All Cache Virtual Machine: A Method for Improving the Cache Performance of Irregular Scientific Codes

Wesley K. Kaplow
Boleslaw K. Szymanski
Peter Tannenbaum
Department of Computer Science and Scientific Computation Research Center
Rensselaer Polytechnic Institute
Troy, N.Y. 12180-3590, USA

and

Viktor K. Decyk
Physics Department, UCLA
Los Angeles, CA. 90024
and
Jet Propulsion Laboratory/California Institute of Technology
Pasadena, CA. 91109

Abstract

In this paper we present the All Cache Virtual Machine (ACVM) method for improving the cache performance of irregular scientific codes. The method divides a real processor into a set of virtual machines based on the real processor cache size, and computation size and data reference pattern. Data references made to arrays distributed over ACVM's are rearranged to improve cache reuse. For dynamic irregular codes there is a runtime component that redistributes these data to maintain the cache reuse high. This run-time component is copied from the load balance code which is a part of the original program and which maintains data locality in the main memory for dynamic irregular parallel programs.

A novel set of compiler directives is introduced that can be used by the programmer to indicate to a compilation system the data structures to which the ACVM method should applied. A preliminary directive preprocessor is described. Results of the ACVM method are shown on Particle-In-Cell (PIC) codes that are used to simulate plasma physics, and for sparse matrix-vector multiplication which is at the heart of many iterative solvers of linear equations. Performance results for execution on several processor types are also presented.

1 Introduction

There is a growing realization that cache performance is one of the important components of overall parallel computation efficiency. Like with other components, e.g., non-local data transfer, the difficulty of ensuring high cache performance increases with the dynamic of the parallel code. The regular scientific codes, in which array references are affine functions of the loop control variables, can be optimized at compile-time, using, for example, analytical models of the cache [9, 19, 10]. The irregular computations in which array references use indirect indices can be categorized into two classes. One class, to which we refer as irregular static computation, have a constant data distribution among the processors. The techniques of the PARTI library [1], with its inspector and executor stages, are adequate to optimize the non-local data transfers and, through blocking (also referred to as tiling [23]) cache performance. The typical example of such computations are sparse matrix algorithms and codes operating on irregular meshes.

The second class includes dynamic irregular computations that have constantly changing data reference patterns. To minimize non-local memory data transfers, these computations include load-balancing procedures that redistribute the data which cause non-local data references. The main result, quite paradoxically, of this paper is that cache performance optimization in such a setting is simpler than in the case of sequential or irregular static codes. We demonstrate how, by using a small set of compiler directives, the programmer can employ load-balancing procedures to optimize the program's cache performance at run-time. The focal point of the needed transformation is the All Cache Virtual Machine (ACVM) defined and discussed below.

The Need for Cache Performance Optimization. The introduction of Reduced Instruction Set Computers (RISC) has provided a number of implementations that can deliver about 100 MFLOPs peak performance. Processors, such as the PowerPC, HP-PA, and i860 have been woven together with high speed communication fabrics to create supercomputers with potential performance of several hundreds of GFLOPS. However, there are two factors that define to what extent this performance is realized.

First, an application must be distributed over available processors and memory. For some problems the decomposition is uniform and static. Thus, the amount of work per processor is essentially constant and can therefore be statically partitioned. For others problems, such as the codes considered in this paper, the per processor work load changes during execution. Since the performance of a Single-Program Multiple-Data (SPMD) parallel program is determined by the most loaded processor, a load-balancing technique must be used to achieve high performance for such applications.

The second factor that limits the overall performance is the actual per processor performance, itself a target of many optimization techniques. These techniques can be grouped into two classes. The first class includes techniques that focus on the instruction execution sequence and functional units of the target processor. These optimizations address the pipeline nature of RISC processors, and attempt to generate code that keeps as many of the functional units as

busy as possible, while also reducing the number of pipeline bubbles due to conditional jumps. Some of the more important of these optimizations are *loop unrolling* and *load/store spreading.* The second class of optimizations focuses not so much on the sequence of operations, but on the data required by those optimizations. The sequence of data is important because of the difference between the processor's pipeline execution speed and the performance of the memory system. For example, the IBM SP2 parallel machine uses the same processor as the IBM RS/6000 model 590 workstation. Typical floating-point operations take 2-3 cycles, whereas it takes more than 10 cycles to read an individual word from memory.

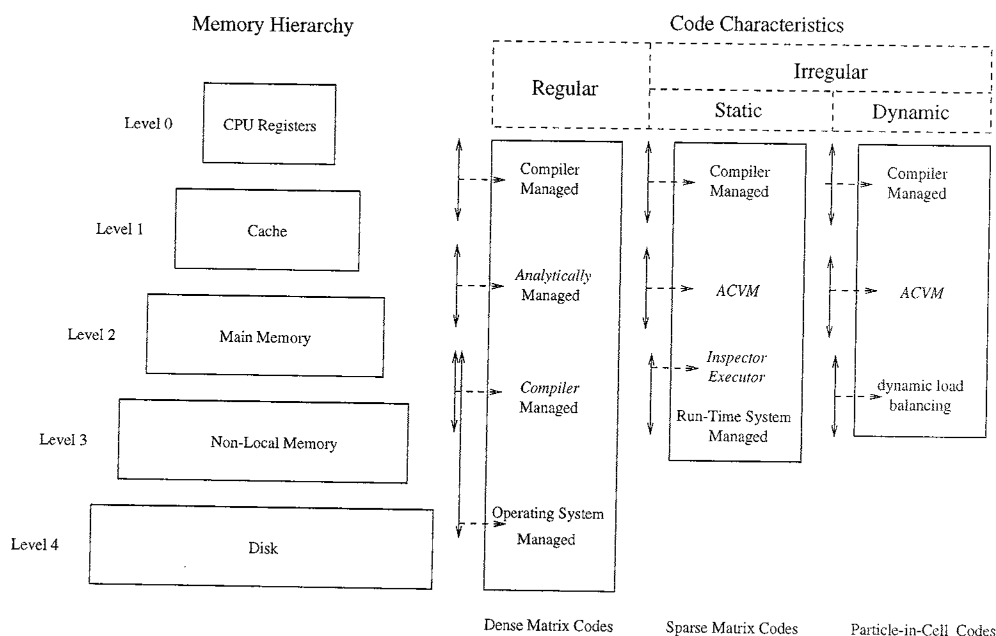

Figure 1: Control of the Memory Hierarchy: Convention vs. ACVM

Memory Hierarchy and Its Management. The overall memory hierarchy of a modern processor can be represented as a stack of progressively larger but slower in access memory memory components (see Figure 1). The first level of the hierarchy represents the processor registers. The allocation and use of these registers is controlled by the compilation system. In general, a compiler tries to make optimum use of the available registers to reduce the number of transfers between registers and memory. Cache memory represents the next level (level 1). This memory reduces the effective memory access time by introducing a fast, but limited in size, memory close to the processor that at any time holds a subset of the entire processor's address space. It is the *effective* access time, which is a function of the *hit-rate* in the cache, that determines how fast the processor can execute.

The main memory of a processor is the next level in the hierarchy (level 2). The interface

between the cache and main memory is managed by hardware. The data in the cache is a function of the cache architecture, which includes its size and replacement algorithm, as well as the sequence of data references generated by an executing program.

