannot easily be incorporated within the bound equations. Cache misses were considered to be unpredictable in Mangione-Smith's thesis [11]. According to Mangione-Smith, "the performance bounds implicitly assume that memory latency is not an important factor for scientific codes." The justification for this observation was that the RS/6000 has a large enough data cache for the data sets in the LFKs used in this study. In this paper, however, cache effects are an important factor in designing the performance bound model for the DEC Alpha. And with a good memory reference strategy and good scheduling, many cache misses may in fact become predictable and be incorporated at the appropriate levels of the bounds hierarchy.

3.3 Counting Essential Cache Misses

To include cache misses in the performance bound model, we need to understand which types of misses are important in a steady-state inner loop and how we can identify them. One intuitive cache model identifies the three sources of cache misses as compulsory, capacity, and con-

flicts. Coherence misses are added for shared memory parallel machines. Compulsory misses, also known as cold start misses, generally do not contribute significantly to steady-state loops. However, capacity and compulsory misses do, and will be modeled in the MACS bound. When the data working sets of loops exceed the size of the cache, capacity misses occur. Conflict misses may occur when two or more accessed data blocks are mapped to the same set in the cache.

Consider a direct-mapped write-through cache, with corresponding elements of data arrays A and B mapped to the same cache lines. With a loop written as

```
For x = 1 to n
    C(x) = A(x) + B(x)
End
```

all the memory references to A and B will become conflict cache misses regardless of the cache line size.

Note that if arrays A and B are larger than the cache, but begin in different cache lines, then there will be one miss each time a new cache line of A or B is referenced, rather than a miss on every reference. This is the normal case for the LFKs running on the Alpha and is the basic assumption of the bounds model and the hand-coding effort.

If arrays A and B are each a little smaller than the cache size, and begin in different cache lines, then the normal assumption above does not hold since a few lines of A and/or B can remain resident in the cache and all references to them will be hits. If the error is small, we simply use the normal assumption. However, for successively smaller arrays, normal assumption is too pessimistic, and the bound model is adjusted. The normal assumption covers about half of the LFKs. A few more are covered by adjustment. The remainder are analyzed as special cases due to having a large number of arrays, two-dimensional arrays with a triangular reference sequence or non-unit strides, etc.

Two simple techniques, loop blocking and adjusting the relative alignment of two arrays, can often eliminate many cache misses in an application. However, they often do not work well on large complex applications. For this reason, such techniques are generally not allowed when running simple benchmark codes and using the results to evaluate competitive performance. Therefore, we do not use these techniques in this paper.

In this paper, we focus on the MA bound level. We count the number of essential loads and stores from the high level language code, as follows. Over m iterations of the inner loop, the number of distinct data array elements that appear on the right hand side of assignment statements will generally be of the form $Am+B$. The number of essential loads is taken to be A. The number of essential stores is found similarly by examining the left-hand side of each statement and counting distinct array elements that appear on the left after they appear on the right side of an assignment statement. Note that the cache of the Alpha is write-through, thus there are no store misses.

We then derive the number of essential load misses from the number of essential loads by examining the sizes of the referenced data arrays and, from the Fortran Common block structure, deduce how they are placed with respect to one another. Then through loop index analysis, the referenced portion and stride pattern are determined for each array. Examples of determining the number of essential load misses are in sections 9.1.5 and 9.3.1.

4.0 Code scheduling with Cache Effects

In the area of fine-grain scheduling, not many compilers today take cache effects into consideration for a number of reasons.

1. Cache effects may differ from one platform to another, making the compilers that deal with them less portable even among the same architecture.

2. The code may become too complicated because reducing cache effects may require extensive use of unrolling and cyclic scheduling. These techniques also increase the number of registers needed. Also, some optimized code may not result in a performance increase due to unwanted side effects, such as register spilling.

In this paper, however, we attempt to hand-schedule the code to see whether reducing cache effects sufficiently to achieve the bound is possible. In this section, we address the issues of fine-grain scheduling while reducing cache effects. First, some background information about the model of the access-execute parallelism will be reviewed briefly, because the scheduling technique used in this paper is based on this concept. Next, the trailing edge effects will be introduced as one of the major types of cache effects being ignored by many compilers. Then the scheduling technique will be developed based on the idea of data prefetching.

4.1 Access-Execute Parallelism

If each iteration of the loop is independent of other iterations, the inputs for one iteration can be fetched from the cache system as the floating-point instructions of previous iterations are being processed.

To exploit the access-execute parallelism, the memory loads are issued well ahead of the operation instructions that require this data. In Mangione-Smith's thesis [11], this technique of pre-issuing memory loads is called slip. This technique is essential for efficient scheduling with cache misses, since issuing memory loads way before using the target data is required to reduce cache miss stall cycles to a minimum. While waiting for the data being fetched from the cache, the processor can run some other useful instructions. The scheduling method used in this paper is based on this technique.

4.2 Cache Misses: Leading-Edge and Trailing-Edge Effects

The leading-edge effect is the penalty for a single isolated miss reference. A subsequent references to the same line, which should be a hit, will suffer a trailing-edge penalty if the miss has not yet been fully serviced. The leading-edge effect of an essential load miss cannot be eliminated because it is simply the time required to bring the requested data into the cache. However, the trailing edge effects caused by a load miss can be eliminated by making sure that subsequent references to that cache line occur after the load miss is completely serviced. The bound therefore assumes that schedules will be optimized to eliminate trailing-edge effects and the hand-coding tries to achieve this.

4.3 Data Prefetching

The main idea behind minimizing the trailing edge effect is to prefetch one of the data words in the cache line as early as necessary to eliminate the trailing-edge effects for the other references to that line. Then the code can be scheduled so that the remaining data in the referenced cache line have been placed into the primary cache before they are referenced. However, as stated earlier, one problem associated with data prefetching is that the transfer time of one cache system may be different from another, making such compiler settings good only on that particular platform.

The rest of this paper will put the idea of prefetching data into practice. This requires extensive loop unrolling and cyclic scheduling. Theoretically, loops can be scheduled so that the trailing edge effects can also be reduced. However, some loops cannot be optimized practically. This paper also points out certain types of the loops which cannot be well-scheduled and why they cannot.

5.0 The Alpha Architecture and its Implementation: The DECchip 21064

The Alpha AXP architecture is a traditional RISC load-store architecture. This architecture was developed by Digital based on the goals of high performance and longevity. In view of these goals, the Alpha was designed as a full 64-bit architecture with emphasis on fast clock speed, multiple instruction issuing, and the capability of implementing multiple processors with shared memory [9][14]. A linear 64-bit virtual address space is adopted (no address segmentation). Addresses, integers, floating-point numbers, and character strings are all operated on as full 64-bit data.

The DECchip 21064 is an implementation of the Alpha architecture. In this paper, for simplicity, this DECchip 21064 will be called the Alpha chip. The Alpha chip has three independent functional units: the integer execution unit (Ebox), the floating point unit (Fbox), and the memory unit (Abox). Each unit can accept at most one instruction per cycle, however if code is properly scheduled, the issue unit (Ibox) of the Alpha chip can issue two instructions to two independent units in a single cycle, provided they are to different functional units. The Ibox issues instructions, maintains the pipeline, and performs all of the PC calculations. The Alpha also has on-chip instruction and data caches (Icache and Dcache). For the purposes of this paper, the following subsections describe the possible bottleneck units of the Alpha.

5.1 Issue unit

The primary function of the issue unit is to issue instructions to the Ebox, Abox, and Fbox. In order to provide those instructions, the Ibox also contains the prefetcher, PC pipeline, instruction translation buffer (ITB), abort logic, register conflict or dirty logic, and exception logic. Every instruction is 32 bits long. The Ibox decodes two instructions, aligned on a 64-bit boundary, in parallel. The IBox decides whether to issue both instructions by checking the following conditions:

1. the required resources must be available for both instructions, and

2. the dual issue rule must be satisfied.

When these conditions are not both satisfied, the first instruction is issued as soon as its resources become available. The second instruction is issued simultaneously, if the two conditions are both satisfied at that time, and, if not, then it is issued as soon as its resources are available after that. Then the issue unit proceeds to the next pair. The Ibox thus never issue instructions out of order and never issues the second instruction of a pair simultaneously with the first instruction of the next pair.

