Sunder: A Programmable Hardware Prefetch Architecture for Numerical Loops

Tzi-cker Chiueh
Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

chiueh@cs.sunysb.edu

Abstract

Beyond data caching, data prefetching is by far the most effective way to address the memory access bottleneck associated with high-performance processors. This is particularly true for scientific programs whose working sets cannot be easily fit into the on-chip data cache. This paper proposes a new data prefetching architecture called **Sunder**, *which combines the flexibility and accurateness of software prefetching and the transparency and low-overhead of hardware prefetching. The heart of the design is a dedicated Prefetch Engine that is programmable at run time by the software. An important design decision is to keep the Prefetch Engine completely isolated from the normal instruction execution pipeline except a loop counter to keep the two synchronized at the boundaries of loop iterations. A detailed simulation study on the* **Sunder** *architecture shows that compared to the cache-only architecture,* **Sunder** *achieves an average relative performance advantage over cache-only architectures ranging from 28% to 46%, with smaller cache block sizes leading to greater performance improvement.*

1 Introduction

The performance of uniprocessors has seen a tremendous improvement within the last five years. Both faster clock rate and more aggressive exploitation of instruction-level parallelism contribute to this seemingly never-ending technology progress. However, there are two emerging factors that may break the evolution of microprocessor architecture away from the path it took since the (re-)invention of RISC. First, almost every architectural idea that has been attempted over the course of microprocessor development had been tried in the supercomputing/mainframe context. The list includes concepts as basic as caching to more advanced techniques such as dynamic instruction scheduling. At this point in time, there seems to be a consensus in the community that we are exhausting most, if not all, of the significant tricks. Therefore, it appears to be a good time to step back and re-think some of the fundamental issues, such as instruction set and overall micro-architecture, from the first principle. Second, ever since the Mead & Conway revolution, computer architects are educated with the mentality that silicon is precious and therefore many design tradeoffs tend to be made towards minimizing transistor count. Although every transistor still counts for attaining the optimal performance, an abundant transistor budget may invalidate some of the basic design assumptions made more than a decade ago. With the projection by Intel that by the end of this century computer architects would have around 100 million transistors to work with, we argue that this technology development shouldn't be interpreted as merely more transistors, but as a strong invitation to re-examine the microprocessor architecture from the ground up.

With the increasing speed mismatch between processors and main memory, on-chip memory subsystem becomes the most critical factor that affects a microprocessor's performance. Two key ideas to improve the effectiveness of on-chip memory are reducing the need to access data remotely and increasing the overlap between computation and remote data access. Caching embodies the first idea and has been studied extensively about various aspects of its design (block size, associativity, etc.) in a wide variety of contexts (TLB, Branch Target Buffer, etc.). Prefetching and non-blocking cache realize the second idea but receive relatively less attention. Prefetching and non-blocking cache are essentially two sides of the same coin since they both intend to overlap with the same piece of computation except that one occurs before cache miss while the other after. The focus of this paper is on prefetching because prefetching seems to offer more performance improvement potential than non-blocking cache, and because prefetching is the architecturally cleaner alternative between the two.

The design space of data prefetching mechanisms are characterized by three parameters: *what* are to be prefetched, *when* are they prefetched, and *where* are prefetched data stored. The first two questions distinguish software-based from hardware-based prefetching mechanisms, while the last question is related to the issue of cache interference between current and future working sets. The simplest form of hardware-based prefetching exploits spatial locality by fetching the memory blocks immediately following the cache line for which a cache miss or hit just occurs. A more sophisticated form of hardware prefetching [CHEN93] uses a special address prediction table, the entries of which correspond to the load/store instructions in the program. When a memory access instruction is executed, the system searches the address prediction table with the address of this instruction. If there is a match, that means this instruction has been executed earlier, perhaps in the previous iterations of a loop. The difference between the target memory addresses of the two execution instances of the instruction would then be calculated and used as a *stride* to predict the target memory address of the next execution instance. Typically there is a built-in history tracking mechanism, implemented through a finite-state machine, to ensure the stability of the stride values. That is, for a memory access instruction, the hardware will defer the prediction decision until at least few matches to the corresponding address prediction table entry have occurred, thus avoiding unnecessary prefetching.

In either of the above two schemes, the *what* and *when* decisions of prefetching are made as side effects of memory access instructions. As a result, the *prefetch distance*, the temporal distance between a load/store instruction and its corresponding prefetch instruction, is usually limited to one loop iteration, which may not be sufficient to mask the off-chip memory access delay completely. Also, the hardware cost associated with maintaining address prediction information and issuing prefetch requests seems to be relatively expensive. Worst of all, the extra processing overhead tends to lengthen the critical path of on-chip data cache access, which usually plays a critical role in determining the processor cycle time.