The next level in the hierarchy represents memory that is non-local to a processor. This level 3 memory could be located on another processor in a multi-processor system, or virtual memory located on disk. There are two principle access methods to manage the level 2 $\Leftrightarrow$ 3 interface in a multi-processor where each processor has a cache and local memory. In a shared memory architecture [15], the interface is managed by hardware which uses a network and distributed cache coherence method. Thus, a reference on one processor to memory on another processor proceeds as if the access were local with an additional latency. In a message-passing architecture, explicit messages must be sent between processors to move data. In general these messages are managed by the operating system with additional hardware to reduce message overhead. An important distinction between these architectures should be noted. A program that requires a non-local memory access can execute correctly on a shared memory machine because the hardware will automatically fetch the required data from the non-local memory to the local memory. However, on a message-passing machine the non-local memory is not visible and therefore a message-passing instruction must be inserted to obtain the required data.

Optimizations to Improve Memory Hierarchy Performance. Since the cache generally represents the fastest memory component (approximately 10 to 20 times faster than local main memory) optimizations have focused on increasing the hit-rate by increasing the re-use of data that is already in the cache. These optimizations generally operate on the loop nests of a program and modify the run-time reference pattern [21, 19, 23, 5]. The optimized codes operate on *blocks* of data at a time, each block being smaller than the cache size of the target processor. Note that these optimization can be applied only if run-time data references can be determined at compile-time, such as references to scalars and arrays indexed by *affine* functions of the loop control variables. We can also view these optimizations as managing, during the compilation phase, the interface between levels 1 and 2.

Unfortunately, many scientific programs have sections of code with references that are not determined until run-time. During compilation of such programs, it is impossible to determine a loop structure that will confine references to the current contents of the cache. It is exactly this type of problems that this paper addresses.

A *blocking* method, similar in spirit to array reference blocking, is given in [22] for irregular mesh and sparse matrix type problems. The authors determine the size of the local region corresponding to the size of cache using an analytical method, and use a domain decomposition scheme at run-time to reorganize data to fit into such local cache regions. They do not present a formal model of their operation, and examine only cache-coherent, shared-memory machines.

The All Cache Virtual Machine. In this paper we introduce a novel method of cache performance optimization for parallel programs based on an All Cache Virtual Machine (ACVM) concept. The ACVM method divides each real processor into a set of virtual processors, each with the same cache size as the original processor. The main loop of the user program is divided

into sections such that the data referenced in each section fit within the processor's cache. For static irregular codes, such as sparse matrix-vector multiplication, the sections and their sizes are determined statically. For dynamic scientific applications, such as Particle-In-Cell codes [4], the section's scopes can change. The run-time management of cache data reallocation among ACVMs is similar to local memory data reallocation done for load balancing through communication.

The ACVM model can be applied to both shared-memory as well as message-passing architectures. A set of directives and their preprocessor have been created to assist the programmer in automatically converting a program to take advantage of the ACVM method.

In Section 2 we describe the characteristics of the applications suitable for our method. Section 3 introduces the ACVM model and Section 4 describes the source-to-source compiler. In Section 5 we describe application of the ACVM method to Sparse Matrix-Vector Multiplication, and to a Plasma Simulation Code. Performance results are given in Section 6. Conclusions and final remarks are given in Section 7.

2 Target Application Characteristics

In this section we introduce type of programs that are amenable to the ACVM model. Those programs contain data references that cannot be determined at compile-time and thus are not candidates for static memory reference optimization techniques. However, as we will show, the use of compile-time cache performance prediction is important, as it is used to determine critical parameters that are used during run-time.

For the ACVM method to be applicable, the computation must involve the following generic operation:

1. There is a dense array of values V, a list of objects O, and an indirect array MAP that defines for each object $O[i]$ the index j to the associated value $V[j]$. The size of O (and MAP) is assumed to be much larger than the size of V (thus MAP is a *surjective* function).
2. There is either an associative and commutative operator $\oplus$ defining $V[j]$ that must be applied to all objects that map into the result j, or conversely an operator $\ominus$ that defines $O[i]$ as the function of the corresponding $V[j]$.

In other words either

$$R[j] = \oplus\{f(O[i]) | MAP[i] = j\}$$

where f is some function of an object $O[i]$, or

$$O[i] = g(R[j]), \text{ iff } MAP[i] = j$$

and g is a function of a value $V[j]$. Examples of such computations are matrix-vector multiplication and Particle-in-Cell plasma simulations. In the first case, the objects are non-zero matrix

elements and the mapping defines their column coordinates. V is a vector and the operation $\oplus$ is an addition. In plasma simulation, the objects are particles moving through the grid, V is the grid matrix and the mapping provides the spatial coordinates of each particle. In the so-called charge deposition stage the $\oplus$ operation is addition and in the particle push stage the $\ominus$ operator is the force computation (see details below).

A key characteristic of these problems is that there is opportunity for memory reuse (due to the difference in the sizes of the arrays V and O). Associativity and commutativity of the operations permit any order of traversal of the objects (i.e., the array O). Hence, there is an opportunity to arbitrarily arrange the sequence of references. This is exactly what the ACVM method relies on.

2.1 Static Irregular Codes

An example of a static irregular code is sparse matrix-vector multiplication. In many programs, such as finite-element solvers, sparse matrix-vector multiplication is a key algorithm that is used to iteratively solve a linear system. The references are static in the sense that the reference pattern is a function of the sparse matrix contents which do not change during iterations. Since dense matrix optimizations for improving cache performance (level $1 \Leftrightarrow 2$) rely on subscript analysis, the use of indirection in subscript expressions in sparse codes makes these methods useless. Compilation and run-time techniques for improving non-local reference access time for static irregular problems on distributed memory machines are based on the *inspector/executor* approach described in [3, 14]. However, these techniques address the $2 \Leftrightarrow 3$ interface, and are not intended to directly improve cache performance. Other techniques for improving the performance of sparse matrix-vector multiplication on parallel architectures have focused on improving processor partitioning to reduce remote processor communication costs [24], which is the level $2 \Leftrightarrow 3$ interface in our memory hierarchy.

Figure 2 shows the code for sparse matrix-vector multiplication taken from ITPACK [18]. In general, a multi-processor implementation of this algorithm would assign to each processor a contiguous number of rows of A,X, and W. Figure 3 shows the access pattern generated for each array during execution. If the dashed line in the figure represents a processor boundary, then the elements of X that fall below the line would have to be communicated to the top processor to calculate all the values for W.