The Ibox may not be able to issue an instruction for one of two reasons. The first reason is due to a pipeline stall. A pipeline stall occurs when a valid instruction is ready but cannot proceed due to a resource conflict. It is the responsibility of the Ibox to insure that all resource conflicts are resolved before that instruction is issued. Once all its resource requirements are satisfied, an

instruction is issued and allowed to continue through the pipeline stages toward completion. After issuing, the instruction cannot be held in a given pipe stage and cannot be stopped except for an abort condition. Abort conditions, which are not controlled by the Ibox, includes exceptions and non-exceptions such as branch mispredictions, subroutine call/return mispredictions and I-cache misses.

The second type of non-issue arises when there is no valid instruction in the pipeline to issue. This situation is caused by the abort conditions listed above, which cause pipeline bubbles in the instruction fetch pipeline. In addition, a single pipeline bubble is always produced whenever a branch instruction is predicted to be taken, as in subroutine calls and returns. Pipeline bubbles are reduced directly by the hardware through bubble squashing, but can also be effectively minimized through careful code scheduling.

The pipeline separates instruction processing into four initial stages in which stalls may occur, follow by various numbers of pipeline stages, depending on the functional unit considered, in which stalls never occur.

5.2 Floating-point unit

The Alpha chip has a pipelined functional unit named the Fbox which is capable of executing both DEC and IEEE-standard floating point instructions. The Fbox contains a 32-entry by 64-bit floating point register file and a user accessible control register. It can accept an instruction every cycle, with the exception of floating point divide instructions, which cannot be pipelined. The latency for data dependent, non-divide instructions is six cycles. Bypass mechanisms are provided to allow the issue of instructions which are dependent on prior results while those results are written to the register file. This bypass implementation saves 2 clock cycles, but for whatever reason the mechanisms do not apply within a single pipeline of the Fbox itself. (e.g. an add instruction result is never bypassed to a second add instruction) Thus the latency for a floating-point store after a floating-point operation is 4 clock cycles. Floating-point operate instructions progress through a ten stage pipeline, of which the last six stages never stall.

5.3 Memory Unit

The Abox functional unit consists of the address generator, data translation buffer (DTB), load silo, and write buffer. The bus interface unit (BIU), which is also in the Abox, communicates with external caches as well as both the on-chip instruction cache and the on-chip data cache.

5.3.1 Bus Interface Unit

The bus interface unit (BIU) controls the interface to the external bus on the Alpha chip pins. It responds to three classes of CPU generated requests: D-cache fills, I-cache fills, and write buffer requests. The bus interface unit resolves simultaneous internal requests using a fixed priority scheme. D-cache fill requests are given highest priority, followed by I-cache fill requests. Write buffer requests have the lowest priority, except when all the entries in the write buffer are full. In that case, store instructions which send data to the write buffer are treated similarly to load miss requests. The BIU contains logic to service internal cache fill requests and writes from the write buffer by directly accessing an external cache. With help from external logic, the BIU also services reads and writes that do not hit in the external cache.

5.3.2 Primary Data Cache and its Behavior

Because the Alpha is a 64-bit architecture, each data reference refers to 64 bits, or 8 bytes. After the cycle in which a load instruction is issued, it takes two more clock cycles for the memory unit to detect whether the load is a hit or a miss. The data requested by hits can be used after the following cycle. For a load miss, the nominal access time starting from the detection of a load miss to the time the data is available in the target register is 8 clock cycles. That is, the requested data is loaded into the target register at the end of the 11^{th} cycle after a request is issued. During the miss penalty cycles when the D-cache is kept busy requesting the data, the issue unit will stall the issue of any subsequent load or store instructions. Also, any instruction that uses the requested data during the miss penalty time will also be frozen until the data is loaded into the target register.

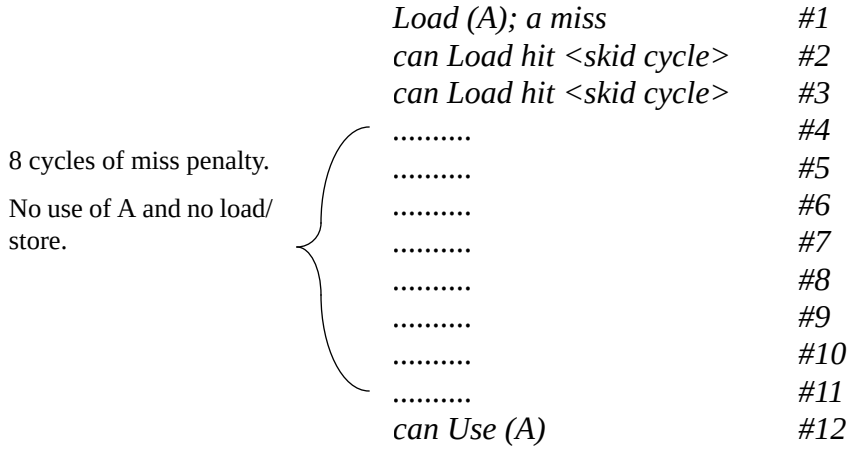

FIGURE 2. A Load Miss

5.3.3 Load Silo

Since a load miss is detected during the third cycle of a load instruction, there may be two instructions in the Abox pipeline behind a load miss instruction. These two instructions are handled as follows:

Load hits are allowed to complete.

Load misses are placed in a silo and re-issued after the first load miss completes. Any references to the same line as the first load miss are treated as misses.

Stores are placed in a silo also, and re-issued after the first load miss completes. Store data is updated in the cache if the referenced line is already present in the cache. Since the D-cache is write through, there are no store misses.

All the instructions in the silos are re-issued in the order received immediately after the load miss completes.

5.3.4 Pending-fill Latch

The entire cache line referenced by a load miss, including the referenced data and the remaining data in the cache line, will be placed into the pending-fill latch as it arrives from the secondary cache. Once the data transfer to the latch completes and the conditions to update the cache are satisfied, the entire cache line is transferred from the pending-fill latch into the cache.

The completion time of the fill depends on the speed of the secondary cache and the interface bandwidth. The word referenced by a miss goes to the processor through a bypass mechanism before the entire line is in the latch.

5.3.5 Secondary Cache

The speed of the secondary cache is a very important factor in the performance of the entire system. How quickly the secondary cache transfers the data is directly related to how fast the primary cache can get its requested cache line filled.

DEC provides its system with a secondary cache which has an access time of 5 cycles. There are interfaces of 64 bits wide and 128 bits wide. Only DEC 3000 Model 300 and 300L use a 64-bit interface, and all the others use a 128-bit interface. The 64-bit interface requires four accesses to fill the latch with a 32 byte line, so that 26 (=3+8+5+5+5) clock cycles are needed from the time the load instruction is issued to the time that the complete cache line has been brought into the primary cache. The 128-bit interface needs only two accesses, which takes 16 (=3+8+5) clock cycles total. The model under current study has the 128-bit interface, thus the rest of the discussion about scheduling techniques will use the 16-cycle cache fill time.

The first access, starting from the time after the load instruction is issued and taking 10 clock cycles total, brings the referenced half of the cache line into the pending-fill latch. A second access cannot be completed until the end of additional 5 clock cycles required to completely fill the latch. Then the data in that cache line is transferred from the pending-fill latch into the primary cache. However, on the last (16^{th}) cycle, if the issued instruction is a load instruction to some other line, this cache line data in the filled latch will *not* be transferred into the corresponding cache line of the primary cache. Until the next clock during which no load instruction is issued, the cache line data will simply stay in the filled latch.

