# Compiler-Enforced Cache Coherence Using a Functional Language

**RICH WOLSKI[1] AND DAVID CANN[2]**

[1]*Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093; e-mail: rich @ cs.ucsd.edu*

[2]*Convex Computer Corporation, P.O. Box 833851, Richardson, TX 75083; e-mail: cann @ convex.com*

## ABSTRACT

The cost of hardware cache coherence, both in terms of execution delay and operational cost, is substantial for scalable systems. Fortunately, compiler-generated cache management can reduce program serialization due to cache contention; increase execution performance; and reduce the cost of parallel systems by eliminating the need for more expensive hardware support. In this article, we use the Sisal functional language system as a vehicle to implement and investigate automatic, compiler-based cache management. We describe our implementation of Sisal for the IBM Power/4. The Power/4, briefly available as a product, represents an early attempt to build a shared memory machine that relies strictly on the language system for cache coherence. We discuss the issues associated with deterministic execution and program correctness on a system without hardware coherence, and demonstrate how Sisal (as a functional language) is able to address those issues. © 1996 John Wiley & Sons, Inc.

## 1 INTRODUCTION

The cost of hardware cache coherence, both in terms of execution delay and operational cost, is substantial for scalable systems [4]. Parallel work must stop while the caches are adjusted [8]. Furthermore, as cache-coherent systems scale in size, the time associated with each consistency operation increases. Small, bus-based systems can typically resolve a cache miss in 5 to 50 processor cycles. Larger systems using distributed memories and directory structures can require up to 500 cycles to resolve a miss, especially if the systems use a local area network as a processor interconnect. In addition, the trend in processor design is toward wider instruction issue on each cycle. For example, both the IBM RIOS 2 and the SGI TFP processing units can issue up to four instructions per cycle. A 100-cycle delay due to a cache miss could imply a relative cost of 400 instructions. Out-of-order instruction issue and hardware write buffering can reduce this cost, but in general, the cache-synchronization delay can seriously impair performance. Fortunately, compiler-generated cache management can reduce the amount of serialization resulting from hardware-based cache coherence. Further, because the dollar cost of scalable systems is high, compiler optimizations for cache coherence can also reduce the need for more expensive hardware support, thereby improving price performance.

In this article, we use the Sisal compiler as a

Received April 1995
Revised June 1995

© 1996 John Wiley & Sons, Inc.

Scientific Programming, Vol. 5, pp. 161–171 (1996)
CCC 1058-9244/96/020161-11

vehicle for implementing automatic, compiler-based cache management. We describe an implementation of Sisal (Streams and Iterations in a Single Assignment Language) [5] for the IBM Power/4. The Power/4 supports shared memory, but relies strictly on the language system to enforce cache coherence. Functional languages are attractive for such a system as they are easily analyzed for parallelism and data dependence. Moreover, the compiler exclusively controls how data is mapped with respect to cache alignment, so it can ensure correct cache management. If future machines are built without hardware coherence, functional programming can drastically reduce their programming cost. Sisal is a good choice for such an implementation as it has been shown to achieve excellent shared memory execution performance for scientific programs on other systems [2].

In the next section, we briefly describe the IBM Power/4. Section 3 details some of the problems associated with software cache management and how we address them using OSC (the Optimizing Sisal Compiler) [1]. In Section 4, we discuss the difficulties in automatically parallelizing imperative languages for the Power/4. In Section 5, we detail and analyze our results in terms of two scientific programs: RICARD and SIMPLE. We discuss both the relative performance (speedup) and the execution time of each program, and identify sources of execution overhead. Finally, in Section 6 we summarize our work and outline future research directions.

## 2 THE IBM POWER/4

The Power/4 architecture, available only briefly from IBM as a product, consists of four RIOS 1 processors connected to a set of seven globally addressable memory banks via a crossbar switch. System software partitions each processor's address space into a privately accessible region and a globally shared region. A different memory module for each processor services private accesses, so in the absence of sharing there is no contention for memory. The system maps contiguous shared memory locations across all memory banks to minimize hot-spot contention. The RIOS 1 processors implement no support for cache coherence. There is no way to externally signal a cache post or invalidate, and no way to bypass the on-board cache and access memory directly. The machine we used was an early prototype supporting 32K bytes of data cache and 8K bytes of instruction cache per processor, both managed using a copy-back policy. In that machine, each cache line is 64 bytes wide and a cache miss causes the processor to stall; there is no support for pre-fetch and no write buffering. Since the cache cannot be bypassed, every read of a memory location, either shared or private, causes a copy of the memory to be cached locally. Similarly, every processor write to memory results in a write to cache only. Data are subsequently moved from cache to memory either when it is evicted from the cache so that the cache line can be reused, or when it is explicitly flushed by the processor. While the Power/4 is no longer commercially available, it represents an early example of a shared memory, cache-based architecture without hardware coherence.

## 3 SOFTWARE CACHE COHERENCE

Previous work in software cache management proposes to reduce or eliminate entirely the need for hardware-coherence mechanisms [3, 4, 6]. A pure software-based approach requires the compiler and run-time system to explicitly address the problems of *stale data* and *false sharing* in order to generate deterministic programs. We describe these problems in greater detail, as well as the way in which our implementation of OSC addresses them, in the following subsections.