Software-based prefetching [KLAI91], on the other hand, requires the compiler to issue explicit **FETCH** instructions far ahead enough in time to mask the memory access latency should the target data words were not already in the cache. **FETCH** instructions behave almost exactly the same as normal memory access instructions except that they are non-blocking and non-trapping, i.e., cache miss would not stall the processor and page fault is ignored. Therefore, the *what* and *when* decisions of prefetching are made at compile time, completely eliminating the need of any prediction hardware. However, software-based prefetching suffers from two pitfalls. First, every memory access now requires two instructions, one to prefetch the data from main memory to the cache, and the other to load the data into the register file. When caching alone can successfully capture most of the programs' locality, the extra prefetch instructions may prove to be a serious performance overhead [CALL91]. Second, the microoperations of **FETCH** instructions could not readily be integrated into the generic processor pipeline because **FETCH** needs to check the data cache and issue off-chip memory requests if the target data is not in the cache. Implementation incompatibility of **FETCH** with other instructions may subtly affect the system performance parameters such as the clock rate. In addition, the fact that **FETCH** and other instructions share the same execution pipeline also limits the flexibility of prefetch timing.

As to the *where* question, there were three possible choices proposed in the literature. The first possibility is to put prefetched data into the data cache, and hope that the cache pollution problem won't arise. The second approach [KLAI91] uses a fully-associative FIFO buffer, with each entry as wide as a cache line, to hold prefetched data, and memory access instructions executed against the FIFO buffer. Cache lines in the prefetch buffer are moved to the data cache when they are actually referenced. One of the serious drawbacks of this approach is the use of fully-associative buffer, which either slows down the on-chip memory access or requires extravagant hardware resources. The third proposal [CHEN92] uses a preload FIFO queue to hold prefetched data. In most cases memory access instructions "pop" the target data words from the preload queue. Only when there is potential inconsistency between the data cache and the preload buffer will data cache access using explicit memory address becomes necessary. Also, data in the preload buffer will be transferred to the cache if and only if the data word is the target of a write access. The problem with this scheme is that complicated matching hardware and additional *prestore* and *skip* instructions are needed for detecting data inconsistency and queue management.

There is a fundamental tradeoff between interference and coherence. To eliminate the interference between current and future working sets, one must resort to separate storage structures to hold the two working sets. However, separate storage structures inevitably lead to data inconsistency when there is a consumer/producer data sharing relationship between the two working sets. *Sunder* addresses this dilemma by making the following crucial observation: Because all array references are brought to on-chip memory by the prefetching hardware, each prefetch request only needs to check with the set of write prefetch requests whose corresponding store instructions haven't yet been executed to detect potential data inconsistency. Consequently, consistency checking can be readily integrated with data prefetching's associative look-up process without additional hardware.

Table 2 summarizes the advantages and disadvantages of the various approaches to the three design issues of data prefetching. The architecture proposed in this paper, *Sunder*, takes a programmable hardware prefetching approach to deal with the what/when problem. This approach combines the flexibility of software prefetching and the transparency of hardware prefetching, and thus achieves the best of both worlds without suffering from their problems. As to the where question, *Sunder* adopts a storage structure called *array register file*, which avoids the expensive hardware associated with prefetch buffer and preload queue, and still reduces the cache interference effect to the minimum.

The remainder of this paper is organized as follows. Section Two describes the basic programming model of the *Sunder* architecture. Examples are presented to illustrate the model and its advantages. Section Three discusses the hardware organization and compiler support for *Sunder.* In Section Four, a preliminary performance analysis of *Sunder* based on the Livermoore Loop set are presented. In Section Five, previous works on data prefetching are briefly reviewed to set the contributions of this work in perspective. Section six concludes this paper with a summary of results and a pointer to the follow-on work.

2 The Sunder Architecture

2.1 Architectural Model

The data prefetching mechanism of *Sunder* is designed specifically for array references in scientific programs. In particular, the focus is on the array references in the inner loops. Each such array reference is characterized by its base address and stride. The loop bound implicitly specifies the maximum number of instances of each array reference at run time. Distinct references to the same array, e.g., B[2*I+1] and B[I+3], are treated as separate array references. *Sunder* features a programmable prefetch engine that takes the inner loop bound and the <base address, stride> pair of each array reference, and performs prefetching of array data according to the array reference characterization pairs. Because the prefetch engine executes prefetching operations completely *independently* of normal instruction execution, there is no extra overhead visible to the applications as in software prefetching. Because the memory addresses of target data words are gathered through compile-time analysis, there is no need for address prediction as in hardware prefetching.