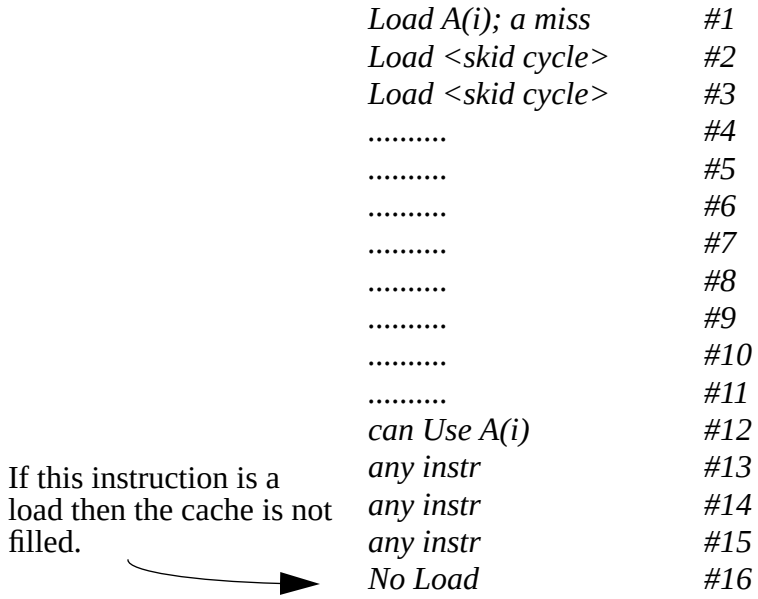

FIGURE 3. Restriction on Filled Latch

5.3.6 Write Buffer and its behavior

The write policy used by Alpha is write-through. Every time a store instruction is issued, the data to be stored is sent to the write buffer, and the primary cache is updated if the referenced line is present. The write buffer has four entries, each of 32 bytes. The issue unit dispatches a store instruction regardless of the limited capacity of the write buffer. Data in the same cache line would get transferred to the same entry in the write buffer as long as the previous data in that entry has not yet been flushed. As other different data elements may fill the rest of the entries, the write buffer dumps the data that have been accumulated in an entry to the secondary cache once another entry starts to get filled.

If all four entries in the write buffer are completely filled and a store instruction is issued, that instruction will be treated like a load miss, where the following two instructions are treated as being in the skid cycles. If half or less of an entry (16 bytes) has been filled when the entry is signaled to get flushed from the buffer, the time to flush that entry only takes 10 clock cycles. This is known as half-entry write. If an entry is ready to get flushed and more than half of the entry has been filled, 15 clock cycles are required for this full-entry write.

When the write buffer attempts to send one of its entries off to the secondary cache, it will not be serviced if one or both of the two instructions following the store instruction are load instructions. If either of those load instructions is a miss, the miss will be serviced first while the data remains in the write buffer. This deferral is a consequence of the BIU's fixed priority scheme for simultaneous requests, where D-cache fill requests have highest priority.

6.0 Performance Bound Model of the DEC Alpha

In this section, the performance bound model which was described in section 2.1 is applied to the DEC Alpha architecture (more specifically, to the DECchip 21064).

6.1 Issue unit

The issue unit is capable of dispatching two instructions per clock cycle, provided they are to different execution units. Let l be the number of essential floating-point loads and s be the number of essential floating-point stores, which were defined in section 3.3. Also, let f_m be the number of essential floating-point multiply instructions and f_a be the number of essential floating-point add instructions. The number of clock cycles needed to dispatch all instructions in a loop iteration is at least

$$t_i = max(l + s, f_m + f_a) \tag{2}$$

6.2 Floating-point unit

The pipelined floating-point unit is capable of accepting an instruction every cycle, with the exception of floating-point divide instructions which are not pipelined. There are no multiply-add triad instructions. Therefore the number of clock cycles required to execute floating-point operations in a loop iteration is at least

$$t_f = f_m + f_a \tag{3}$$

6.3 Memory port unit

The memory unit can also accept a load instruction or a store instruction per cycle. Without any memory stalls, the number of clock cycles required to execute load and store instructions is at least

$$t_m = l + s \tag{4}$$

However, due to the small size of the primary data cache of the Alpha chip, the effects of cache misses should be considered. In addition, store instructions may cause the write buffer to tie up the interface port to the secondary cache. So the new bound model must take into account how the secondary cache deals with the cache misses as well as the write updates.

Let m_l be the number of essential load misses as defined in section 3.3. To illustrate, if a loop code satisfies our normal assumption, the loop body can be unrolled four times, and load references are in unit stride, then the number of essential load misses becomes 1/4 per data array reference. Also, let s_{ft} be the number of full-entry write throughs per loop iteration. This full-store takes 15 clock cycles in the Alpha system. The half-store takes 10 clock cycles, which is s_{ht}, the number of half-entry write throughs per loop iteration.

With the effects of load misses, the bound equation for the memory unit becomes

$$t_m = 8 \times m_l + max\,(\,(l + s)\,,\, 3 \times m_l) \tag{5}$$

The value 8 in the above equation represents the miss penalty time. The value 3 is the load delayed time plus 1. Since load and store hits are allowed in the load delayed cycles, the max term counts one clock for each essential memory reference or three clocks per essential load miss, whichever is greater.

To take the write buffer effects into account, another term is added to the max expression.

$$t_m = 8 \times m_l + max\,(\,(l + s)\,,\, 3 \times m_l,\ (15 \times s_{ft} + 10 \times s_{ht})\,) \tag{6}$$

This equation can be rewritten as three formulas representing three cases; t_m is then the maximum of these formulas. For the first case, t_{m1} is

$$t_{m1} = 8 \times m_l + l + s \tag{7}$$

This is the case all delay slots of load misses are filled with essential loads and stores, and the write buffer does not contribute effects on performance.

For the second case, t_{m2}, is

$$t_{m2} = 11 \times m_l \tag{8}$$

This is the case when load misses simply dominate. It happens when all load hits and all stores can be placed within the delay slots of essential load misses. From the time that a load miss instruction is issued to the time the target data is loaded is 11 clock cycles. All load hit and store instructions can be serviced within the skid sections, and other types of instructions can be issued without interfering the memory unit.

For the third case, t_{m3}, is

$$t_{m3} = 8 \times m_l + 15 \times s_{ft} + 10 \times s_{ht} \tag{9}$$

This is the case when the secondary cache is kept continuously busy serving both the primary cache for load misses and the write buffer for write throughs. All instructions not related to the memory unit can be served while the secondary cache is busy.

6.3.1 Effects of the secondary cache

The access time of the cache depends on the speed of the secondary cache: how fast can the secondary cache provide the data? In the Alpha-based workstations, the miss penalty is typically 8 clock cycles. However, when the miss penalty cycles increase, the memory unit term, t_m, increases. Therefore, for those machines with high miss penalty, or no secondary cache at all, such as in the case of the Alpha processor on the T3D, t_m would become extremely large, thus making the memory unit the bottleneck unit for most applications.

6.3.2 Effects of the write buffer

The effects of the write buffer related to the secondary cache utilization. Increasing latency time for flushing data from the buffer may cause the processor to stall longer, causing the memory unit time to be larger. Therefore, a fast write buffer flush latency is an important enhancement for the design of the cache system.

6.4 Loop carried dependence unit

The loop carried dependence pseudo-unit is used to model a potential performance bottleneck for loops with recursion. Those loops can be identified when a result of one iteration depends on the corresponding result of a previous iteration. Whenever there is such a cycle in the dependence graph of the floating point arithmetic operations, the time is computed as the sum of the latencies of the operations in one tour of the cycle divided by the number of iterations in the cycle. Otherwise, the dependence time is zero. The latency of an operation is related to pipeline depth and is computed as the minimum number of clocks between issuing that operation and issuing a succeeding operation that uses its result as an operand. The latency of any floating-point operation, except divide, is 6 clock cycles.

7.0 Scheduling Methods for the DEC Alpha

In the following sections, we assume that the memory unit is the bottleneck. This section shows how to transform the old scheduling method to schedule the code with cache effects in mind. The idea of the new scheduling method came from the slip technique, where memory loads are issued long before the requested data are used. In this research study, however, the load miss instructions are issued as early as possible. We developed a technique to first distinguish essential load misses, and then to apply cyclic scheduling which was described in section 2.2, so that the miss effects are reduced to the minimum.

Definition 3: Iteration Interval

The iteration interval (II) of a compiled program kernel is the number of clock cycles that occur between issuing the first instruction for an iteration of the loop, and issuing the first instruction for the next iteration.

Definition 4: Instruction Templates

The template for a machine instruction indicates which hardware resources are needed to execute that instruction, and when (relative to instruction issue) those resources are reserved exclusively for this instruction.

For the Alpha, an instruction template of a load operation that hits in the cache look as follows (ignoring the first four pipeline stages of the issue unit since these stages of all instructions are the same).