### 3.1 Stale Data

Stale data are copies of a data item that do not reflect their most current value. If a computation inadvertently uses a stale data item, all descendant computations are potentially invalid. To ensure data "freshness" without hardware support, a parallel program must execute cache *invalidate*, *post*, and *flush* operations to explicitly control the interaction of the local caches with shared memory.

*Post:* Data associated with an address is copied back to global memory. The processor's cache retains its copy of the data. A processor writing a shared data element into its cache must *post* it to memory some time before another processor attempts to access it.

*Invalidate:* Data associated with an address are marked invalid and the data are not copied back to global memory. The next reference for this address by the processor will be to global mem-

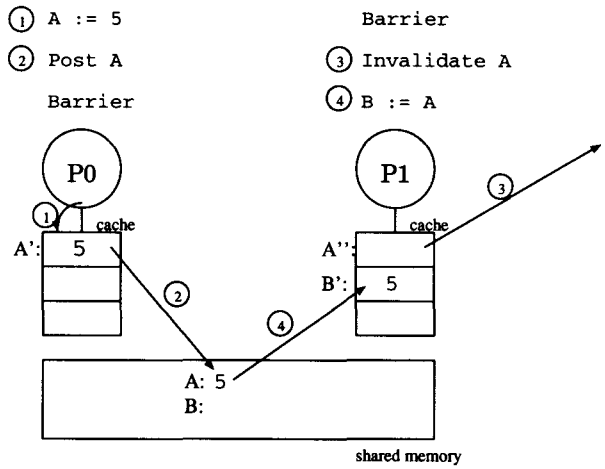



**FIGURE 1** Explicit cache synchronization.

ory. Any processor reading a memory location that has been updated by another processor must *invalidate* its own copy before reading. Otherwise, it may read a stale copy from its own cache and not the valid copy from shared memory.

*Flush:* The atomic combination of *invalidate* and *post.*

In Figure 1 we show the communication of a shared variable from one processor to another. The circled numbers show the order of execution and associated data movement with each instruction. Processor $P0$ assigns the value 5 into shared variable $A$. The value is written into $P0$'s local cached version of $A$ (labeled $A'$ in Fig. 1) as a result of the store. $P0$ then posts $A$ to memory, causing the data value cached therein to be copied to shared memory. $P0$ and $P1$ synchronize using a barrier so that $P1$ does not attempt to read the value of $A$ before $P0$ posts it. Before $P1$ reads the value for $A$, it must invalidate its cached copy (labeled $A''$) so that the read will come from memory and not its local cache. Note that this invalidate can take place any time before $P1$ attempts its read, although we show it colocated with the read itself. Finally, $P1$ stores the value fetched from $A$ into its local cached copy of $B$.

## 3.2 OSC and Stale Data

The current version of OSC implements a master/slave model of parallelism. All code except that implementing parallel loops is executed sequentially by the master thread. When the master reaches a parallel loop, it spawns slave tasks by writing an activation record (AR) into a predefined shared memory location for each slave. The AR describes all of the loop inputs, a loop body entry point, and an index range over which the slave is to execute. Upon receipt of an AR, a slave executes the loop body over the specified range and then enters a barrier waiting for the other slaves participating in the computation to complete. Once all slaves spawned by the master have entered the barrier, the master is free to proceed.

Sisal's strict functional semantics ensure that no communication will occur between slaves once they are activated. All loop inputs must be completely available before the slaves are spawned, and no loop output will be consumed until the master and the slaves synchronize at the end of the loop. On the IBM Power/4, both the post and invalidate instructions are combined into a single flush operation (implemented as an operating-system call). Therefore, to avoid stale-data accesses, the master must flush its cache before it spawns any parallel work, and each slave must flush its own cache before it enters the barrier at the end of a loop (see Fig. 2). The master flushes both posts any data it has written, and invalidates any cache entries for the memory that the slaves will write with the loop's outputs. Similarly, the slaves

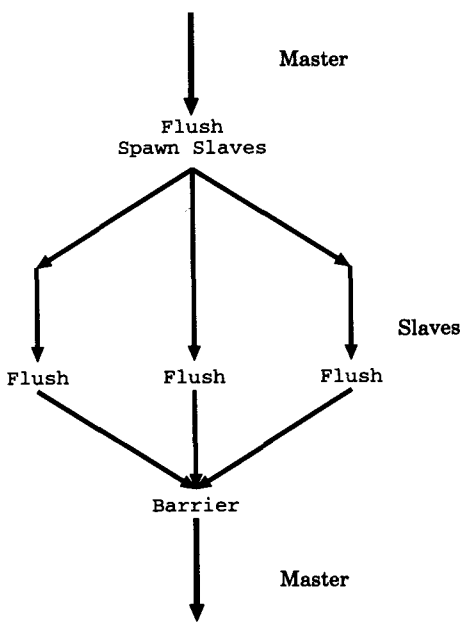



**FIGURE 2** Flush operations and the master/slave execution model.

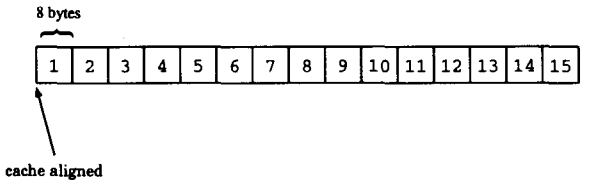



**FIGURE 3** Cache-aligned vector.

post their outputs and invalidate the cache entries corresponding to the memory that held the loop's inputs. By flushing the caches at the communication points, both master and slaves ensure that no stale-data accesses will occur. The functional semantics of Sisal make those communication points easy for the compiler to identify.