Before the inner loop starts, the loop bound and the characterization pairs of the array references are deposited into a hardware structure of the prefetch engine called the *Array Reference Table.* After receiving a special *prefetch_start* instruction, the prefetch engine begins to process the array references in a round robin fashion completely autonomously, until reaching the specified loop bound. The synchronization between the prefetch engine and the instruction execution pipeline is through a loop counter that keeps track of the iteration count of the inner loop. That is, when the loop counter is incremented to I, the prefetch engine starts to prefetch the working set for the next iteration, I+1. Normally the prefetch engine has less work than the execution pipeline, and so it will sit idle after the completion of prefetching the current round, until the loop counter is incremented. Note that the proposed prefetch mechanism doesn't require that the loop bound and the strides be known at compile time. In fact, strides are allowed to change dynamically without affecting the correctness of the proposed approach.

There are two copies of *Array Register Files* (ARF) in *Sunder.* At any point during the execution of the inner loop, one is receiving the data prefetched from main memory (the background ARF) and the other is servicing normal memory accesses (the active ARF). At run time, each loop iteration is bound to one of the ARF's, which alternate between the active and background roles as loop iterations unfold. At compile time, an array register is allocated for each array reference in the inner loop, regardless of whether that array reference will be executed at run time. The size of the array register file is an architecture parameter and is set to accommodate the need of the inner loops in most scientific programs. When the number of array references is much smaller than the size of the array register file, loops should be unrolled to make more efficient use of array register files. Unrolling the loops also increases the effective prefetch distance, which in turn could mask longer memory access delay.

The array references in the inner loop access the array register files directly rather than the data cache. That is, they take the following form:

```
load     scalar-register-n  array-register-m
store    scalar-register-n  array-register-m
```

where *array-register-m* is the array register allocated for the array reference corresponding to the load/store instruction. One important advantage of this approach is that address computation completely disappears from the instruction stream because the prefetch engine performs this computation implicitly. Moreover, the hardware implementation for this memory access architecture is even simpler than conventional load/store instructions, and can be fit nicely with a generic RISC processor pipeline, as we will show later.

2.2 An Example

Let's use the first loop in the Livermoore Loop suite as an example to illustrate the architectural model of *Sunder*. For simplicity, only the inner loop is shown here.

```
for(k=0; k < 400; k++) {
    a1 = r*z[k+10];
    a2 = t*z[k+11];
    a3 = (a1+a2);
    a4 = y[k] * a3;
    x[k] = q + a4;
}
```

The MIPS assembly code for this loop is

```
$33:
 li      $14, 0          /* load k */
 la      $12, z          /* load z[]'s BA */
 la      $7, y           /* load y[]'s BA */
 la      $6, x           /* load x[]'s BA */
 l.s     $f4, 260($sp) /* get r */
 l.s     $f10, 256($sp)/* get t */
 l.s     $f16, 172($sp)/* get q */
$34:
 mul     $11, $14, 4
 addu    $13, $11, $12 /* form z[k]'s address */
 mul     $17, $14, 8
 addu    $19, $17, $7  /* form y[2*k]'s address */
 addu    $18, $11, $6  /* form x[k]'s address */
 l.s     $f6, 40($13)  /* get z[k+10] */
 mul.s   $f8, $f4, $f6
 l.s     $f16, 44($13) /* get z[k+11] */
 mul.s   $f18, $f10, $f16
 add.s   $f4, $f8, $f18
 l.s     $f6, 55($19)  /* get y[2*k] */
 mul.s   $f10, $f6, $f4
 add.s   $f8, $f16, $f10
 s.s     $f8, 8($18)   /* store x[k] */
 addu    $14, $14, 1
 blt     $14, 400, $34
```

In this case, there are four array references in the loop, three loads and one store. Under *Sunder*, the same code is compiled into the following sequence:

```
 la      $12, z
 addi    $12, $12, 40
 store_array_reference_table   #1, $12, 4, 0
 addi    $12, $12, 4
 store_array_reference_table   #2, $12, 4, 0
 la      $7, y
 store_array_reference_table   #3, $7, 8, 0
 la      $6, x
 store_array_reference_table   #4, $6, 4, 1
 prefetch_start 400
 l.s     $f4, 260($sp) /* get r */
 l.s     $f10, 256($sp)/* get t */
 l.s     $f16, 172($sp)/* get q */
 nop     20
$34:
 l       $f6, $ARF1    /* get z[k+10] */
 mul.s   $f8, $f4, $f6
 l       $f16, $ARF2   /* get z[k+11] */
 mul.s   $f18, $f10, $f16
 add.s   $f4, $f8, $f18
 l       $f6, $ARF3    /* get y[2*k] */
 mul.s   $f10, $f6, $f4
 add.s   $f8, $f16, $f10
 s       $f8, $ARF4    /* store x[k] */
 addu    $14, $14, 1
 blt     $14, 400, $34
```