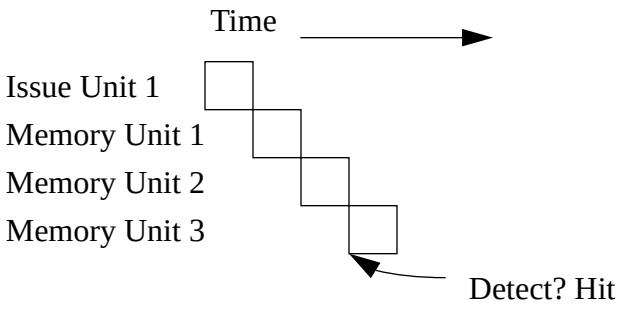

FIGURE 4. Instruction Template for a Load Hit

However, for a load miss, the template looks as follows:

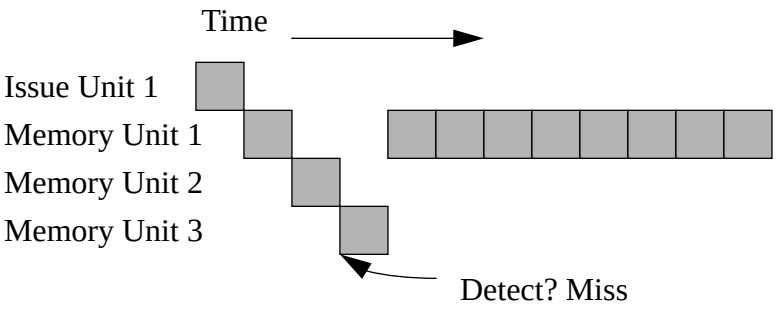

FIGURE 5. Instruction Template for a Load Miss

The 8 penalty cycles in memory unit 1 block out possible instructions which might attempt to use the memory system. In other words, no load or store instructions which deal with the memory system are allowed to be issued during the penalty cycles.

Definition 5: Modulo Reservation Table

A modulo reservation table, in which instruction templates are placed, contains II columns and one row for each resource used by any of the templates.

7.1 Scheduling with load misses only

To illustrate load scheduling, consider a loop contains only load instructions, the lower bound cycles per loop iteration is the number of essential load misses times the number of cycles in a load template plus the extra load instructions, i.e. load hits that do not fit in the skid slots. The method to hand-schedule such code is as follows:

1. Determine the number of essential load misses.

2. For each essential load miss, place a load miss template into the modulo reservation table.

3. Start inserting load hits into the skid section, i.e. where the load delay cycles occur.

4. Place the remaining load hits, if any, into the modulo reservation table.

From intuition, load hits should be placed as early as possible in the code, so that the arithmetic operations done on the data can be issued as soon as possible.

5. Generate actual code and check for trailing edge effects

To eliminate trailing edge effects,

5.1 Try reordering load hit instructions.

5.2 Try reordering the templates of all types in the modulo reservation table.

5.3 Try to prefetch data in earlier iterations, i.e. relabel some placed load templates to refer instead to the corresponding load for a later iteration.

7.2 Example: Two essential load misses (LFK 3)

Suppose that there are two references to two different arrays of data, where the size of each of the data arrays is equal to or larger than the size of the primary cache (8 KB). Suppose there are enough registers and iteration in independence so that the loop can be unrolled four times. With a well structured code, for each load miss, two load hits can be put into the skid cycles.

Step 1: We found that there are two essential load misses.

Step 2: We place them into the modulo reservation table.

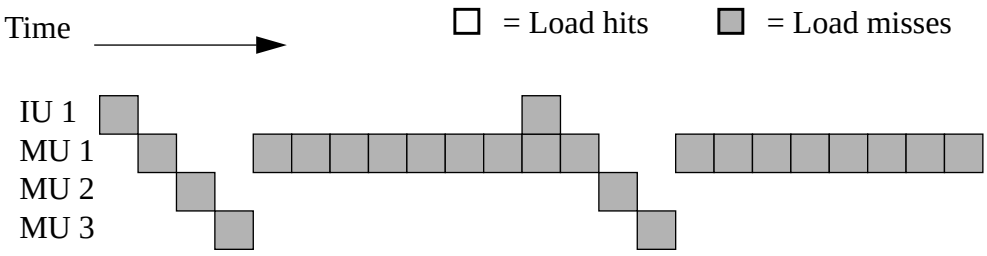

FIGURE 6. MRT of Two Misses

Step 3: We place load hits into the skid areas

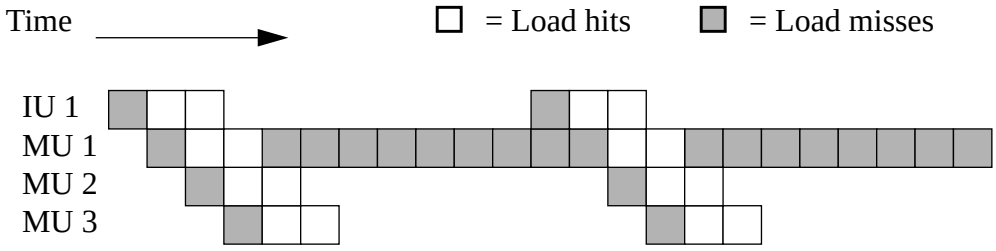

FIGURE 7. MRT of Two Misses and Four Hits

Step 4: In order to allow the arithmetic operations to be issued as soon as possible, the remaining load instructions should be placed before the templates that have already been placed. This placement results in a tight schedule in which MU1 is continuously busy. Note that in a modulo reservation table, the last use of MU1 can be wrapped around and placed in the first time slot of the table, e.g. this would be done for Fig. 7 if II =22. However, in some cases, there may not be

enough registers to permit this placement. Then a different order of instruction templates needs to be tried.

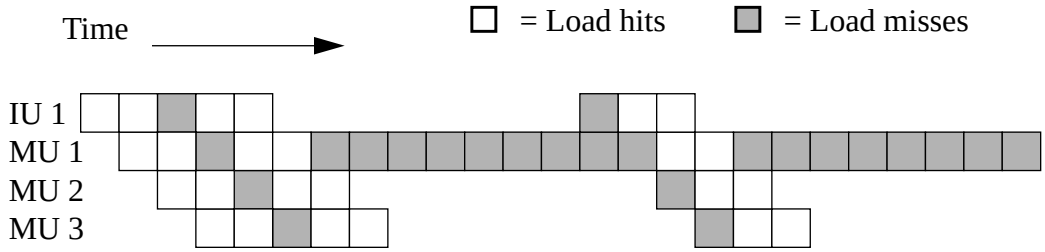

FIGURE 8. MRT of Two Misses and Six Hits

7.3 Hand-coded Schedule

After the instruction templates are placed, the next step is to write the actual assembly level code from the modulo reservation table. Since load hits and load misses are clearly distinguished by using different instruction templates in the modulo reservation table, the actual code schedule must also follow this pattern.

How can we be sure in a loop iteration that certain load instructions are hits and certain instructions are misses? This is an important point since by knowing which load instructions would produce hits and which would cause misses, cache behavior can become predictable as described above.

Assume that the data array elements to be referenced are not in the primary cache yet, but reside in the secondary cache. To implement data prefetching without special software and hardware support, instructions which prefetch the target data from the secondary cache are simply load misses. After the cache line associated with the target data is brought into the primary cache, any references to the remaining data in the same cache line are load hits. With a scheduling approach that prefetches one element per cache line and references others only after the prefetch is known to be completed, load instructions in a loop iterations can be distinguished as hits or misses and thus become predictable.

As mentioned earlier, the memory system that DEC provides requires 16 clock cycles from the time the load miss instruction is issued to the time the referenced cache line has been brought into the secondary cache. This implies that any references to the cache line being transferred must be statically scheduled at least 16 clock cycles after the prefetch miss in order to avoid incurring trailing-edge effects.

To illustrate, we derive the assembly level code from the modulo reservation table in Fig. 8. The assumption was that the size of the data arrays A and B is larger than the size of the cache that the iteration independence and the number of registers are sufficient to permit unrolling 4 times. Suppose that references to array A in a loop iteration consist of A(i), A(i+1), A(i+2), and A(i+3), and i is incremented by 4 per iteration. Since the cache line size is 4 words, out of the four references there is only one essential load miss. To prefetch data, A(i+3) is the best candidate because such a reference brings in the next cache line regardless of the array alignment with respect to cache line boundaries. Prefetching the current iteration's A(i+3) data one iteration earlier, whose index i is 4 less than the current index i, requires a reference to A(i+7) in each iteration. Data array B is handled likewise.