Examining the level $1 \Leftrightarrow 2$ hierarchy, we can make the following observations. First, since each sparse row of A, is stored in contiguous addresses on each processor (in *VALUES*), its cache line reuse is optimal. Moreover, since the algorithm moves one row at a time, references to the W array will also be linear. However, access to the X array follows the sequence of non-zero elements in a row access; this is where there may be opportunity for managing the $1 \Leftrightarrow 2$ memory interface. The number of non-zeros is generally below a few precent of the total number of elements in the matrix. Therefore, accesses to X will be widely spaced. Unless all locally kept elements of X fit into cache, there will be no cache reuse on those references. The

X data locality can be improved by applying the All Cache Virtual Machine (ACVM) method to partition the rows on each processor.

```
1          SUBROUTINE PMULT(...)
2              ...
3          DO I=1,N
4              IBGN = COLPTR(I)
5              IEND = COLPTR(I+1)-1
6              SUM = 0.0
7              IF (IBGN .GT. IEND) GO TO 20
8                  DO J = IBGN,IEND
9                      JAJJ = ROWIND(J)
10                     SUM = SUM + VALUES(J)*X(JAJJ)
11                 ENDDO
12  20         W(I) = SUM
13         ENDDO
14         RETURN
```

Figure 2: ITPACK Matrix-Vector Multiplication Code

2.2 Dynamic Irregular Codes

Dynamic Irregular codes are those in which the reference pattern is created by indirect array references which change during program execution. The example in Figure 4 is a typical template for sections of Particle-in-Cell codes. The plasma Particle In Cell simulation model [7] integrates in time the trajectories of millions of charged particles in their self-consistent electromagnetic fields. Particle interactions are not modeled directly, but through the fields which they produce. Particles can be located anywhere in the spatial domain; however, the field quantities are calculated on a fixed grid. In the application discussed here only the electrostatic (coulomb) interactions are included.

The General Concurrent Particle in Cell (GCPIC) Algorithm [20] partitions the particles and grid points among the processors of the MIMD (multiple-instruction, multiple-data) distributed-memory machine. The particles are evenly distributed among processors in the primary decomposition, which makes advancing particle positions and velocities in space efficient. A secondary decomposition partitions the simulation space evenly among processors, which makes solving the field equations on the grid efficient. As particles move among partitioned regions, they are passed to the processor responsible for the new region.

FX

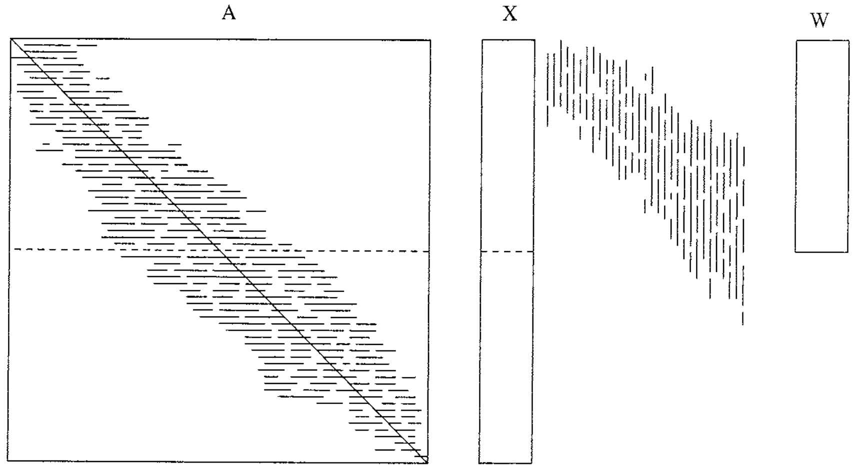

Figure 3: ITPACK Sparse Matrix-Vector Multiplication Traversal Pattern

There are three principle arrays in the PIC code in Figure 4. The array *PART* is used to store the *X* and *Y* positions of all of the simulated particles (there are *NOP* total particles). The *FX* and *FY* arrays represent the cells of the simulation and a physical parameter associated with each cell. In a 2D simulation these arrays are two dimensional, and each represents the electric field in one of the dimensions. The operation of the code is generally as follows. Each particle is mapped to a cell. A function is applied to the information at the mapped cell, and the results are used to update each particle's data.

This computation is repeated at the *particle push* and *charge deposit* stages of PIC codes and it consumes the majority of the execution time of a simulation. The order of traversal of the *PART* array is not important, in that all orders of traversal are mathematically equivalent in the simulation. It is this freedom that the ACVM technique exploits.

The performance problem with the code template above is that the references to the *FX* and *FY* arrays are dependent on the run-time values of the *PART* array. *PART* represents the locations of the particles that are simulated. Each particle moves during the simulation and therefore the references to the *FX* and *FY* arrays will, in general, constantly change. This particle motion eliminates locality, driving down the cache hit-rate. In [8] the authors show that the performance of sorted particles can be as much as 70% higher than when the particle positions are randomized.

Their performance improvement lead us to the hypothesis that the reference order into the *FX* and *FY* arrays could be managed in such a way as to maintain locality. One obvious way to to this is to sort the particles on each processor to maintain the sorted order. This is at best an $O(n \log n)$ operation for particles. However, as we show below, this is not necessary. The references that the particle positions represent only need to be rearranged enough to maintain

```
1          SUBROUTINE PUSH(PART, FX, FY, NX, NY, NOP)
2          DIMENSION FX(NX, NY), FY(NX, NY)
3          DIMENSION PART(2, NOP)
4
5          DO 10 I=1,NOP
6          NN = PART(1,J) + 0.5
7          MM = PART(2,J) + 0.5
8
9          DX = FUNCA(FX(NN,MM), FX(NN+1,MM), FX(NN-1,MM),
10              (NN,MM+1),FX(NN+1,MM+1),FX(NN-1,MM+1),
11              FX(NN,MM-1),FX(NN+1,MM-1),FX(NN-1,MM-1))
12
13         DY = FUNCA(FY(NN,MM), FY(NN+1,MM), FY(NN-1,MM),
14              FY(NN,MM+1),FY(NN+1,MM+1),FY(NN-1,MM+1),
15              FY(NN,MM-1),FY(NN+1,MM-1),FY(NN-1,MM-1))
16
17         PART(1,J) = FUNCB(DX)
18         PART(2,J) = FUNCC(DY)
19   10    CONTINUE
```

Figure 4: Example Code Template Amenable to ACVM Application

locality within a set of All Cache Virtual Machines (ACVM) described in the next section.

3 All Cache Virtual Machine Model

This section introduces the All Cache Virtual Machine (ACVM) model. We start by introducing a formal model for ACVM and outlining the compile-time and run-time issues associated with the model. Next, we show how the ACVM model applies iteration blocking to irregular codes. Finally, we show the relationship of the ACVM model to current data assignment techniques that are used to distribute data over a processor set, and the run-time management required.

3.1 Formal Model

Consider a computation that operates on a dense array of values V, a list of objects O, and an indirect array MAP that, for each object $O[i]$, defines the index j into the associated value $V[j]$. We assume that the size of array O (and MAP) is much larger than the size of array V. We assume also that the majority of the execution time is spent operating on those data structures either (i) applying an associative and commutative operator $\oplus$ over objects and assigning the result to the corresponding elements of V, or conversely (ii) for each object in O computing a function over the corresponding element of V. The target machine is described by the set of cache parameters that are critical in determining the performance of the application code. This set of parameters is referred to as *CacheParams*. Finally *AppCode* represents the section of code that implements operations on arrays V and O.