## 3.3 False Sharing

The unit of caching on the IBM Power/4 is a 64-byte cache line. When a processor accesses a memory location, the entire 64-byte cache line in which it resides is fetched into the processor's cache. If data items written by different processors are mapped to the same cache line, their accesses must be sequentialized. Otherwise, processors will update different copies of the same cache line. Since they are not updating the same memory location within the line (each memory location has a single writer in a correct parallel section), each processor will contain the cache line's original contents and its own updates, *but not the updates made by the other processors.* All of the cache line copies map to the same set of memory locations, so when the copies are flushed back to memory, only the last write prevails. We refer to this condition as *false sharing.*

For example, consider a parallel program executing on a system that uses 32-byte cache lines.* Assume that the program forms a contiguous vector of 15 double-precision floating-point numbers in parallel, using two processors, and that the first element of the memory allocated to hold the vector is cache aligned (Fig. 3). Note that in the figure, each double-precision number occupies 8 bytes, and that we label the elements 1 through 15, respectively. Assume further that the production of the loop has been partitioned so that processor $P0$ produces elements 1 through 7, and processors $P1$ produces elements 8 through 15 (Fig. 4a). In Figure 4b, we show the cache-line partitioning imposed by the hardware. A cache line size of 32 bytes is large enough to hold four double-precision vector elements. Since element 1 is cache aligned and the vector occupies contiguous memory, elements 5, 9, and 13 are also cache aligned. Note that the values in the cache line containing elements 5 through 8 are falsely shared between processors $P0$ and $P1$. When $P0$ produces elements 5 through 7 into its cache, the space for element 8 in the cache line will be left untouched. Similarly, $P1$ will produce element 8 into the rightmost slot of the cache line, leaving the slots for elements 5 through 7 untouched. The values of the untouched elements are undefined. In practice, however, they will contain whatever random data happened to be in the memory locations corresponding to the cache line before the first element is produced. After each processor produces the elements it has been assigned, it must post the values to memory. However, the hardware will post the entire cache line as a unit, thereby writing undefined values into the vector. Figure 5 depicts the cache lines and memory for the shared line just before the post. The *X*s in Figure 5 represent undefined values. Note that the result is nondeterministic, with the last temporal post taking precedence. If $P0$ posts first, then the undefined values from $P1$'s copy of the cache line will overwrite elements 5 through 7. Otherwise, $P1$ posts first, and the undefined value for element 8 in $P0$'s cache line will be written into the vector.

The short cache line containing elements 13 through 15 in Figure 4b may also create the possi-

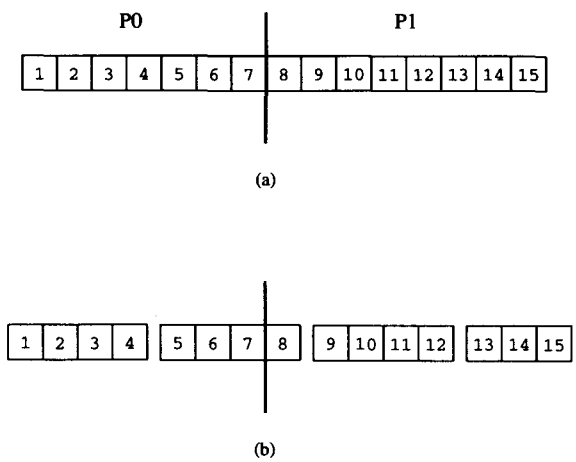



**FIGURE 4** Data and cache-line partitions.

* We use 32-byte cache lines in this example to make the explanations and the subsequent figures less complex.

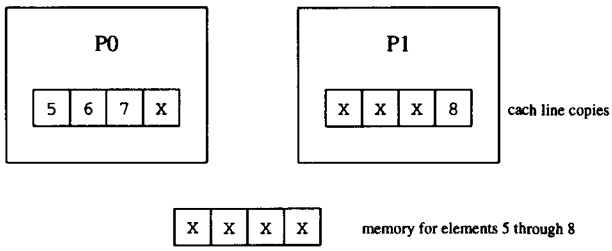



**FIGURE 5** Shared cache-line and memory prior to postoperation.

bility for false sharing. When the cache line is written to memory, the slot corresponding to what would be the 16th element will also be written. Since there is no 16th element in the vector produced by the computation, this slot will contain an undefined value with respect to the program. If another unrelated data structure happens to be contiguous with the vector, its first 8 bytes will be overwritten when the cache line is posted to memory. Again, in practice, the cache line will be fetched from memory before *P*1 fills in element 13 so that whatever bit pattern is present in the 16th element will be written back. If there are no processors updating the memory corresponding to the last slot in the cache line, the correct value will be posted back to the memory, and the program is correct. In general, however, that memory may also be updated in parallel since it is potentially used by an independent data structure.