The *store_array_reference_table* instruction stores a <base address, stride> pair into a particular entry of the array reference table, as specified by the first argument. The last argument indicates whether the prefetch is on behalf of a read (0) or a write (1). After the initialization, the *prefetch_start* instruction starts the prefetch engine by filling the loop bound information, in this case, 400. Before the loop starts, the program executes a "nop 20" instruction, which stalls the main CPU for 20 cycles to allow the prefetch engine to bring in the working set associated with the first iteration. The number of stall cycles depends on the number of array references and the memory system performance. Within the loop, the array references turn into data movement among the array register file and the normal scalar register file. Note that the address computation code sequence in the original program is completely eliminated in the new program.

3 Hardware and Software Support

3.1 Hardware Organization

The most important design decision of *Sunder*'s data memory architecture is to take a programmable hardware-prefetching approach. Because of its programmability, compile-time knowledge of program access patterns can be readily incorporated into the prefetching mechanism. Because of the dedicated prefetch hardware, no additional instruction overhead is visible to the applications. Moreover, *Sunder* decouples prefetching hardware from the instruction execution pipeline, thus reducing the potential performance impact on processor cycle time to the minimum.

The on-chip memory system hardware of *Sunder* is shown in Figure 1. The Array Reference Table (ART) contains one entry for each array reference in the inner loop. Each entry contains the target address of the current instance, the type (read or write), and the stride of the corresponding array reference. The Prefetch Engine processes one ART entry at a time in a round robin fashion, and synchronizes with the normal instruction stream via the Loop Counter. Specifically, after one round through the ART entries, the Prefetch Engine will stop until the Loop Counter is incremented by the loop-back branch instruction associated with the inner loop, at which point a new prefetch cycle is repeated. The Loop Counter is updated implicitly the loop-back instruction and therefore is transparent to the software. The Prefetch Engine determines when to stop the prefetching process by comparing the Loop Counter with the Loop Bound register.

If an off-chip memory transaction is needed to service a prefetch request, the prefetch address is entered into the Pending Prefetch Queue (PPQ). When a memory request is returned, the corresponding prefetch request in the PPQ is removed. If a prefetch request is issued on behalf of a write array reference, the associated target address is also entered into the Pending Store Queue (PSQ) regardless of whether an off-chip memory access is needed. The existence of PSQ greatly facilitates the run-time detection of read-after-write data dependencies without introducing special *prestore* instructions as in [CHEN92]. The PSQ entries corresponding to the stores within one iteration are removed as one batch, again triggered by an increment in the Loop Counter. Both PPQ and PSQ are organized as a fully-associative queue, with the target addresses of the prefetched cache lines and stored words as tags, respectively. Because the Prefetch Engine is decoupled from the instruction execution pipeline, full associativity poses little effect on the processor cycle time. Note also that PPQ is mandatory in *all* data prefetching schemes.

In addition to the data cache and the scalar register file, there are two additional storage structures called Array Register Files (ARF). When a cache line is prefetched from the memory, it is put into the data cache. However, the target data word is put into one of the ARF's. The array references in the loop are executed against the ARF's rather than the data cache. The two ARF's alternate their roles between active and background array register files. That is, suppose the I-th iteration is currently accessing the first ARF, then the Prefetch Engine is prefetching the data words needed for the (I+1)-th iteration into the second ARF. When the I-th iteration completes, the two ARF's switch roles.

Each array reference in the loop body is associated with an array register and corresponds to a prefetch request. There are two issues associated with the implementation of a prefetch request: 1) Is there a need to issue an off-chip memory request; 2) Should the prefetched data be put into the array register file? 3) Update the current address for the next iteration. The detailed operations for read prefetch are slightly different from write prefetch.

The system decides whether an off-chip memory access is necessary by using the target address associated with a prefetch request to simultaneously look up the tag memory of the data cache (DC) and the Prending Prefetch Queue (PPQ). In the case of read prefetch, an additional look-up into the Pending Store Queue (PSQ) is needed to prevent data incoherence between Array Register Files and the data cache. When neither DC nor PPQ look-up succeeds, a memory request for the cache line that includes the target data word is sent to either Level-2 cache or the main memory in the case of write prefetch. For read prefetch requests, an off-chip memory request is needed only when both DC and PPQ look-ups fail and the PSQ look-up result is also negative. The extra check ensures that the prefetched data will not be obsoleted by a subsequent store. After the memory request is issued, an <*offset, register identifier*> source/destination pair is appended to the tail of the PPQ, where *offset* represents the offset of the target word within the fetched cache line, and *register identifier* denotes the prefetch request's associated array register. The tag of this new entry is the base address of the fetched cache line. Each PPQ entry is allowed to hold M such pairs. where M is greater than 1 and is an architectural parameter. In other words, when a prefetched cache line arrives, the hardware may need to extract multiple target words and put them to their associated array registers.