Definition 3.1 *An ACVM is defined as a tripple of functions:*

1. $CacheMap(V, MAP, CacheParams, AppCode)$: *is a mapping that divides the processor allocated arrays among the several virtual machines based on the size of the cache and application code.*

2. $VirtMap(CacheMap, OpReq)$: *is a function that returns the virtual machine identifier for an operation request.*

3. $SendOpReq(OpReq, ACVM_i, ACVM_j)$: *is an operation that moves the OpReq from the ith to the jth ACVM.*

The creation of the ACVMs on a processor requires a compile-time determination of the ranges of the arrays that should be allocated to a virtual machine. In reality, the arrays are not reallocated, but their statically determined size defines the $VirtMap$ function. Figure 5 shows an example of the ACVM model. Without loss of generality, the figure shows two processors. Each processor has a set of allocated arrays, and a list of operation requests. In this example the operation requests use particle lists typical of a PIC application. In the figure, the particle lists implicitly imply references to array elements. In general, the reference order into the arrays will be random. Below the *real* processors is the set of ACVMs. Each virtual machine contains a section of the arrays, plus the operation requests *that are local to that section.* By construction, the array allocated to a virtual machine, and the operation requests (which again may be random in their implicit set of references in the arrays), will provide good cache performance.

The key feature of the ACVMs is that the operation request lists contain references local to the arrays allocated to them. This ensures good cache performance. However, since we are dealing with dynamic codes, the operation request's effective addresses (in the case of PIC code, their location) will change during execution. Thus, there is a need to maintain the order of the lists.

An important observation can be made from Figure 5. Toward the top of the figure there is an operation request (particle list) which now, because of the execution of the simulation, references array locations that are on another processor. These references (labeled A in the figure), are typical of multi-processor simulations. In general, a message will be created that moves this operation request (particle) to the list on the appropriate processor. A completely analogous situation exists on the ACVM for virtual processor 0. The operation requests labeled B represent requests that have moved, due to the execution of the simulation, to another virtual machine. In exactly the same way as above a message is sent from the first ACVM to the second ACVM. However, the reasons for A and B are different. In the case of a message-passing multicomputer, request A must be sent because that is where the data is. Request B is moved to another ACVM because we wish to maintain the locality of reference in each ACVM. This communication ensures that the all of the operation requests of the first ACVM are local to its allocated arrays (and therefore the cache).

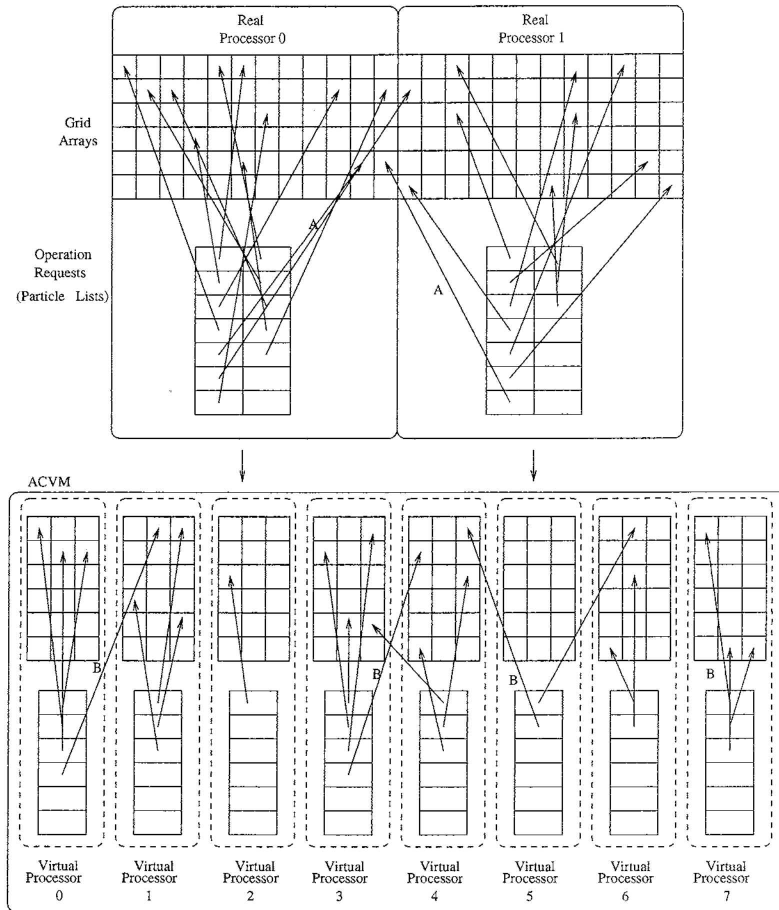

Figure 5: All Cache Virtual Machine

3.1.1 Relationship To Traditional Blocking Algorithms

The goal of ACVM is to reduce the execution time of irregular codes by improving processor cache performance. To do this, ACVM performs a function that is similar to *blocking.* Blocking, also called *tiling* is a technique that has been applied to scientific problems to improve memory hierarchy utilization [21, 19, 23, 16]. For regular programs such as dense matrix multiplication, or Jacobi iteration, blocking is done by partitioning the matrices into rectangular blocks, and modifying the code to operate on a dingle submatrix at a time instead of on the entire matrix.

In Figure 6 the transformation from the *real processor data distribution* to *ACVM distribution* is an example of a blocking optimization. This type of blocking can be used to improve the performance of the simple Jacobi iteration kernel. In this case, the reference pattern is

known at compile time, so the iteration space can be transformed into a blocked pattern by a code transformation that changes the iteration pattern from a sweep of the entire array to a successive sweep of the blocks. The blocking is shown in the figure as the transformation of the data distributed onto each real processor to contain nine submatricies. Figure 7 shows the performance improvement due to blocking on a single node of an IBM SP1 processor.

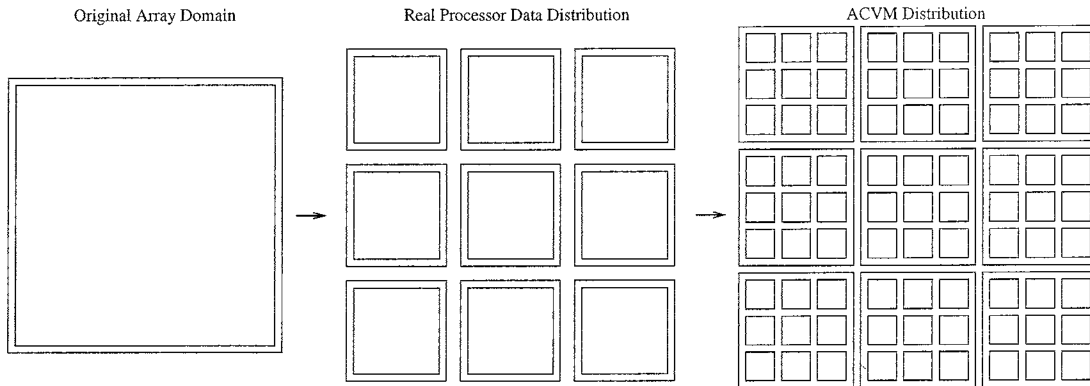

Figure 6: Multi-processor Data Distribution and Blocking

Blocking for the ACVM goes one step further, subdividing the problem on each processor. In this way the ACVM model can be seen as a generalization of dense code blocking in that run-time methods, as opposed to compile-time loop structure modification, are used to create and maintain the block order of references.

An important issue is the determination of the block size. This depends on the algorithm's data reference pattern and on the architecture of the cache. For regular codes, there are several methodologies which range from analytical [19, 10] to run-time [12, 13] to execution simulation [17]. This issue will be discussed more fully in a following section.