## 3.4 Padding and Tessellation

The general solution to the problem of false sharing within a parallel section of code requires that

1. The memory used to implement all data structures is an integral number of cache lines.
2. No two processors share a cache line in parallel.

When implemented by a compiler and run-time system, the first requirement translates to *padding* in any memory allocation. For both imperative and functional languages, statically defined data structures can be easily padded. However, if the programmer is allowed to manipulate pointers to dynamically allocated memory, the language system cannot guarantee safe padding. Since functional languages deal with values (i.e., names and not memory locations), storage management is implicit and completely under the control of the compiler and run-time software.† So they are good candidate languages for efficient implementation on parallel architectures without cache coherence.

To satisfy the second requirement, the program partitioner must understand the mapping between logical data structures and the memory that implements them. In particular, the partitioner must *tessellate* each shared data structure with an integral number of cache lines. If the programmer is allowed to specify a partition that does not tessellate, the compiler and run-time must sequentialize all or part of the computation. However, the functional language compiler and run-time are free to coordinate memory alignment and partitioning to ensure tessellation.

## 3.5 OSC and False Sharing

We modified OSC to pad all data structures to an integral number of cache lines. We then changed the dynamic memory allocation system used by the run-time to allocate cache-aligned regions, and to round all allocation requests up to the nearest cache-line size. The result is that all statically and dynamically defined data structures are cache aligned and padded in the modified compiler.

As mentioned previously, the compiler crafts a set of activation records (one per active processor) in shared memory for each parallel loop. An activation record specifies a loop-body entry point, a list of inputs, and an index range. It is the index range that controls partitioning under OSC, as the functional semantics of the loop dictate that the computation associated with each index is independent.

To effect tessellation, we needed to change the AR generator to take into account cache alignment. Sisal is statically typed so the elemental data types within any aggregate (such as an array) are known at compile time. Using the example in Figure 3, the compiler knows *a priori* that the parallel loop will produce a vector of double-precision elements. The actual size of the array may not be known until run-time, hence the activation record is not crafted until the program actually executes. However, by knowing the cache-line size and the size of each element produced by the loop, the AR

† Advanced imperative languages like Modula-3 disallow pointer arithmetic, and so the false sharing problem is solvable within these languages as well.

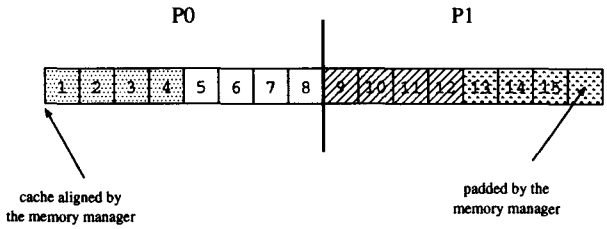



**FIGURE 6** Cache-aligned partition.

generator can calculate how many indices correspond to a single cache line. Once the number of processors and the total index range for the loop are known, each processor can be assigned an integral number of cache lines to produce.

Returning to the example shown in Figure 3, the compiler knows that four elements will fit in each cache line. When the AR generator is called at run-time, it is parameterized with this information, the total index range (1 through 15), and the processor ids (*P*0 and *P*1). It calculates that four cache lines are required to hold the 15 elements produced by the loop and splits the work evenly, two per processor. It then assigns indices 1 through 8 to processor *P*0, and 9 through 15 to processor *P*1. We show the resulting partition in Figure 6. Each cache line is differentiated by a different shading. Note that the memory manager ensures both the alignment of the first element, and that padding is inserted between the end of the vector and the next cache-line boundary in memory.

## 3.6 Interference with Other Optimizations

While the changes to the memory manager and AR generator were all that were necessary to effect cache-line tessellation in the general sense, OSC includes several other optimizations that potentially interfere with tessellation. In particular, loop fusion and storage preallocation cause difficulties that we are forced to address.

OSC attempts to fuse loops whenever possible, both to reduce the need for intermediate storage variables and to reduce overall loop overhead.‡ The result is that a single loop generates several output variables, each with a potentially different elemental type. For example, consider the fusion of the loop producing the vector shown in Figure 6 with one that produces a 15-element vector of 2-byte integers. Since both loops produce 15 elements, they can be fused into a single loop to save loop overhead. In Figure 7 we show both vectors with their respective data and cache partitions. Notice that all 15 two-byte integers will fit into a single cache line. Therefore, the loop that produces this vector cannot be parallelized if false sharing is to be avoided. In general, each loop producing more than one output must be partitioned according to the least common multiple among the elemental data types of its outputs. Since there is no possibility for memory aliasing and no implicit state in a functional language, each loop's outputs are unambiguous. Further, Sisal's strong typing makes the elemental data type known at compile time. The size, however, may not be known. For example, if a parallel loop produces a vector-of-vectors (which is the way two-dimensional arrays are represented in Sisal 1.2), the size of each inner vector may not be known until run-time. OSC implements such aggregates using pointers to non-contiguous storage. That is, the outer vector contains memory pointers, each referring to a different inner vector. If the production of the outer vector is parallelized, each loop body produces some number of inner vectors and returns pointers to them. The elemental data type for the outer vector is therefore a memory pointers, the size of which is known at compile time.