If the DC look-up returns a match, the target data word is transferred directly form the data cache to the prefetch request's associated array register. Otherwise if the PPQ look-up returns a match, that means a previous prefetch with the same target cache line has been sent. If the matched PPQ entry is not full, an <*offset, register identifier*> source/destination pair is added to the matched PPQ entry, and there is no need to issue another off-chip memory access. However, in the case that the matched PPQ entry is full, then the PPQ look-up is treated as a mismatch.

Memory requests are assumed to be serviced in the same order as they are issued. So when a cache line is returned, it will be stored into the data cache, target words are extracted from the cache line and loaded into the corresponding array registers, as specified by the source/destination pairs in the PPQ's head entry, and then the head of PPQ advances.

The next question is whether the prefetched data word should be put in the corresponding array register. In the case of write prefetch, there is no need to do this because the target data word is supposed to be overwritten. For read requests whose PSQ look-up returns a match, it is semantically incorrect to load target data words into the associated array registers because the prefetched word is going to be modified by a subsequent write before its associated memory access. In both cases, the associated array registers are instead loaded with the memory addresses of the target words and appropriately so flagged at the time when the prefetch requests were processed. For these prefetch requests, the *register identifier* fields in the corresponding source/destination pairs, if any, are filled with **0X0**, effectively bypassing the loads when cache lines return. Table 3 summarizes the conditions under which an off-chip memory request should be issued and whether the target data words should be loaded into the array register files. Not every hit/miss combination makes sense.

Normal memory access instructions actually move the data word from the array register file to the scalar register file if the corresponding array register is not flagged. Otherwise, the system will use the contents of the array register as a memory address and performs a cache access. Assume a generic RISC pipeline: instruction fetch (IF), register fetch (RF), execution (ALU), memory access (MEM), and write back (WB). The contents of the array register are *always* transferred to the ALU during the RF cycle. Then depending on the flag of the array register, either the data word is moved to the destination scalar register directly (ALU), or an effective memory address is formed out of the data word (ALU) and sent to the cache memory (MEM). As a result, unlike previous approaches, the proposed data prefetching mechanism *doesn't* affect the memory access pipeline and thus the critical path remains the same. In fact, under *Sunder* the instruction immediately following a load instruction can use the result from the load without experiencing any load latency. This is possible because all the associative look-ups are delegated to the Prefetch Engine, which is completely isolated from the execution of normal instruction streams. It should be noted that in most cases the flags are off, which means that the memory load instructions actually obtain the target data words from the array register file without accessing the data cache. In these cases, the prefetched working set is guaranteed to be immune from cache interference.

3.2 Compiler Support

There are two parameters that the compiler needs to determine to take advantage of *Sunder*'s data prefetching mechanism: the degree of loop unrolling and the initial no-op cycle count. The latter actually depends on the former, so we will focus on the degree of loop unrolling, which determines the number of iterations to look ahead and thus the amount of prefetching. In order to minimize the cache pollution effect, the ideal amount of prefetching should be just enough to mask the memory access delay. Anything more than that would not improve the on-chip memory system performance and in effect could hurt because of cache interference.

Suppose that the numbers of processor cycles needed to execute the loop body and the loop overhead are C_{body} and $C_{overhead}$, respectively. Furthermore, let the number of memory access instructions in one iteration be I_{access}, the ratio between memory cycle time and processor cycle time is C_{cycle}, the number of banks in the memory system be N_{bank}, and the number of array registers be $N_{array\ register}$. Because excessive prefetching could lead to cache pollution, the degree of unrolling should be the minimum K that satisfies the inequality:

$$K * \frac{I_{access} * C_{cycle}}{N_{bank}} \leq K * C_{body} + C_{overhead} \qquad (1)$$

subject to the constraint that the number of array registers available for one unrolled loop iteration is bounded.

$$N_{array\ register} \geq I_{access} * K \tag{2}$$

Here we are assuming that the memory references are uniformly spread across the memory banks, i.e., no contention, and memory references don't overlap with one another. More sophisticated data flow analysis techniques [WOLF89] can be applied to determine whether some of the memory references in the loop actually target at the same cache lines, and therefore could share the cost of off-chip memory accesses. In addition, memory bank disambiguation techniques [ELLI86] can be used to calculate the impact of memory contention on the effective memory system bandwidth. For instance, in the loop example shown earlier, two of the array references, z[k+10] and z[k+11], tend to reside in the same cache line most of the time. Therefore, only three, rather than four, off-chip memory accesses are needed for one iteration. As loop iterations get unrolled, sharing of common memory access tends to occur more often. For example, the array references x[k] in the I-th iteration and (I+1)-th iteration will most likely occupy the same cache lines, thus reducing off-chip memory accesses.