So, the code from the modulo reservation table for two data references may look like the following.

```
Load A(i); hit        #1
Load B(i); hit        #2
Load A(i+7); miss     #3
Load A(i+1); hit      #4
Load B(i+1); hit      #5
......                #6
......                .....
......                .....
......                #12
......                #13
Load B(i+7); miss     #14
Load A(i+2); hit      #15
Load B(i+2); hit      #16
......                ......
......                #24
```

FIGURE 9. Pseudo-code for Two Misses and Six Hits

7.3.1 Trailing-edge problem

The MII for the above code is 24 clock cycles and, as stated above, the transfer time for any cache line is 16 clock cycles. The above code fits the modulo reservation table but may produce the trailing edge effects when referencing the data array B. This is because the cache line alignment is unknown to compilers, and thus it is possible that B(i+3) data is in the same cache line as B(i+1) and B(i). In that case, B(i) and B(i+1) may be considered by the Alpha processor as load misses, thus resulting in trailing-edge effects.

7.3.2 Reorder the Hit References

Sometimes, changing the order of load hits may eliminate the trailing edge effects. In this example, we try to reschedule the “hits” for data array B after we are sure that the corresponding cache line has been brought in. However, in the schedule below, B(i) is still issued within 16 cycles after B(i+7) in the previous iteration was issued, so if B(i) and B(i+3) are in the same cache line, B(i) will still experience trailing-edge effects. So in this case, reordering the load hits does not eliminate all the unwanted effects. Note that some improvement has been achieved since if B(i+1), B(i+2) and B(i+3) are in the same cache line, but B(i) is not, there are no trailing-edge effects.

7.3.3 Reorder the Instruction Templates

If reordering the load hits in a fixed modulo reservation table cannot eliminate the possibility of trailing-edge effects, then the instruction templates may need to be reordered in the modulo reservation table. The following code shows that two earliest load hits in the loop iteration have been moved in between the two load misses.

This code eliminates all possible trailing-edge effects. However, because two hits are placed in between the two load miss templates, it is harder to control the schedule, e.g. to pair up

On the 6^{th} cycle, the cache line containing B(i+3) data, which is B(i+7) of the previous iteration, is in the primary cache.

B(i), B(i+1) and B(i+3) might be in the same cache line. Then the B(i) reference on the 2^{nd} cycle and the B(i+1) reference on the 5th cycle would be treated by Alpha as misses, not hits.

The cache line containing B(i+7) will be in the primary cache on the $14+16 = 30^{th}$ cycle, which is $30-24 = 6^{th}$ clock cycle in the next iteration.

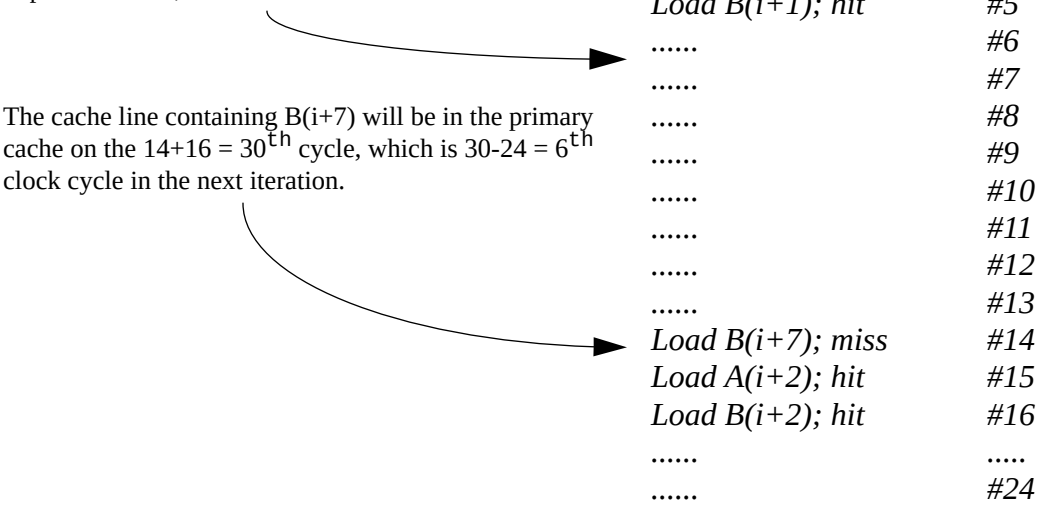

FIGURE 10. Code with Trailing-edge Problem

Again, B(i) and B(i+3) may be in the same cache line. Reference to B(i) may be a miss, not a hit.

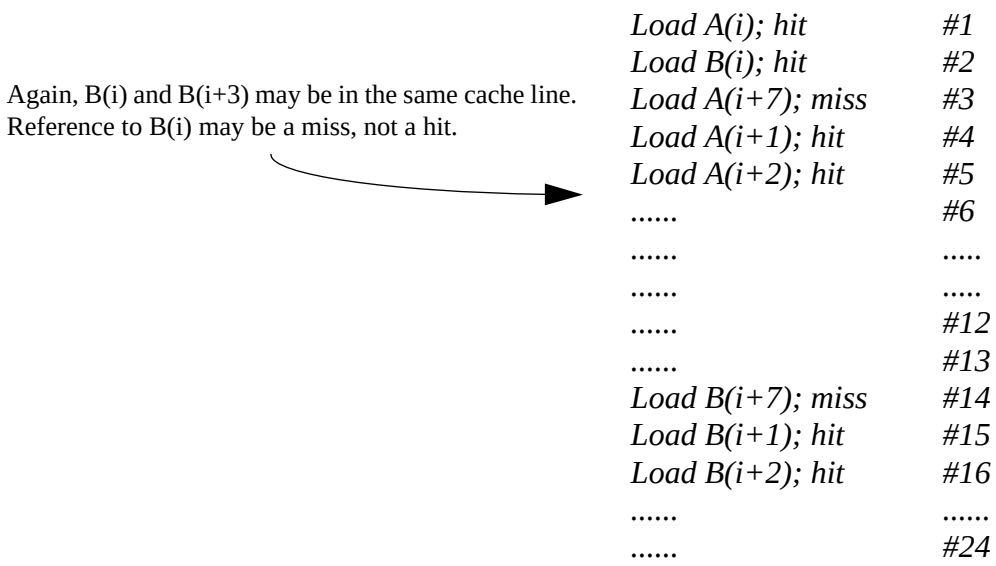

FIGURE 11. Code after Reordering Hit References

load instructions with other dependent instructions. Therefore whether this scheduling structure can be used effectively depends on the complexity of the code.

```
Load A(i+7); miss      #1
Load A(i); hit         #2
Load A(i+1); hit       #3
......                 #4
......                 .....
......                 #11
Load B(i); hit         #12
Load B(i+1); hit       #13
Load B(i+7); miss      #14
Load A(i+2); hit       #15
Load B(i+2); hit       #16
.......                .....
.......                #24
```

FIGURE 12. Code after Reordering Instruction Templates

7.3.4 Referencing Data Earlier

Finally, we can stick with the original modulo reservation table but instead of referencing data in the previous iteration, we may try to reference data two iterations earlier. This may eliminate the trailing-edge effects if there are enough registers to use. Notice that A(i+7) is now changed to A(i+11). Data array B is handled likewise.

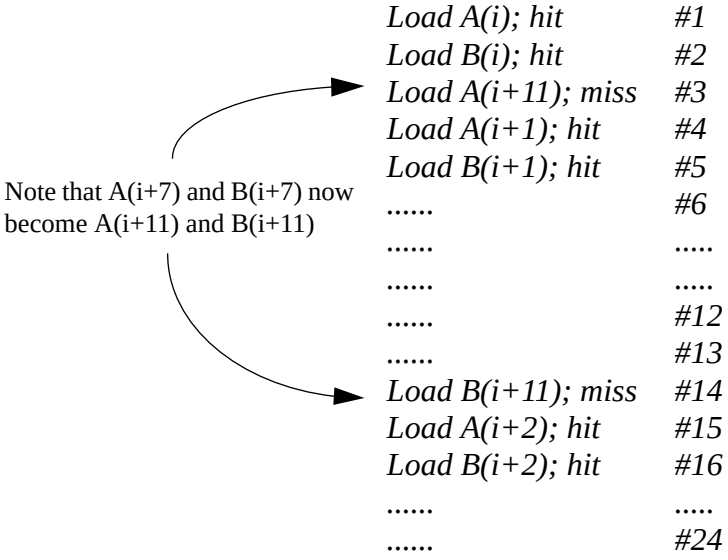

FIGURE 13. Code with References to Data Two Iterations Earlier

One drawback is that additional instructions (fmov) are needed to save the pre-loaded values in other registers. However, this method of referencing two iterations earlier does generate consistent results. Therefore, it is generally used whenever possible throughout this paper.