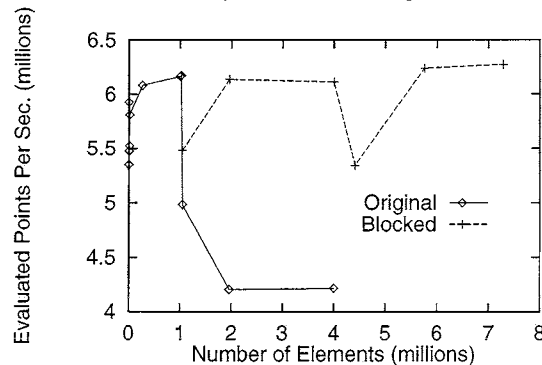

Figure 7: Performance Improvement Due to Blocking

3.1.2 Relationship to Multiprocessor Data Distribution

The creation of ACVMs is similar to the data distribution used in SPMD parallelizations of code. The data distribution among ACVMs, as suggested by Figure 6, can be seen as an additional block distribution of the data assigned to a real processor.

As in the SPMD case, ACVMs can conceptually execute in parallel, thus providing additional threads of execution if the target architecture can support multithreading. This mode is limited by the nature of the ACVM run-time discussed later.

3.2 Compile-Time Determination of ACVM Size

There are several methods that can be used to determine the size of the arrays allocated to the virtual processors. An *execution* or *benchmark* method can be used, or a cache performance estimation technique can be employed.

In the benchmark case, a series of executions are performed to experimentally determine the best size of the virtual machine. For best results the benchmark is actual code extracted from the application. The total size of the array that will be allocated to each processor (if there are more than one) is used as the real machine grid size. Next, the optimal range values (representing each dimension of the allocated arrays) are selected. Those ranges are used to define the locations in the array that refer to the current contents of cache. An important issue is that each of the virtual machines will represent a contiguous section of the arrays, but these sections will not in general be contiguous in memory. Thus, it is important for this, and other cache performance prediction methods, to determine the performance (and, indirectly, the cache performance) by executing from within the size, or close to the size of the arrays that are representative of the real execution.

Another method uses compile-time execution to determine the size of the virtual machine. This method, described in [16, 17], can be used to determine quickly the optimal range values by partial simulated execution of the application code and an architecturally correct simulation of the target processor's cache.

Once the range values are determined, then the functions $CacheMap$ and $VirtMap$ can be fully defined.

3.3 Run-Time Component

There are several operations that must be performed at run-time in the ACVM model:

1. Each ACVM must be scheduled to execute.

2. Operation requests must be obtained from an ACVM's list.

3. Operation requests must be determined to be either local to the currently scheduled ACVM or *non-local.*

4. A method to send an operation request from one ACVM to another must be provided.

Since the driving force of the ACVM is to manage the cache contents to improve cache performance, the overhead of implementing each of the above operations must be small enough so that it does not negate the performance improvement due to better cache hit-rates.

A template for implementing the ACVM on a single processor is shown in Figure 8. In this template the scheduling function is provided by a simple loop structure. A loop also returns each operation request from the current ACVM. The request is determined to be local or non-local based on the $VirtMap$ function. If it is local then processing continues. If it is non-local (*i.e.*, it maps to ACVM *id*, where *id* is not the current ACVM) then the operation request is sent to the correct ACVM's *OpReq* list using the *SendOpReq* function.

imim

```
1  for ACVM_i ∈ {ACVM} {
2      for req ∈ {OpReq(ACVM_i)} {
3          req_new =function (Grid, req)
4          dest = VirtMap(CacheMap(Grid, Cache, AppCode), req_new)
5          if dest ≠ i {
6              SendOpReq(req_new, ACVM_i, ACVM_dest)
7          }
8      }
9  }
10 dosendreq({ACVM})
```

Figure 8: ACVM Run-Time Execution Template

One of the problems with the above code template is that it tests every operation request to see if it has moved to another ACVM. Since we are dealing with the statistical nature of the cache, it is clear that some number of non-local ACVM references will not significantly impact the performance of the ACVM. Therefore, we can define two types of run-time systems:

Definition 3.2 *An ACVM run-time system is* strict *if during any ACVM execution, all non-local references are sent to their proper ACVM.*

Definition 3.3 *An ACVM run-time system is* non-strict *if during an ACVM execution some of the executed operation requests are non-local.*

The *strict* run-time incurs a cost to manage the operation request lists during every execution. This may be desirable if the *SendOpReq* function, the conditional test required, and the list management times are small, and the number of off-ACVM requests are expected to be few during the computation. However, a *non-strict* method could be employed following one of these possible schemes:

1. Count non-local ACVM requests. When a threshold is exceeded, a redistribution of the operation requests would be made to restore ACVM locality.

2. A periodic reordering could be used where the operation request lists are changed every n virtual machine executions.

For a multi-processor implementation, the operation requests may refer to data that are assigned to another real processor. Therefore, a check must be made to determine the real processor to which the operation request must be sent. This is completely analogous to the movement of operation requests between ACVMs on the the same processor.

Shared-memory multiprocessor implementations of these types of codes can follow strict or non-strict movement of operation requests between processors. In this case the movement of operation requests between processors, like between ACVMs, is not required for correctness, but is performed to balance the load among the processor set. The situation is similar for distributed memory message-passing multiprocessors. However, in this case, operation requests must be strict because there is no direct access to the required array information—the required information is located on another processor. Figure 9 shows a template for message passing architectures. The *processor* function is a mapping that returns identity of the real processor onto which an ACVM is mapped. The procedure *LocalSend* uses memory copies and local management of the ACVM reference request lists, and *MessageSend* forms inter-processor messages.

```
1  SendOpReq(req, ACVM_i, ACVM_j)
2  {
3     if processor(ACVM_i) = processor(ACVM_j) {
4        LocalSend(req, ACVM_i, ACVM_j)
5     }
6     else {
7        MessageSend(req, processor(ACVM_i), processor(ACVM_j))
8     }
7  }
```

Figure 9: ACVM SendOpReq for Message Passing Architectures

4 Implementation in a Compilation System

The incorporation of the ACVM model into a compilation system requires certain information about the data structures and their use in the transformed program. First, the programmer

must identify which of the various arrays in the program represent the set of values V and objects O, and which function represents the mapping MAP. Additionally, a correspondence of the indexes of those arrays must be established (in case the array V is multidimensional). To this end, we have introduced several static and run-time compiler directives similar to those defined in High-Performance Fortran [11].