Therefore a more general version of Equation (1) is

$$\mathcal{F}(K,\ I_{access},\ C_{cycle},\ N_{bank}) \geq K * C_{body} + C_{overhead} \tag{3}$$

where $\mathcal{F}(K,\ I_{access},\ C_{cycle},\ N_{bank})$ is the number of processor cycles to prefetch the array references in K iterations of the loop. Based on the constraint imposed by Equation (2), the compiler computes the maximum value of K, $K_{maximum}$. By taking into account both spatial locality and memory contention, the compiler determines the minimum K, $K_{qualify}$, such that Equation (3) holds. The final K is chosen to be the smaller of $K_{maximum}$ and $K_{qualify}$.

After K is determined, the number of no-op cycles to be put in the beginning of the inner loop, is exactly $\mathcal{F}(K,\ I_{access},\ C_{cycle},\ N_{bank})$. It is possible to dispense with the no-op instructions and let the array references in the first iteration cause cache misses. However, handling a sequence of individual cache misses usually is more expensive than brining the needed data words beforehand in one batch. Therefore, it is preferable to prefetch the working set of the first iteration before the the loop starts. By applying this compiler algorithm to the loop shown in the previous section, the optimal degree of unrolling, K, is found to be 5, and the initial waiting period is 70 cycles.

4 Performance Evaluation

Based on the MIPS R2000/3000 instruction set, we construct a detailed simulator that models the functional behavior of the *Sunder* architecture. The following architecture parameters are built into the simulator. Each Array Reference Table entry corresponds a prefetch request and requires two processing cycles, one for associative look-up and one for issuing an off-chip memory access if necessary. The latency of prefetching N bytes is $6 + N$ cycles, while the cache miss handling overhead is $12 + N$ cycles, where N is the cache block size in bytes. The addition, multiplication, and division take 1, 3, and 12 cycles, respectively. The on-chip data cache is direct-mapped and has 16 Kbytes. Since we are mainly interested in numerical programs, we will base the performance evaluation on the Livermoore loop suite. The inner loop in each of the kernels is first unrolled according to the algorithm described above, with appropriate initialization code inserted. Each unrolled kernel, after compiled, is run to completion on the simulator, which then reports the total cycle count. During the process of unrolling, we make the worst case assumption in that potentially aliased array references are assumed *not* to be aliased. That is, we assume that the spatial locality among array references is lower than what actually is available. We also assume there is only one memory bank. Therefore, the results we report in the following tend to under-estimate the performance potential of *Sunder*.

We also run the kernels on a generic MIPS R2000/3000 simulator with caching only, and measure the total cycle counts. To make a fair comparison, each loop is also unrolled to the same extent before being executed on this simulator. The performance metric we used is

$$\frac{Cycle\_Count_{caching}\ -\ Cycle\_Count_{prefetching}}{Cycle\_Count_{caching}}$$

Table 1 shows the performance advantage of *Sunder*'s prefetching mechanism over the cache-only architecture for several loops and various cache block sizes. The performance improvement ranges from under 15% to over 60%, with the harmonic mean between 28% to 46%.

The most important factor affecting data prefetching's effectiveness is the ratio between computation work and memory access, $R_{computation/memory}$, and the spatial locality of the array references. If $R_{computation/memory}$ is too small, then even loop un-

Program Name	Cache Block Size (Biytes)		
	16	32	48
Kernel 1	50.96%	40.87%	29.78%
Kernel 2	29.63%	20.83%	13.64%
Kernel 7	49.10%	37.96%	33.07%
Kernel 9	59.14%	54.87%	49.60%
Kernel 10	62.26%	52.38%	47.83%
Kernel 14	42.50%	33.81%	30.30%
Harmony Mean	46.07%	36.21%	28.41%

Table 1: Relative Performance Advantage of *Sunder* over Simple-Caching

rolling cannot help prefetching to completely mask the memory access latency. As a result, the relative performance improvement of *Sunder* over the cache-only architecture diminishes. The comparatively small performance advantage shown by Kernel 2 is an example of this effect. When the spatial locality among array references in the loop increases, the cache-only architecture could avoid most of the memory access penalty. Consequently, prefetching is not as effective in relative terms. This is especially true when the cache block size is large, because more spatial locality could be captured. That is the reason why the relative performance advantage of *Sunder* decreases as cache block size increases. Although one of the advantages of the *Sunder* architecture is the elimination of array base address computation, it turns out not to be a significant performance factor because loops are unrolled. However, the fact that *Sunder* doesn't require separate prefetch instructions for each memory access, unlike software prefetching, does play a crucial role. We have also tried different cache sizes and degrees of associativity. In general, the performance difference is rather marginal because in the programs we examined spatial locality almost always dominates temporal locality.