7.4 Scheduling with load misses and stores

Loops with both load and store instructions are scheduled similarly to the method for loops with load misses only. However, scheduling with store instructions can be unpredictable

due to the way that the Alpha was implemented. How the data is aligned with respect to an entry in the write buffer is unknown, which is the major reason for the unpredictable behavior of the store instructions. However if the number of half stores and the number of full stores can be determined, some scheduled code may still approach the performance bound.

7.4.1 Comments on the Store instructions

As stated previously, a filled entry in the write buffer starts to be flushed when another empty entry starts to get filled. The prefetching method for scheduling around the cache misses can also be applied to the store instructions, but it may not work well because the alignment of the cache boundaries with respect to the write buffer entries is not known to the compiler. For example, with each buffer entry able to hold four words, the compiler doesn't know whether four consecutive stores are two half stores (two words in each of the two half entries), a single full store (four words in one buffer entry), or a half store and a full store (one word in a half entry and three words in a full entry).

Furthermore, issuing a store instruction apart from the other store instructions will *not* have the same effects as pre-loading data. Instead of a pre-load instruction, if a loop has a post-store instruction (storing data long after the iteration completes), that instruction will cause *at least* one entry in the write buffer to flush. If the cache line boundaries which are aligned to the write buffer entries, are known to the compiler, there could be only one updated buffer entry. However, since the compiler has no idea where the cache boundaries are, it is very likely that two buffer entries are ready to get flushed at some times. This effect can cause all four entries in the write buffer to become filled up over a few iterations and thus degrade performance greatly.

There is no solution yet for this problem. There is very little that the compiler can do if the cache line boundaries are unknown with respect to the data array elements. Therefore, in this paper, the store instructions are arbitrarily placed near the end of an iteration. Regardless of where store instructions are placed, some performance losses may be expected. A few LFK loops that experience this kind of loss will be mentioned in section 9.0.

8.0 Characterization of the LFK loops

The first twelve LFK loops are used in this study. Common characteristics of the loops can be distinguished by the existence of essential load misses and store instructions in the code. We characterize these loops in the following subsections. The LFK workload and the bound without

cache effects are summarized in Table 1. The bound with cache effects and all the related vari-

	f_a	f_m	l	s	t_i	t_f	t_m	t_d	t_l	CPF
1	2	3	2	1	5	5	3	0	5	1.00
2	2	2	4	1	5	4	5	0	5	1.25
3	1	1	2	0	2	2	2	0	2	1.00
4	1	1	2	0	2	2	2	0	2	1.00
5	1	1	2	1	3	2	3	12	12	6.00
6	1	1	2	0	2	2	2	0	2	1.00
7	8	8	3	1	16	16	4	0	16	1.00
8	21	15	9	6	36	36	15	0	36	1.00
9	9	8	10	1	17	17	11	0	17	1.00
10	9	0	10	10	20	9	20	0	20	2.22
11	1	0	1	1	2	1	2	6	6	6.00
12	1	0	1	1	2	1	2	0	2	2.00

TABLE 1. Workload and CPF bound

ables are summarized in Table 2.

	m_l	s_{ft}	s_{ht}	t_m'	t_l'	CPF'
1	0.50	0.25	0.00	7.75	7.75	1.55
2	1.00	0.25	0.00	11.75	11.75	2.94
3	0.50	0.00	0.00	6.00	6.00	3.00
4	0.67	0.00	0.00	6.80	6.80	3.68
5	0.50	0.25	0.00	7.75	12.00	6.00
6	1.25	0.00	0.00	13.75	13.75	6.88
7	0.75	0.25	0.00	10.00	16.00	1.00
8	3.00	0.75	3.00	65.25	65.25	1.81
9	4.00	0.00	1.00	44.00	44.00	2.59
10	3.00	2.00	1.00	64.00	64.00	7.11
11	0.00	0.25	0.00	3.75	6.00	6.00
12	0.00	0.25	0.00	3.75	3.75	3.75

TABLE 2. Essential load misses, write throughs, and new CPF bound

8.1 Loops with No essential load misses

An example of this type of loop is LFK12, where there is only one load instruction that references a data array, and the size of that array is less than the size of the cache. This type of loop depends on the number of store instructions, because any floating-point operations can run independently while the write buffer is flushing the data into the secondary memory system.

8.2 Loops with No store instructions

This type of loop is also easy to hand-code. Since there are no store instructions, the write buffer does not interfere with the load activity between the primary cache and the secondary cache. These loops are LFK3, LFK4, LFK6.

8.3 Loops with loop-dependency

In this type of loop each iteration requires a result from the previous iteration. These loops are LFK5 and LFK11, and have non-zero t_d.

8.4 Loops with essential load misses and store instructions

These loops include LFK1, LFK2, LFK9, and LFK10. However, LFK 2 also contains different data sizes for different iterations, so the bound model may be inaccurate in that particular case. For the methods used in this paper, irregular data sets make the effects of data references unpredictable.

8.5 Loops with not enough registers

LFK 7 and LFK 8 are quite complex and require extensive use of registers. Cyclic scheduling would cause register spilling problems.

9.0 Results from scheduling the LFK loops

The following graph shows the results of optimizing the LFK loops with the scheduling methods proposed in this paper.

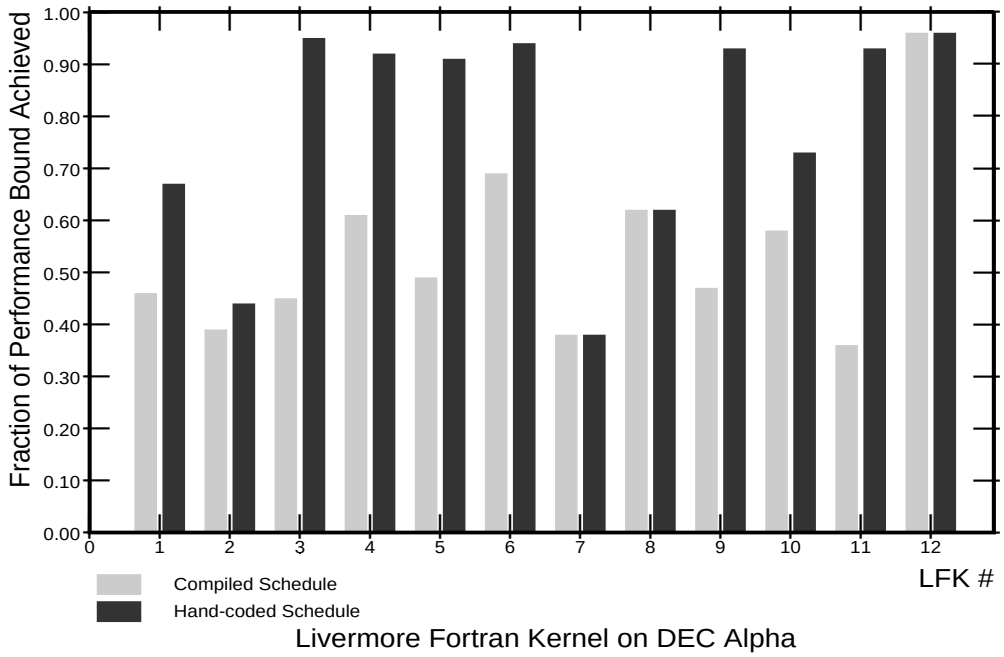

FIGURE 14. Code Improvements through hand-scheduling

	Compiled	Hand code	Bound CPF	% bound
1	3.38	2.30	1.55	67
2	7.05	6.27	2.94	47
3	6.63	3.17	3.00	95
4	5.61	4.00	3.68	92
5	12.34	6.61	6.00	91
6	10.02	7.30	6.88	94
7	2.65	2.65	1.00	37
8	2.93	2.93	1.81	62
9	5.54	2.77	2.59	93
10	12.16	9.73	7.11	73
11	16.72	6.42	6.00	93
12	3.89	3.89	3.75	96

TABLE 3. Compiled Code vs. Hand Code Performance in CPF

9.1 Approaching Performance Bounds

With hand-coding seven out of the twelve loops, LFK 3, LFK 4, LFK 5, LFK 6, LFK 9, LFK 11, and LFK 12 achieve at least 91% of the performance bound. For the compiled code, only five loops, LFK 4, LFK 5, LFK 8, LFK 10, and LFK 12 achieve 50% or more of the performance bound.