The other form of interference comes from the build-in-place optimizations specified in [7]. These optimizations will cause contiguous memory to be preallocated for data structures that are built separately and then concatenated. For example, if a vector is produced, and a border is then concatenated with either end of the vector, OSC will perform a single memory allocation for both the vector and the border. The loop producing the vector is then passed a memory pointer referring to the

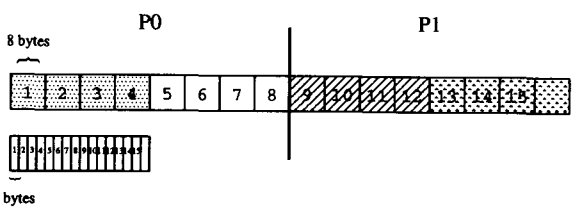



**FIGURE 7** Two vectors produced by the same loop.

‡ Parallel Sisal loops may return multiple values of different types. OSC will compute these values using a single-loop implementation (thereby fusing their production) by default. In version 12.9.1 of OSC, this default could not be overridden although the functionality should be part of future versions.

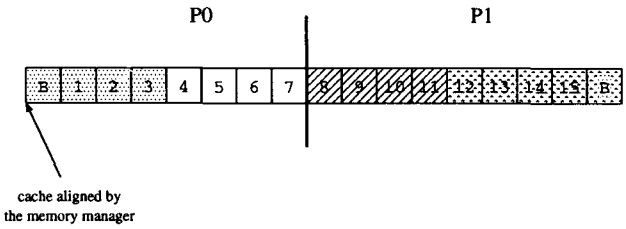



**FIGURE 8** Vector with border elements.

location within the allocated memory where the vector is to be stored. In Figure 8 we depict the 15-element vector from Figure 6 with a single border element on each end. Notice that the memory manager has aligned the storage for the vector, which in effect aligns the first border element and not the first data value. The AR generator, therefore, must consider offset from the beginning of a cache line to the location where the first data value is produced when it assigns an index range to each loop body. In Figure 6 where there is no border, *P*0 produces elements 1 through 8 since a cache line boundary falls between elements 8 and 9. With a cache-aligned border, however, the same cache-line boundary is shifted and subsequently falls between elements 7 and 8. The result is that *P*0 must produce elements 1 through 7, and *P*1 is assigned 8 through 15. In the examples, the load balance remains the same even though the partition changes. In general, however, cache aligning the border elements can cause a load imbalance. If, for example, two border elements were appended to the vector, *P*0 would produce 6 values, and *P*1 would produce 9, rather than the 7 and 8 split shown in Figure 8.

## 4 COMPILING IMPERATIVE LANGUAGES

Compiling imperative languages (such as Fortran and C) for the Power/4 is difficult. The designers wished to support shared memory parallel computing, but the task of correctly managing the per-processor caches is extremely complex and error prone. The system, therefore, includes parallelization tools for Fortran (C is not supported) that automatically insert cache-management system calls. In addition, any parallelized loop must be partitioned so that false sharing is avoided. Since memory can be aliased via the Fortran common block and unaligned addresses passed across subroutine boundaries, it is not generally possible to determine the alignment between arrays and memory at compile time. That is, it is not possible for the compile-time Fortran loop partitioner to determine the cache-line boundaries within an arbitrary array. Further, the dialect of Fortran 77 supported on the Power/4 allows dynamic allocation of memory at run-time. Since the alignment between array elements and cache lines will be set at run-time by the programmer, the parallelization tools must insert code to dynamically define the loop partition at run-time as well.

For example, assume that the vector shown in Figure 9 consists of 15 double-precision values, and that the target machine supports 32-byte cache lines. Further assume that the vector is defined as a parameter to the enclosing routine so that its first element is not cache aligned, and that four double-precision vector elements will fit in a 32-byte cache line. In Figure 9, shading shows the cache-line decomposition of the vector, with each shade corresponding to a different cache line. Notice that element 1 occupies the third element of the cache line that contains it. Since the vector may be embedded in some other array, the memory adjacent to either end must be treated as valid. In the example, data occupying the first and second cache line elements (each marked *X*) immediately adjacent to vector element 1 may be valid data from an enclosing array. Therefore, the compiler and run-time system cannot arbitrarily add padding to ensure cache alignment.

### 4.1 Parallelizing a Loop

The parallelization tools query the programmer for dependence information in the cases where static analysis fails. If the tool or the programmer determines that it is safe to parallelize a loop, code is inserted that automatically determines the alignment of the first element at run-time. All elements occupying partially filled cache lines are computed sequentially unless the tool can determine unambiguously that the falsely shared values are not updated. A parallel loop then computes all full cache lines, partitioning them evenly between the processors to avoid false sharing. In Figure 10, assume that three processors (*P*0, *P*1, and *P*2) are available. Elements 1, 2, and 15 are computed

**FIGURE 9** Noncache-aligned Fortran vector.

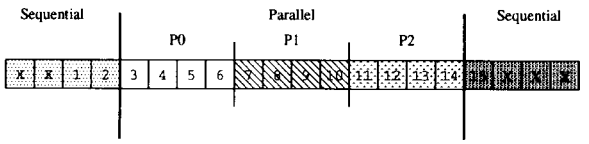



**FIGURE 10** Power/4 Fortran loop partition.

sequentially (in the case where the compilation system must be conservative) by the first loop on one of the processors. Then elements 3 through 14 are computed in parallel (one cache line per processor). While several optimizations to this general scheme are possible, the loop must still be partitioned at run-time as the compiler cannot know *a priori* what the memory alignment for the data will be.