5 Related Works

Sophisticated data prefetching techniques didn't emerge until the last four years. We have discussed the general principles of software-based and hardware-based data prefetching in the **Introduction** section, and will refer readers who are interested in detailed citations to [CHEN93]. *Decoupled architecture* [SMIT82] is based on a design philosophy somewhat similar to the *Sunder* architecture. Decoupled architecture addresses the memory access bottleneck by decomposing an instruction stream into two streams: the memory access stream and the data manipulation stream. Data prefetching is implicitly built into decoupled architectures in that memory access streams are always ahead of data manipulation streams, and the two streams synchronize with each other through FIFO queues. There are two problems associated with decoupled architectures. First, decoupled architecture is essentially a restricted form of homogeneous superscalar architecture because the types of instructions that are allowed to execute on the two streams are architecturally defined. When the loads on the two streams are imbalanced, there is bound to be performance penalty due to this architectural asymmetry. Like software prefetching, each memory access also requires two instructions, one to bring the data from main memory to FIFO queues and the other from FIFO queues to register files. Second, to mask the memory access delay, the memory access stream must be sufficiently far ahead of the data manipulation stream. But this requires accurate prediction of program control flows, which usually are determined by the data manipulation stream. So, there seems to be a fundamental limit on the temporal distance between the memory access and data manipulation streams, as imposed by the architecture. In addition, FIFO queues are particularly vulnerable to inaccurate predictions because the contents of FIFO queues have to be flushed completely every time the control flows are mispredicted.

In some sense, the prefetching mechanism of *Sunder* is just a scalar version of vector instruction. However, there are two important differences that render *Sunder* a more effective architectural choice. First, *Sunder* allows data dependency among concurrently-executing vector memory access instructions because it checks data dependencies before accessing any vector element. This in essence allows conventionally unvectorizable code to run in the vector mode [CHIU91]. Second, rather than using a large vector register file, *Sunder* uses a pair of alternating register files. Not only is the hardware cost significantly reduced, but also the memory access mechanism won't suffer long start-up overhead and is able to keep close synchronization with the normal instruction execution stream.

6 Conclusion

In this paper, we propose a new data prefetching architecture called *Sunder* that attempts to address the inefficiencies associated with previously proposed software-based and hardware-based prefetching schemes. *Sunder* takes a programmable hardware prefetching approach that combines the flexibility and accurateness of software prefetching and the transparency and low-overhead of hardware prefetching. The heart of the design is a dedicated Prefetch Engine that is programmable at run time by the software. An important design decision is to keep the Prefetch Engine completely isolated from the normal instruction execution pipeline except a loop counter to keep the two synchronized at the boundaries of loop iterations. As a result, *Sunder* doesn't need to pay additional prefetch instruction overhead and the hardware performance of the Prefetch Engine would not affect the processor cycle time. *Sunder* also features an interesting array register-file pair to effectively addresses the dilemma between data coherence and cache interference that plagues previous data prefetching mechanisms. To validate the performance goal of the proposed architecture, we carry out a detailed simulation study based on several complicated loops in the Livermoore loop suite. Compared to the cache-only architecture, *Sunder* achieves an average relative performance advantage ranging from 28% to 46%, with smaller cache block size leading to greater performance improvement.

As for future work, we are currently investigating the extension of the proposed mechanism to accommodate outer loops. The main problem is to keep the contexts associated with the outer loops while the inner loops are unfolding. An interesting alternative is to use *loop collapsing* to flatten nested loops into singly-nested forms and directly apply the mechanism described in this paper. In any case, we will first examine the potential performance gain to justify special hardware/software support for outer-loop data prefetching. We are also looking more closely at the hardware implementation cost of *Sunder* to get a more concrete understanding of the cost-effectiveness of the proposed mechanism.

7 Reference

[CALL91] Callahan, D.; Kennedy, K.; Porterfield, A., "Software prefetching." Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, USA, 8-11 April 1991. SIGPLAN Notices, April 1991, vol.26, (no.4):40-52.

[CHEN92] Chen, W.Y.; Bringmann, R.A.; Mahlke, S.A.; Hank, R.E.; and others. "An efficient architecture for loop based data preloading." 25th Annual International Symposium on Microarchitecture. MICRO 25, Portland, OR, USA, 1-4 Dec. 1992. SIGMICRO Newsletter, Dec. 1992, vol.23, (no.1-2):92-101.

[CHEN93] Chen, T.F., *Data prefetching for high-performance processors.* Ph.D. Thesis, TR-93-07-01, Department of Computer Science and Engineering, University of Washington, July 1993.