9.1.1 LFK 12

The following code is the inner loop of LFK 12:

```
      DO 12 k = 1,n
12        X(k)= Y(k+1) - Y(k)
```

Notice that there is only one essential memory reference to the data array Y. Since Y is smaller than the size of the cache, there are no essential load misses. There is only one full store per four iterations. There is thus a lot of room to schedule other operations during the cycles caused by the write buffer flushes. In fact, there is so much leeway that the compiler has already done a good job of scheduling. The CPF bound of the LFK 12 is 3.75, and the compiled code runs at 3.89 CPF, which is 96% of the performance bound.

9.1.2 LFK 3 and LFK 9

LFK 3 have two essential load misses per cache line reference and LFK 9 have four essential load misses per iteration. These loops require hand-scheduling with some trial and error. But the performance turn out to be quite impressive. LFK 3 hits 96% of the performance bound and LFK 9 reaches 93%.

9.1.3 LFK 5 and LFK 11

The performance of both LFK 5 and LFK 11 is limited by loop-carried dependence. There is no need to unroll the loop or to use the cyclic scheduling technique. Thus these loops are fairly easy to optimize by simply making sure that the dependent instructions follow one after another.

The compiled performance of LFK 5 is 12.34 CPF. After hand-scheduling, the performance is improved to 6.61 CPF, which is 91% of the performance bound. For LFK 11, the performance is improved from 16.72 CPF to 6.42 CPF, reaching 93% of the performance bound.

9.1.4 LFK 6

LFK 6 contains 2 essential load instructions. The following code is the inner loop:

```
      DO 6 i= 2,n
      DO 6 k= 1,i-1
           W(i)= W(i) + B(i,k) * W(i-k)
6     CONTINUE
```

As for LFK 6, it appears at first glance to have two essential load misses per iteration, assuming it is unrolled 4 times. However, the difference is that B(i,k) is a two-dimensional data array and k is the inner loop index. Therefore, it experiences one essential load miss per load B instruction, not one per four load B instructions. This loop is, however, easy to schedule because it has much fewer floating-point operation cycles than the cycles caused by the essential load misses. The compiled performance, 10.02 CPF, is improved to 7.30 CPF through hand-scheduling, reaching 94% of the performance bound.

9.1.5 LFK 4

LFK 4 is a little tricky, not in scheduling the code, but in counting the number of essential load misses. The code is as follows:

```
      DO 444 k= 7,1001,m
           lw= k-6
           temp= X(k-1)
           DO 4 j= 5,n,5
                temp = temp - XZ(lw)*Y(j)
4               lw= lw+1
           X(k-1)= Y(5)*temp
444   CONTINUE
```

A rough estimate of the data conflicts is as follows. The variable n is set at 1001. At the beginning of each outer-loop iteration, the variable lw takes on the value of 1, 498, and 995, and is incremented by 1 for each inner-loop iteration. The variable j is incremented by 5. Therefore the inner-loop is run for 200 iterations. The areas where the data arrays certainly do not overlap are from XZ(200) to XZ(498) and from XZ(698) to XZ(995). That is about 595 data words where hits occur for certain, which is about 58% of the referenced array. Roughly assuming that load misses are about 42%, the number of essential load misses is calculated as 0.25 (XZ, assuming that we unroll 4 times) plus 42% times 1 (Y with no unrolling), which turns out to be 0.67. If the bound is calculated with m_l as 0.67, the performance of the hand-coded schedule reaches about 92% of the bound, which implies that this is not a bad estimate for m_l.

This is an example of where the percent of data conflicts is important in calculating the number of essential load misses. In this case, it cannot be assumed that loads are conflict misses anymore. The performance bound is calculated to be 3.68, and the performance is improved from 5.61 CPF to 4.00 CPF, which is 92% of the performance bound.

9.2 Moderate Improvement

The loops, LFK 1 and LFK 10, have only moderate improvement. Some performance losses may be due to the unpredictable write buffer effects.

9.2.1 LFK 1

LFK 1 contains five floating-point operations but with two loads and a store. There is not much leeway to fit those instructions tightly. Much time has spent on improving this loop and creating very complicated schedules, but result in some moderate improvement. The performance gets to 67% of the performance bound, improved from 3.38 CPF to 2.30 CPF.

9.2.2 LFK 10

LFK 10 is a loop with 10 essential load instructions and 10 essential store instructions. Assuming an arbitrary data alignment, there are 3 essential load misses, 2 full stores, and 1 half store. LFK 10 also has 9 essential floating point operations. The loop is not unrolled because, first, the Alpha doesn't have enough registers, and second, the penalty cycles caused by the load and store instructions are long enough to compensate for the cycles caused by the dependence of the floating-point instructions. This loop stays around 70% of the bound even with different ordering of the store instructions. As previously stated, store instructions cause unpredictable events. However, the performance still reaches 73% of the performance bound, jumping from 12.16 CPF to 9.73 CPF.

9.3 Small Improvement

LFK 2 references different amounts of data sizes from execution of an inner loop to another. The code is small enough to apply all the scheduling techniques. However, because of the different iteration intervals and irregular data sizes, the iterations are absolutely unpredictable.

9.3.1 LFK 2

The following is the loop code for LFK 2:

```
      II=n
      IPNTP= 0
222   IPNT= IPNTP
      IPNTP= IPNTP+II
      II= II/2
      i= IPNTP
      DO 2 k= IPNT+2,IPNTP,2
            i= i+1
2           X(i)= X(k) - V(k)*X(k-1) -V(k+1)*X(k+1)
      IF (II.GT.1) GO TO 222
```

When n is set at 101, the inner loop is repeated 6 times, with the loop executing for 50, 24, 12, 6, 3, and 1 times iterations, respectively, for the six executions. These executions of the inner loop thus reference100, 48, 24, 12, 6, 2 words of each data array, respectively. The following diagram shows the number of referenced data elements for one data array, which sums up to 194 words.

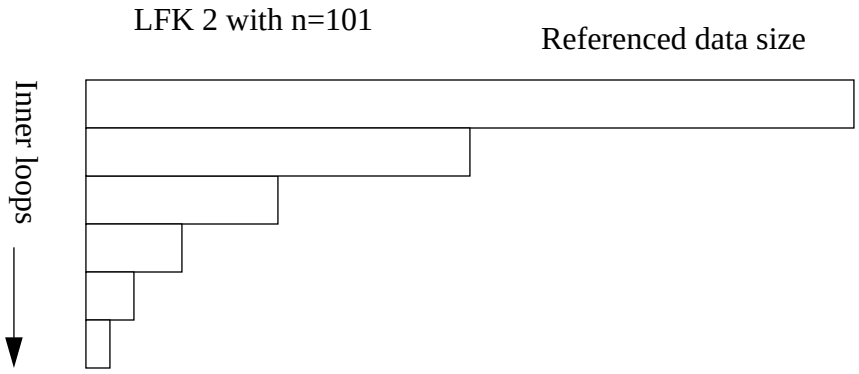

FIGURE 15. Referenced Array Elements for LFK 2

The following diagram shows how the data arrays are placed in the 1024 of words primary cache. Assume that the first element of array V starts at the beginning of a line that maps to the first line of the cache, and that arrays w and x follow consecutively in virtual memory space.

Clearly, each iteration of the inner loop is unpredictable. There is no code schedule which could generate steady state inner loop. Therefore, the performance bound model with cache effects can not be applied in this case. Using cyclic scheduling with unrolling shows a little improvement, but the references are still unpredictable. A small improvement from 7.05 CPF to 6.27 CPF only, barely reaching 47% of the performance bound.