To convert a program to ACVM form, the compiler must track the definition and use of the object array O. Since the data contained in this array defines the pattern of references into the array V, this is the array that must be run-time managed. If the compiler is unable to determine where in the code the values in the reference array are modified, then a run-time compilation directive must be inserted to indicate where the reference request management run-time component needs to be added.

virtualized

```
1          PARAMETER(NX=256,NY=512,IDIMP=4,NOP=256000)
2          DIMENSION PART(IDIMP,nop)
3    ACVM$ VIRTUAL PROCESSORS mymachine(SP2) (One of SP2, SPARC, T3D)
4    ACVM$ TEMPLATE grid(NX,NY)
5    ACVM$ ALIGN q, fx, fy WITH grid
6    ACVM$ DISTRIBUTE grid(mymachine)
7    ACVM$ MAP grid(part(1,*),part(2,*))
8    ACVM$ METHOD INSORT (One of EXSORT, INSORT, SEND)
```

Figure 10: ACVM Static Compiler Directives

Figure 10 shows the static directives used by the programmer to direct the ACVM compilation process. Lines 1 and 2 in the figure show conventional FORTRAN used to define some compile-time parameters and allocate an array. The ACVM specific directives in lines 3 through 8 are used to define the data structures and method used for ACVM application. The `VIRTUAL PROCESSORS` directive defines the class of ACVM machines. In this particular case, the ACVM is an IBM SP2 processor. The size of the arrays will therefore correspond to the cache size of the SP2. The `TEMPLATE` directive, similar to the corresponding HPF directive, defines an array shape and gives it a name without actually allocating storage. The `ALIGN` directive, although similar to the corresponding HPF directive, does not allocate storage, but indicates that the `q`, `fx`, and `fy` arrays all have the same alignment, and that these are the arrays that we want to distribute to the created ACVMs. The `DISTRIBUTE` directive is used to indicate which template class is to be distributed among ACVMs as defined in line 3. Line 7 is extremely important as this is where the connection between the object, in this case stored in `part`, and the values, in this case represented `grid` is made. The `MAP` directive tells the compiler that the first index of arrays aligned with `grid` is indexed by the first column of the first index of the `part` array. The second index of `grid` is indexed by the second second column of the same index of `part`.

This indicates to the ACVM compiler that the references to the `grid` arrays are found in the array `part` and therefore these are the references that have to be run-time managed to enforce a reference order that meets the definition of ACVM.

The `METHOD` directive in line 8 defines the method used for run-time management. There are three options: `INSORT`, `EXSORT`, and `SEND`. The first two of these options tell the compiler to use a sorting method to maintain the ACVM order of references in the `part` array. This is very similar to Array Remapping [6]. The `EXSORT` method indicates that `grid` array will be sorted using an additional external array. Using the `INSORT` method, the source code will be modified such that the ACVM order is maintained using an internal sorting method. The last method is the most interesting because it is closest to the inter-processor communication used to implement multi-processor versions of SPMD programs. In the `SEND` method, the compiler uses an inter-ACVM communications primitive to *send* a reference from one ACVM to another ACVM when it moves during run-time.

```
1   ACVM$ SORT part
2   ACVM$ SEND part
3   ACVM$ WAIT part
```

Figure 11: ACVM Run-Time Indicator Compiler Directives

The set of directives shown in Figure 11 is used to help the ACVM compiler to determine where and which code to insert to manage the run-time changes to data in the distributed array. The `SORT` run-time directive identifies the point in the code where either the `EXSORT` or `INSORT` method should be applied. The `SEND` and `WAIT` directives are used for the `SEND` method. The `SEND` directive indicates where changes to the distributed array are made, and that a check should be made to see if a reference have moved from its current ACVM to another. The `WAIT` directive is used as a synchronization command. Since we may think of the execution of each ACVM as occuring in parallel, this directive must be used before the execution of another function that requires the distributed array's data. In many cases *definition-use chaining* [2] can automatically determine where this directive needs to be placed.

5 Application of ACVM

In this section we describe the application of the ACVM method on two examples: sparse matrix-vector multiplication (a static irregular code) and Particle-in-Cell simulation (a dynamic irregular code).

5.1 Sparse Matrix-Vector Multiplication

Figure 12 shows the ACVM transformed sparse matrix-vector multiplication. Lines 3 through 8 show the ACVM compiler directives used to transform the code. The `VIRTUAL PROCESSORS` directive indicates that the target architecture is an SP2 node. Directives 3,5, and 6 indicate that it is references to the `VALUE` array that are to be put into ACVMs. Line 7 associates the `COLPTR` array with the arrays that it references. Finally, the `INSORT` method indicates that an internal sorting method is used to change the order of references to move each into its correct ACVM. Because of the data structure used to store the sparse matrix, a function must be supplied by the user to enable the sorting function to move references in the original `COLPTR` array, into `COLPTRA` which defines the ACVMs.

5.2 Particle-in-Cell Codes

In this section we show the application of the ACVM method to Particle-in-Cell code. Figure 13 shows the transformation of the code fragment from Figure 4. The ACVM compiler directives in lines 3 through 8 are the directives used to generate the subsequent code. Lines 10 through 15 represent parameters that define the characteristics of each ACVM. Line 10 defines the size of each ACVM. In line 11 the number of ACVMs in each dimension of `PART` is defined. These are used to determine the ACVM for each reference request. The total number of ACVMs is defined in line 12. Since the total number of operation requests are distributed across the total number of ACVMs, line 13 defines the maximum number of reference requests per ACVM. This number is increased by 30% to account for request migration from ACVM to ACVM. Line 14 defines the maximum number of send requests that are allowed. Finally, the frequency of the ACVM run-time operation is defined in line 15.

The run-time requires some additional arrays to support data distribution and the `SEND` method. The particle array, defined in line 7, is modified to have an additional dimension which represents the distribution among ACVMs. The size of the distributed array is changed to `mxperacvm`. The additional arrays define the number of reference requests in each ACVM (`npacvm`), the list of references that are to be moved to another ACVM (`sendlist`), the destination of each send request (`destlist`), the number of requests to send (`npsend`), and finally a pointer to a list of empty locations in the PART array of each ACVM.

The executable code changes start in line 28. Code added or changed by the ACVM compiler are shown in lowercase. The execution of each ACVM is simulated by the `DO` loop from line 28 to 50. The loop iterates over all ACVMs. The references to the array `PART` are changed to reflect the data distribution by adding an index for this loop. There are no other changes to the code until, by *definition-use chaining*, it is determined that the value of one of the components of `PART` is changed. When this occurs, the compiler adds an additional variable to capture the new value. In this case line 41 saves the value of the first *mapped* index and line 43 saves the value of the second mapped index. Line 45 checks if the run-time functions must be executed in

```
1           PARAMETER(mxvalues=1000000,nrows=5000,ncols=5000,ncolsp1=ncols+1)
2 C         DIMENSION COLPTR(ncolsp1)
3 C  ACVM$  VIRTUAL PROCESSORS mymachine(SP2)
4 C  ACVM$  TEMPLATE grid(mxvalues)
5 C  ACVM$  ALIGN VALUES WITH grid
6 C  ACVM$  DISTRIBUTE grid(mymachine)
7 C  ACVM$  MAP grid(COLPTR(*))
8 C  ACVM$  METHOD INSORT
9           parameter(nsacvmx = 3000,nsacvmy = 5000)
10          parameter(ndacvmx = ncols/nsacvmx, ndacvmy = nrow/nsacvmy)
11          parameter(nacvm = ((ndacvmx) * (ndacvmy)))
12          parameter(ncolsa = ncols*nacvm)
13          DIMENSION COLPTRA(ncolsa)
14          DIMENSION ROWIND(mxvalues),VALUES(mxvalues)
15          DO IACVM = 1, NACVM
16              IBR = (IACVM-1)*(NCOL+1) + 1
17              IER = IBR + NCOL - 1
18              ICOL = 0
19              DO I = IBR, IER
20                  ICOL = ICOL + 1
21                  IBGN = COLPTRA(I)
22                  IEND = COLPTRA(I+1)-1
23                  SUM = 0.0D00
24                  IF (IBGN .LE. IEND) THEN
25                      DO J = IBGN,IEND
26                          JAJJ = ROWIND(J)
27                          SUM = SUM + VALUES(J)*X(JAJJ)
mmHp28                  ENDDO
29                  ENDIF
30                  W(ICOL) = W(ICOL) + SUM
31              ENDDO
32          ENDDO
33          DO I = 1,NCOL
34              X(I) = W(I)
35          ENDDO
```