[CHIU91] Chiueh, T., "Multi-threaded vectorization." Proceedings of the 18th International Symposium on Computer Architecture, Toronto, Ont., Canada, 27-30 May 1991. Computer Architecture News, May 1991, vol.19, (no.3):352-61.

[ELLI86] Ellis, John R., *Bulldog : a compiler for VLIW architectures.* Cambridge, Mass. : MIT Press, c1986.

[FUPA91] Fu, J.W.C.; Patel, J.H., "Data prefetching in multiprocessor vector cache memories." Proceedings of the 18th International Symposium on Computer Architecture, Toronto, Ont., Canada, 27-30 May 1991. Computer Architecture News, May 1991, vol.19, (no.3):54-63.

[FUPA92] Fu, J.W.C.; Patel, J.H., "Stride directed prefetching in scalar processors." 25th Annual International Symposium on Microarchitecture. MICRO 25, Portland, OR, USA, 1-4 Dec. 1992. SIGMICRO Newsletter, Dec. 1992, vol.23, (no.1-2):102-10.

[KLAI91] Klaiber, A.C.; Levy, H.M., "An architecture for software-controlled data prefetching." Proceedings of the 18th International Symposium on Computer Architecture, Toronto, Ont., Canada, 27-30 May 1991. Computer Architecture News, May 1991, vol.19, (no.3):43-53.

[MOWR92] Mowry, T.C.; Lam, S.; Gupta, A., "Design and evaluation of a compiler algorithm for prefetching." Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, MA, USA, 12-15 Oct. 1992. SIGPLAN Notices, Sept. 1992, vol.27, (no.9):62-73.

[SMIT82] Smith, J.E., "Decoupled access/execute computer architectures." Proceedings of the 9th International Symposium on Computer Architecture. p 112-119.

[WOLF89] Wolfe, Michael Joseph, *Optimizing supercompilers for supercomputers.* Cambridge, Mass. : MIT Press ; London, England : Pitman Pub., 1989.

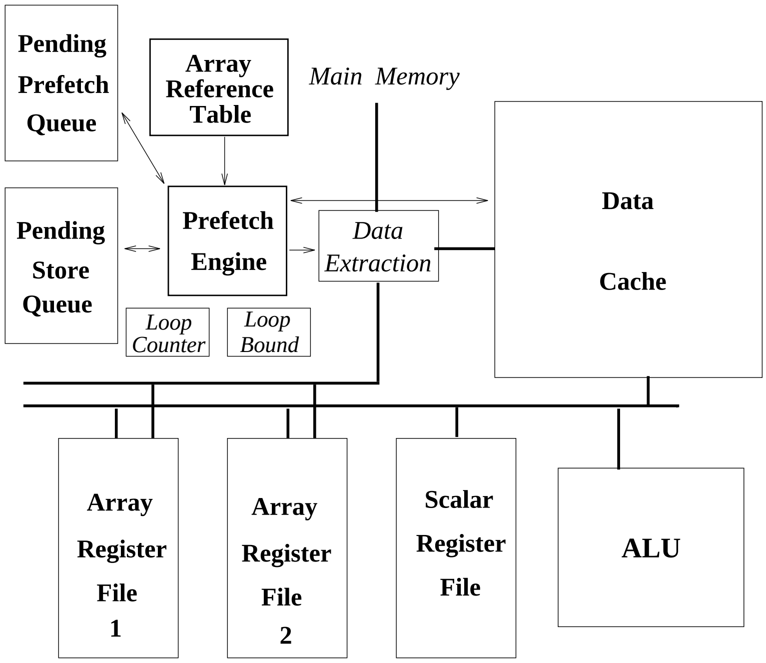

Figure 1: Hardware Architecture of *Sunder*

Issue	Alternative	Problems
What/When	Software	extra instruction overhead
		lengthening the critical path
	Hardware	too late, inaccurate target address
		complicated history tracking
		lengthening the critical path
	Programmable Hardware	extra hardware
Where	Cache	interference between neighboring working sets
	Fully-associative FIFO Buffer	expensive in performance and cost
	Preload FIFO Queue	complicated coherence maintenance
	Array Register File	simpler coherence maintenance

Table 2: Comparison of Approaches to Three Design Issues of Data Prefetching

Look-Up Result			Write Prefetch		Read Prefetch	
Cache	**PPQ**	**PSQ**	**Memory Request**	**To ARF**	**Memory Request**	**To ARF**
hit	miss	hit	No	No	No	No
hit	miss	miss	No	No	No	Yes
miss	hit	hit	No	No	No	No
miss	miss	hit	Yes	No	Yes	No
miss	miss	miss	Yes	No	Yes	Yes

Table 3: Conditions for External Memory Requests and Target Word Load