## 4.2 Discussion

At the time of the Power/4's viability, the scheme outlined in the previous section was experimental and under development.§ Unfortunately, project constraints prevented us from testing equivalent Sisal and Fortran versions of the same program before the project's termination. We note, however, that because Fortran allows the programmer to dynamically associate array elements with the memory that holds them, there is no easy way to avoid the run-time overhead associated with dynamic loop partitioning. By way of contrast, the Sisal compiler and run-time system are completely responsible for memory management and loop partitioning. Sisal's semantics do not permit the programmer to specify how a particular data structure is implemented. In particular, it is not possible for a program to explicitly control how and what memory is allocated for each data structure. The compiler and run-time system, then, are free to force tessellation and padding of all data structures in a way that is completely transparent to the program, and which need not be analyzed at run-time.

# 5 RESULTS

We have limited results to report as our access to a Power/4 proved somewhat problematic. The machine on which we were able to develop our compiler was an early prototype used to test software upgrades and new products; however, we were able to validate the correctness of our compiler and evaluate its effectiveness. Achieving correctness demonstrates that a functional language system can automatically realize software coherence without programmer intervention and source code modification. To show the viability of our implementation, we present execution times for RICARD and SIMPLE, two scientific codes written in Sisal 1.2.

**Table 1. RICARD and SIMPLE**

| | Lines of Code | Floating-Point Operations |
|---|---|---|
| SIMPLE | 1550 | 119 052 334 |
| RICARD | 297 | 83 578 423 |

NOTE: SIMPLE is a Lagrangian hydrodynamics benchmark developed at the Lawrence Livermore National Laboratory which simulates the behavior of fluid in a sphere. The code is reasonably complex containing multiply fused loops, both with and without border constructions. RICARD is a production code developed at the University of Colorado Medical Center to simulate the elution patterns of proteins in a gel.

Table 1 details the number of lines and the floating-point operation counts for both codes. Both codes perform both double-precision floating-point arithmetic and integer calculations, but the integer operations are primarily for array indexing. Also, both codes define and read multidimensional arrays (implemented in OSC as vectors-of-pointers to vectors).

In Figure 11 and Figure 12 we show the execution time and speedup profile for RICARD and SIMPLE, respectively. Time, measured in seconds, is the processor time returned by the system. To compute speedup, we compiled each code for parallel execution and measured their execution times on one, two, three, and four processors, respectively. Each speedup value is calculated as the ratio of one-processor execution time to parallel execution time.

## 5.1 RICARD

The lack of a good speedup in Figure 11 (1.85 on four processors) comes from a poor load balance due to cache partitioning. In particular, OSC partitions only the outer loop of nested parallel constructs by default. The dominant loop of RICARD produces a two-dimensional array having only four

§ We base our understanding of this scheme on several conversations with the software developers. Other parallelization products were planned for this system, at the time the project was cancelled, which may have used different partitioning methods.

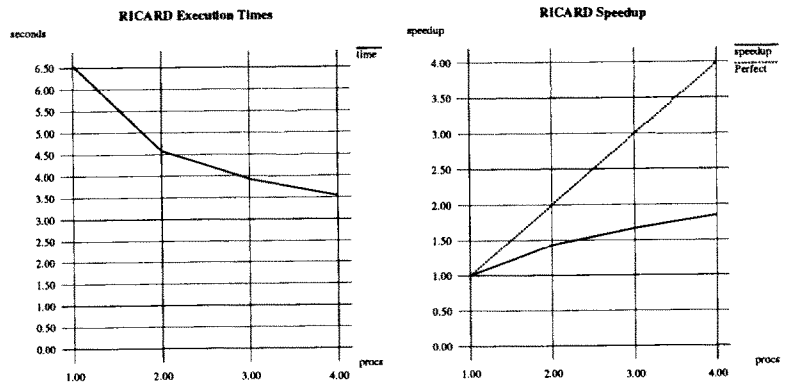



**FIGURE 11** Results: RICARD.

rows. Since the output from the loop is four memory pointers (each to a row vector), and all four row pointers will fit into a single cache line, that loop cannot be parallelized without producing a sparse outer dimension. We reran the program instructing the partitioner to attempt the parallelization of both inner and outer loops and the speedup improved: however, the overall execution time increased due to the control overhead associated with exploiting innermost loop parallelism.