9.4 No improvement

No scheduling improvements have been found for LFK 7 and LFK 8. Attempts to apply cyclic scheduling methods and unrolling resulted in *worse* performance due to not having enough registers.

9.4.1 LFK 7

LFK 7 is the only loop that is bottlenecked by the floating-point unit, not the memory unit. It has 8 essential floating-point add operations and 8 essential floating-point multiply operations. From the flow graph of the floating-point operations, there are at least 8 dependences. With the latency of each floating-point operation being 6 clock cycles, the total number of cycles for an iteration without unrolling is 48 clock cycles. If the compiler applies cyclic scheduling, unrolling is also necessary in order to prevent the code from expanding over too many iterations. But then the interaction between the fixed positions of the load instructions and the dependence among the floating-point instructions will require more registers than expected. In any case, even with optimum reuse of the registers, there will not be enough registers. The performance of the hand-coded schedule is actually worse than the compiled performance, dropping to 37% of the performance bound.

9.4.2 LFK 8

LFK 8 deals with six data arrays; three of them are three-dimensional. It has the highest memory busy time among all the LFK loops, and with 36 floating-point operations needed to hide under the memory time. The Alpha does not have enough registers to make this already complicated loop even more complicated. The loop stays at 62% of the performance bound.

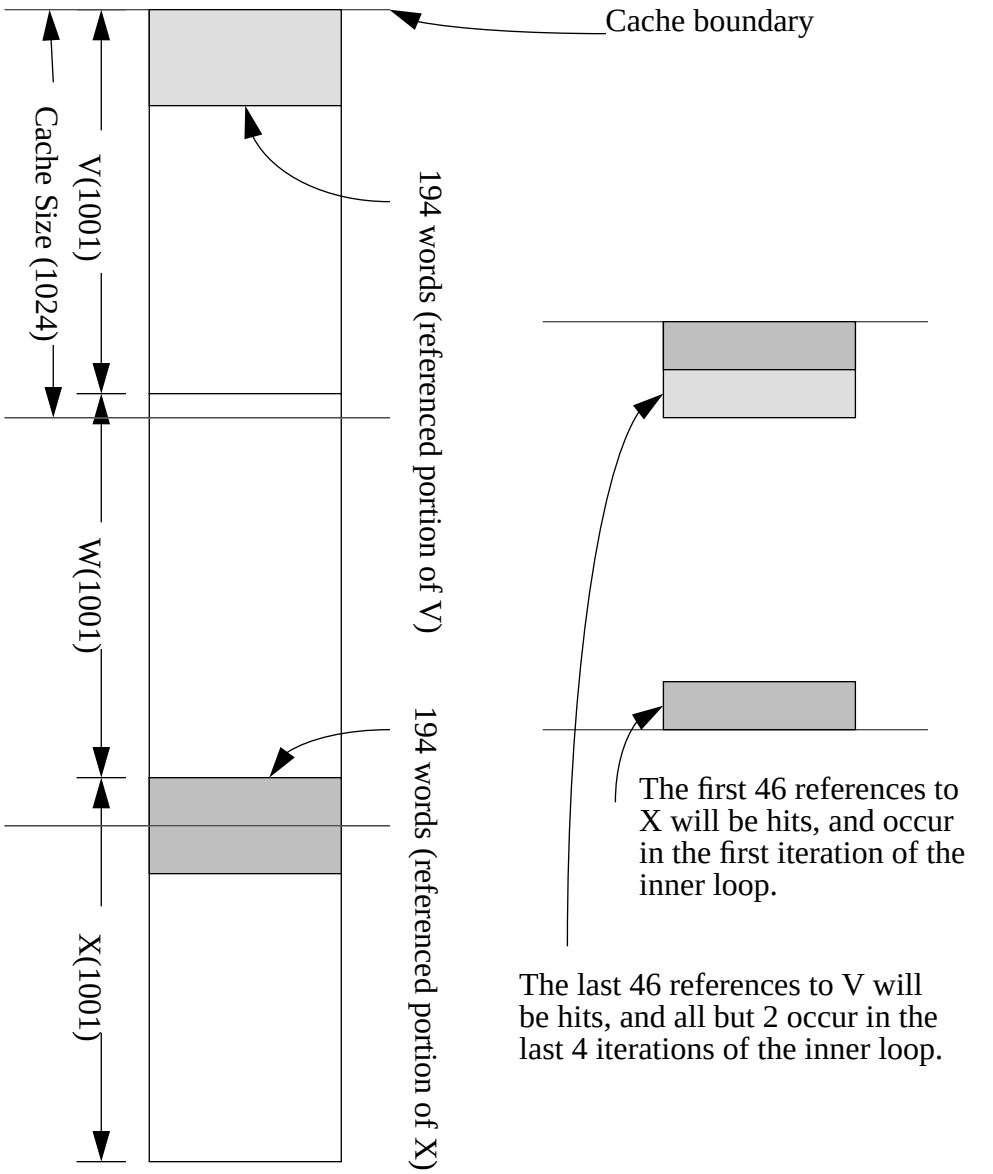

FIGURE 16. Cache Locations of the Arrays of LFK2

10.0 Conclusion

We argued that performance bound models can be extended to include the effects of the unavoidable performance loss, as long as the scheduling of the code is stable and the cache behavior is predictable. We presented performance bound model that includes cache effects, and applied this model to the DEC Alpha machine.

We showed that the performance can be improved dramatically through careful scheduling, guided by the results of applying the performance bound and an assessment of the compiled code. When dealing with load misses, we must schedule in a way that the load miss templates are

placed into the modulo reservation table first. When dealing with store instructions, there is not much we can do because of the unpredictable behavior of the write buffer. When dealing with trailing-edge effects, which directly relate to the speed of the secondary cache, an analysis of the code can locate the possible occurrences of these effects. With these aspects in mind, the cyclic scheduling technique was used to form this new approach to loop scheduling which takes the cache effects into consideration.

We experimented with these techniques using the LFK kernels on the Alpha. With careful hand-scheduling, seven out of twelve loops actually achieved high performance, close to the performance bound. Other loops showed only moderate or slight improvement due to factors such as unpredictable write buffer behavior, irregular data array layouts, and/or register spilling. Two of the loops are remained far from the bound and were not improvable by our techniques. Only one had compiled code that was already close to the bound. In general, if load misses can be predicted, significant performance gains can be achieved by hand-scheduling. The techniques used for hand-scheduling are thus attractive for incorporation in future compilers.

References

[6] W. Azeem, *Modeling and Approaching the Deliverable Performance Capability of the KSR1 Processor.* Master thesis, Computer Science and Engineering Division CSE-TR-164-93, University of Michigan, 1993.

[7] E. L. Boyd, W. Azeem, H. H. Lee, T. P. Shih, S. H. Hung, E. S. Davidson, "A Hierarchical Approach to Modeling and Improving the Performance of Scientific Applications on the KSR1" in *Proceeding of International Conference on Parallel Processing,* August 1994.

[8] E. L. Boyd, E. S. Davidson, "Hierarchical Performance Modeling with MACS: A Case Study of the Convex C-240" in *Proceedings International Symposium on Computer Architecture,* pp. 203-212. May 1993.

[9] Digital, *DECChip 21064 -- AA RISC Microprocessor, Preliminary Data Sheet.* Digital Equipment Corporation, Maynard, Mass., 1992.

[10] P. Y. Hsu, *Highly Concurrent Scalar Processing.* PhD thesis, Coordinated Science Laboratory Report #CSG-49, University of Illinois at Urbana-Champaign, 1986.

[11] W. H. Mangione-Smith, *Performance Bounds and Buffer Space Requirements for Concurrent Processors.* PhD thesis, Computer Science and Engineering Division CSE-TR-129-92, University of Michigan, 1992.

[12] W. H. Mangione-Smith, T. P. Shih, S. G. Abraham, E. S. Davidson, "Approaching a Machine-Application Bound in Delivered Performance on Scientific Code" in *Proceedings of the IEEE. Vol 81. No. 8.*, pp. 1166-1178. August 1993.

[13] T. P. Shih, *Performance Evaluation of IBM RS/6000,* Directed Study Report, Department of Electrical Engineering and Computer Science, May 1992.

[14] R. L. Sites, Ed. *Alpha Architecture Reference Manual.* Digital Press, Bedford, Mass., 1992.