Figure 12: Sparse Matrix-Vector Multiplication Source Code After ACVM Compilation

```
1             PARAMETER(NX=256,NY=512,IDIMP=4,NOP=256000)
2  C          DIMENSION PART(IDIMP,NOP)
3  C  ACVM$   VIRTUAL PROCESSORS mymachine(SP2)
4  C  ACVM$   TEMPLATE grid(NX,NY)
5  C  ACVM$   ALIGN q, fx, fy WITH grid
6  C  ACVM$   DISTRIBUTE grid(mymachine)
7  C  ACVM$   MAP grid(part(1,*),part(2,*))
8  C  ACVM$   METHOD SEND
9
10            parameter(nsacvmx = 64,nsacvmy = 64)
11            parameter(ndacvmx = nx/nsacvmx, ndacvmy = ny/nsacvmy)
12            parameter(nacvm = ((ndacvmx) * (ndacvmy)))
13            parameter(mxperacvm = NOP/nacvm*1.3)
14            parameter(mxsndacvm = NOP/nacvm*1.3)
15            parameter(nsratio = 10)
16
17            SUBROUTINE PUSH(PART, FX, FY, NX, NY, NOP,IDIMP,
18                 nacvm,mxperacvm,mxsndacvm,
19                 nsacvmx, nsacvmy, ndacvmx,
20                 sendlist, destlist, npsend, iphole
21            DIMENSION FX(NX, NY), FY(NX, NY)
22            DIMENSION PART(IDIMP, mxperacvm,nacvm)
23            dimension npacvm(nacvm)
24            dimension sendlist(idimp,mxsndacvm,nacvm)
25            dimension destlist(mxsndacvm,nacvm)
26            dimension npsend(nacvm),iphole(nacvm)
27
28            do 20 iacvm=1,nacvm
29            DO 10 I=1,npacvm(iacvm)
30            NN = PART(1,J,iacvm) + 0.5
31            MM = PART(2,J,iacvm) + 0.5
32
33            DX = FUNCA(FX(NN,MM), FX(NN+1,MM), FX(NN-1,MM),
34                 FX(NN,MM+1),FX(NN+1,MM+1),FX(NN-1,MM+1),
35                 FX(NN,MM-1),FX(NN+1,MM-1),FX(NN-1,MM-1)
36
37            = (FY(NN,MM), FY(NN+1,MM), FY(NN-1,MM),
38                 FY(NN,MM+1),FY(NN+1,MM+1),FY(NN-1,MM+1),
39                 FY(NN,MM-1),FY(NN+1,MM-1),FY(NN-1,MM-1)
40
41            col1 = FUNCB(DX)
42            PART(1,J,iacvm) = col1
43            col2 = FUNCC(DX)
44            PART(2,J,iacvm) = col2
45            if(freq(nsratio) .eq.  1) then
46                 ipdest = virtmap(col1,col2)
47                 if (ipdest .ne.  iacvm) then
48                      call send(J,IACVM,ipdest,...,PART,sendlist,
49                           destlist,npsend,iphole,...)
50                 endif
51            endif
52    10      CONTINUE
53    20      continue
54            call pdoreq(PART,npacvm,nacvm,sendlist,destlist,npsend,iphole,...)
```

Figure 13: Source Code After ACVM Compilation

this iteration. If the check is positive, the saved values are used by the run-time test in line 47 to determine if the reference request has moved to another ACVM. If the reference now belongs to a different ACVM, the `send` procedure is called to make a send request.

After all of the ACVMs execute, which is guaranteed due to scheduling via a loop, the send requests are executed by the `pdoreq` procedure call in line 54.

6 Results

In this section we present the performance results of applying the ACVM method to both sparse matrix-vector multiplication and to the particle-in-cell code described in the previous section.

6.1 Sparse Matrix-Vector Multiplication

As suggested in Figure 3, many problems that are solved using large sparse matrices have most of their elements located close the the diagonal. Therefore, in order to provide a realistic test of the ACVM modified matrix-vector multiplication, a sparse matrix generation program was created with the following user-defined parameters: the number of rows and columns, density, and variance. The density determines the number of non-zero elements in the array. The normal distribution is used to cluster non-zeroes around the diagonal with the given variance. Raising the variance widens the band.

Figure 14 shows the performance improvement of the ACVM method over the original matrix-vector multiplication on a Sun SPARC-20. The graph is plotted against the variance with the ACVM size fixed at $1,000$ columns. This ACVM size was picked because it provided the best performance improvement for this architecture. If the variance is too small, then there is little or no improvement because when all non-zero elements are close to the diagonal, they are already accessed in nearly ACVM order. As the variance increase the performance improvement grows to 32%.

In a parallel implementation of the sparse matrix-vector multiplication with n processors each processor would be allocated $\frac{1}{n}^{th}$ of rows of the matrix. Figure 15 shows the performance improvement on a SPARC-20 workstation executing as one node of a 16 processor execution of the sparse matrix multiplication on a matrix with a uniform density non-zeros of 2.5%.

In Figure 16 results for an IBM SP2 node are shown. In this case a diagonal sparse matrix is used and both the variance and ACVM size are varied. There are two important features in the data. First, as in the SPARC-20 results, the performance is proportional to the variance. Second, there is an optimal value for the ACVM size. If the size it too small then the extra overhead of managing the ACVM, due in this case to the sparse data storage method, causes a significant performance degradation. Also, if the size to too large, then the size of the locality that the ACVM defines is larger than the cache.

Given this data, it is possible to determine for a given matrix whether or not it will benefit from the application of the ACVM method.

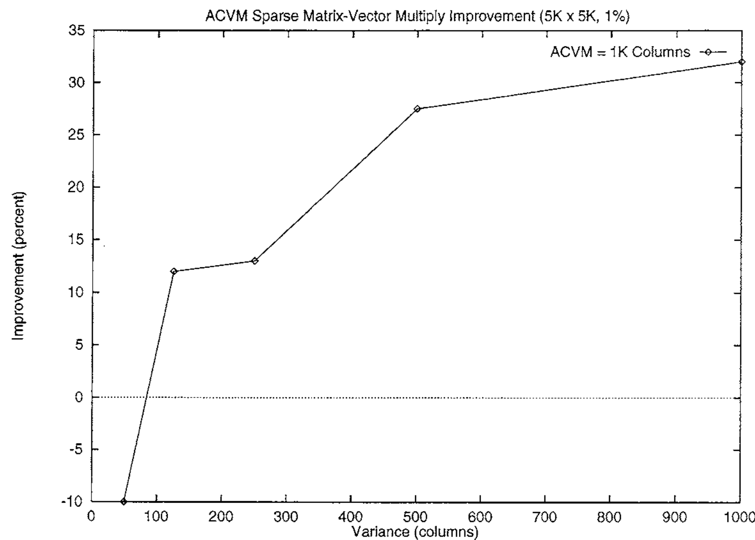

Figure 14: ACVM Performance Improvement vs. Distribution Variance on a SPARC-20 for Sparse Matrix-Vector Multiplication

In a parallel implementation of the sparse matrix-vector multiplication with n processors each processor would be allocated $\frac{1}{n}^{th}$ of rows of the matrix. Figure 17 shows the performance results for a single processor in this configuration. The total size of the matrix is $50,000x50,000$ with $1,250,000$ non-zeros. With 16 processors, 3125 rows are assigned to each processor.