## 5.2 SIMPLE

The speedup for SIMPLE is better than that for RICARD despite the load imbalance caused by fused loops and array border constructions. First, the problem size of SIMPLE is larger than RICARD and more symmetric: i.e., the iteration domains of the outer parallel loops are wide enough to provide cache-based parallelization. Second, the time-critical computations in SIMPLE are spread across many parallel loop nests (not just one narrow loop nest as in RICARD). Third, most of the parallel loops in SIMPLE define both vectors and vectors-of-vectors (pointers to rows of double-precision floating-point values and double-precision point values). On the Power/4, pointer values occupy 4 bytes and double-precision values occupy 8 bytes. If a parallel loop returns both, it must be partitioned for the least common multiple between the two (4 bytes), yielding a poor load balance for the other. However, most of the parallel loops in SIMPLE expend more computation defining the rows than the single elements. OSC partitioned each of these load-imbalanced loop forms for the pointers so that the load imbalance favors the production of the more expensive rows.

Neither SIMPLE nor RICARD achieve good absolute performance on the Power/4. SIMPLE executes in 17.52 s on four processors yielding 6.8 MFLOPS per second and RICARD's four-processor performance (3.54 s) is 23.6 MFLOPS per sec-

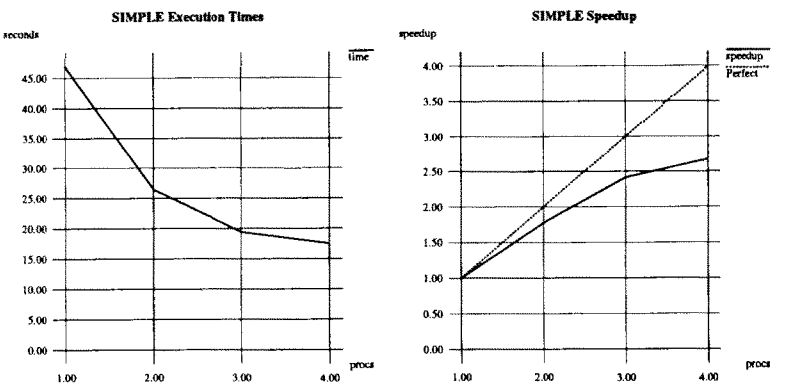



**FIGURE 12** Results: SIMPLE.

ond.‖ We believe that there are essentially two reasons, unrelated to the functional nature of Sisal, for these less than sterling performance numbers. First, the implementation does not take advantage of caching between parallel constructions. Like many scientific codes, SIMPLE and RICARD contain parallel loops that are repeatedly executed across time steps. Frequently, the arrays produced during one time step serve as the inputs to the next. However, the caches are flushed at the end of every parallel loop, so no cached values are carried over. It is possible to identify values that can remain cached across iterations, but the overhead associated with the flush operation for selected cache elements is much greater than that for flushing the entire cache. Therefore, unless all values could remain cached between iterations, the cost of selectively flushing a few would overshadow the benefit gained from caching. We conducted a few preliminary experiments and verified this hypothesis on the prototype machine. Notice that it is not the functional language but the hardware implementation of cache sychronization instructions that impairs performance. If the hardware implemented separate invalidate and flush operations as instructions (and not a single flush system call), we believe we could modify OSC to take advantage of caching. Slaves would *invalidate* inputs and *post* (without invalidating) outputs at the end of each loop. Subsequent parallel sections would then access valid data from their local caches if they were present.

The second performance problem is due to the overhead associated with locks. Each lock takes about 250 clock cycles to execute for reasons we were not able to discern. OSC implements storage reclamation via reference counting, which makes moderate use of the native lock primitives. Further, the mechanisms used within OSC to spawn nested parallelism require locks to manage the work queues for each slave task. The version of RICARD that uses nested parallelism incurs this overhead, while the more sequential version does not. The use of locks to implement memory management is a potential deficiency of OSC, however, it is not related to hardware cache coherence. Presumably, all parallel programs would suffer from expensive lock synchronization. At the time that we performed these experiments, we did not have access to any parallel Fortran or C equivalent codes so we could not test this hypothesis directly. We note, however, that the use of locks and reference counting does not hurt Sisal performance on other systems [2], so we do not believe it to be a serious liability.

In summary, Sisal and OSC enable the compiler to coordinate memory management and loop partitioning so that parallelism can be exploited. The interference with other optimizations introduced by cache tessellation does not seem to dramatically impact performance. The somewhat disappointing execution times, we believe, are due to inefficiencies in the base system and not the language model or implementation.

## 6 CONCLUSIONS AND FUTURE WORK

In this study, we investigated the use of a functional language as a vehicle for software cache management. We have implemented Sisal 1.2 (in the form of OSC) for the IBM Power/4, a machine that relies solely on the compiler and run-time system for cache coherence. The strict functional semantics of Sisal facilitate the elimination of stale data accesses. In addition, because the run-time system controls all memory allocations, the functional language system can coordinate memory alignment and partitioning to avoid false sharing. One drawback of the underlying system is the potential for poor load balance, especially in the presence of other optimizations such as loop fusion. However, our results show that the impact of such load imbalance is not substantial. While the speedup values we observe for SIMPLE and RICARD are reasonable, the absolute execution performance of SIMPLE in particular is somewhat low. We attribute much of the overhead in each execution to a lack of cache reuse and an excessive cost associated with locks.

We therefore conclude that functional languages provide good vehicles for software cache coherence. They shield the programmer from the problems of stale data and false sharing while exploiting parallelism automatically. Absolute performance, however, hinges on an efficient underlying hardware implementation. In particular, post and invalidate operations should be implemented as efficiently as possible, and flush by itself is not sufficient.

As part of our future work, we plan to investigate the tradeoff between cache tessellation and load balance on systems with cache-coherent hardware. We hope to develop a parameterized model

‖ Theoretical peak performance of the prototype is approximately 45 MFLOPS/s per processor, as the processors were "detuned" to 25 MHz.

to determine when the reduced contention due to cache partitioning will overshadow any subsequent loss of performance due to load imbalance. Also, we hope to study the efficacy of compiler-controlled cache *invalidate*, *post*, and *flush* operations as optimizations for hardware-coherent systems.

## ACKNOWLEDGMENTS

The authors thank the members of the Computer Research Group at Lawrence Livermore National Laboratory for their invaluable aid and insights. OSC and many Sisal programs are available via anonymous ftp from *sisal.llnl.gov* (128.115.19.65). Consult *http : //www.llnl.gov/sisal* for more information. Also, we thank Jim Van Fleet and the Advanced Workstation Group from IBM-Austin for their tireless assistance in this project. This work was supported in part by NSF grant ASC-9308900 and by DOE Contract W-7405-Eng-48.

## DISCLAIMER

This document was prepared as an account of work partially sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply endorsement, recommendations, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

## REFERENCES

[1] D. Cann, "Graph transformation algorithms for array memory optimization in applicative languages," Colorado State University, Fort Collins, CO, Tech. Rep. CS-89-108, May 1989.

[2] D. Cann, "Retire fortran? A debate rekindled," *Commun. ACM*, vol. 35, pp. 81–89, Aug. 1992.

[3] L. Choi, and P.-C. Yew, "A compiler-directed cache scheme with improved intertask locality," in *Proceedings of Supercomputing 1994*, 1994, pp. 773–782.

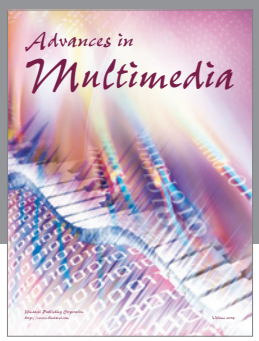
[4] R. Cytron, S. Karlovsky, and K. McAuliffe. "Automatic management of programmable caches," in *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988, pp. 229–238.

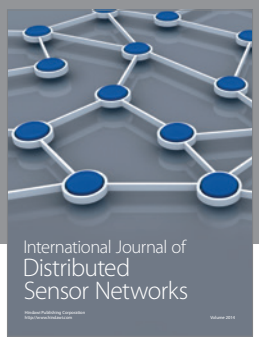
[5] Lawrence Livermore National Laboratory Computer Research Group, *Streams and Iterations in a Single Assignment Language, Version 1.2*. Livermore, CA, March 1985.

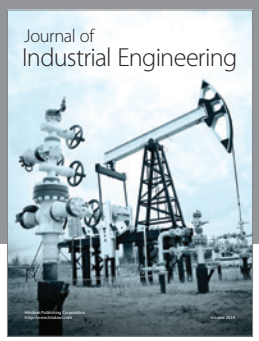
[6] T. Nguyen, F. Mounes-Toussi, D. Lilja, and Z. Li. "A compiler-assisted scheme for adaptive cache coherence enforcement," *IFIP Trans. A (Computer Sci. Technol.,)* A-50, pp. 69–78, Aug. 1994.

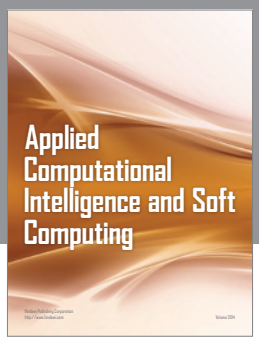
[7] J. Raneletti. "Graph transformation algorithms for array memory optimization in applicative languages," Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. UCRL-53832, Nov. 1987.

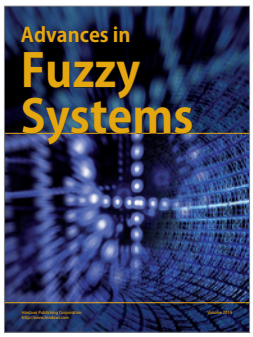
[8] W. Yen, W. Yen, and K.-S. Fu. "Data coherence problems in a multicache system," *IEEE Trans. Computers*, vol. 34, pp. 56–65, 1985.

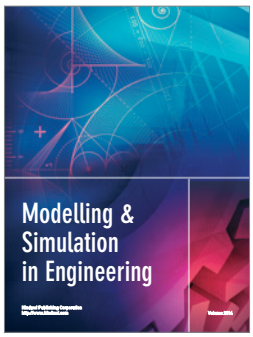


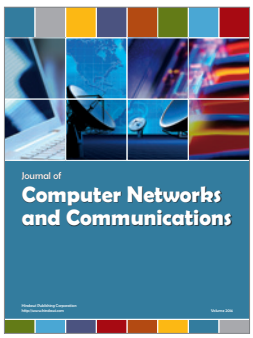












Hindawi
Submit your manuscripts at
http://www.hindawi.com