Figure 17 shows the performance results for a single processor in this configuration. The total size of the matrix is $50,000x50,000$ with $1,250,000$ non-zeros. With 16 processors, 3125 rows are assigned to each processor. The results show, similar to the uniprocessor case in Figure 16, that there is no performance increase if the data is grouped too closely.

6.2 Particle-in-Cell

In this section we present performance results of applying the ACVM method to Particle-in-Cell (PIC) Plasma simulation codes. The x-axis of all of the graphs represents the iteration step of the simulation. The y-axis represents either the time of the particle push and deposit charge routine, or a ratio indicating performance improvement. Three processor architectures are used to show the application of the method: the Sun SuperSPARC processor, a Cray T3D node, and an IBM SP2 node.

Figures 18 and 19 show a performance comparison between the original PIC code and ACVM optimized code. The PIC code example included a `grid` array of size 64×128 and `part` array with 327680 entries. The ACVM machine size is 32×32, and therefore in this case there are eight ACVMs.

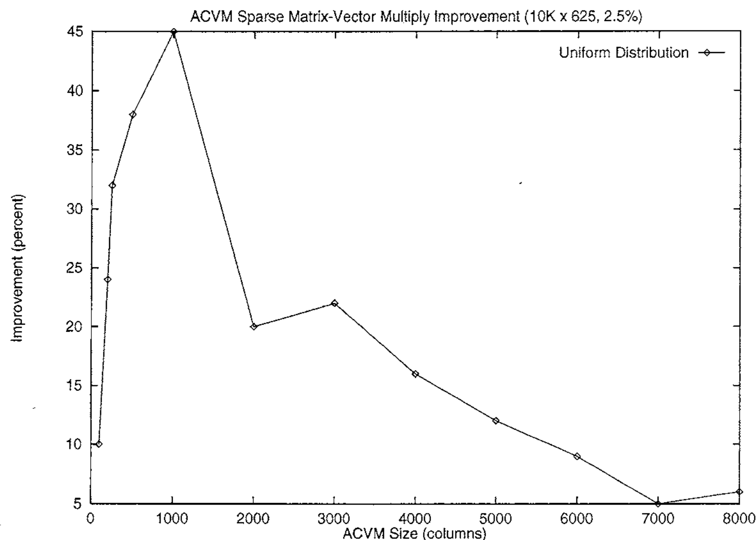

Figure 15: ACVM Performance Improvement vs. Distribution Variance on a SPARC-20 for Sparse Matrix-Vector Multiplication

In both cases the performance of the original code deteriorates as the simulation proceeds. The reason for this deterioration is that the `part` array is initialized with all the particles sorted by their positions, which means that consecutive reference to `grid` are made to adjacent values in the cache. Thus, for the first several simulation steps the references nearly represent an ACVM order. However, the particles move over time and change the referenced grid points; by the 175th iteration the particles have moved to such an extent that locality is lost, and performance has degraded by 13% and 20% respectively. These figures also show two different ACVM implementation methods. Both of these methods are able to maintain performance that is within a few percent of the optimum (which we assume to be the execution time of the first iteration of the original code), but differ in the overhead required to maintain strict ACVM order. In Figures 18 and 19 the time to perform the `EXSORT` method is approximately 7.5 and 5.35 seconds respectively. This compares to the time required by the `SEND` method which is 6.45 and 4.6 seconds respectively. Of course, the `EXSORT` method must move every reference in the particle array twice, and therefore we would expect that this would take longer than the `SEND` method. However, by allowing non-strict operation of the ACVM run-time we can amortize this cost over a number of iterations. In this case the run-time attempts to make a strict ordering once every 10 iterations. In this case the average execution time of these methods become 5.72 seconds for the `EXSORT` and 5.6 seconds for the `SEND` on the SPARC-10, and approximately 4.2 seconds for both methods on the SPARC-20.

Due to the rather large cache size on an IBM SP2 node, we would expect a different run-time behavior of the problem previously run on the SPARC processors. Figure 20 shows the results

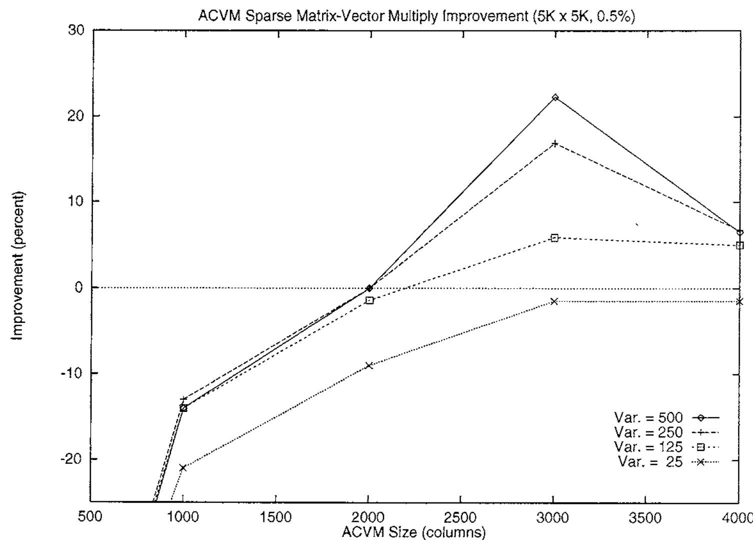

Figure 16: ACVM Performance Improvement vs. ACVM Size and Distribution on an SP2 Node for Sparse Matrix-Vector Multiplication

for the same problem size as before, but executed in a single node of an IBM SP2. There is neither appreciable performance degradation for the original code, nor a significant negative performance impact applying the ACVM **SEND** method. A larger problem size is required to see the effect of cache.

Figure 21 shows a PIC problem size where the `grid` array has been increased to be 256×512. The size of the ACVM is 32×32. In this case, the performance of the original code degrades from 1.3 to 2.2 seconds, a 41% performance drop. The average cost is 1.72 seconds for the **EXSORT** method, and 1.31 seconds for the **SEND** method.

The last architecture is the Cray T3D. This processor has a small direct mapped cache, and therefore we would expect that the ACVM size would have to be much smaller. In Figure 22 the performance of the original code decreases by 36%. Two sizes of ACVMs are also shown. The first size represents approximately 50% of the entire cache capacity, while the second size (16×16) is $\frac{1}{8}^{th}$ the capacity. The performance of the **SEND** code represents 11% and 34% improvement respectively.

Figure 23 shows the results of a comparison of different ACVM sizes. As expected, over some threshold size, the performance of too large an ACVM decreases as the simulation proceeds. This effect is caused by the deterioration of locality from the initial sorted order, and the self-interference misses caused by having too large a locality for the cache.

Another observation is that selecting an ACVM smaller than necessary does not have a large impact on performance. The cache locality provided by ACVMs is not influenced by a small

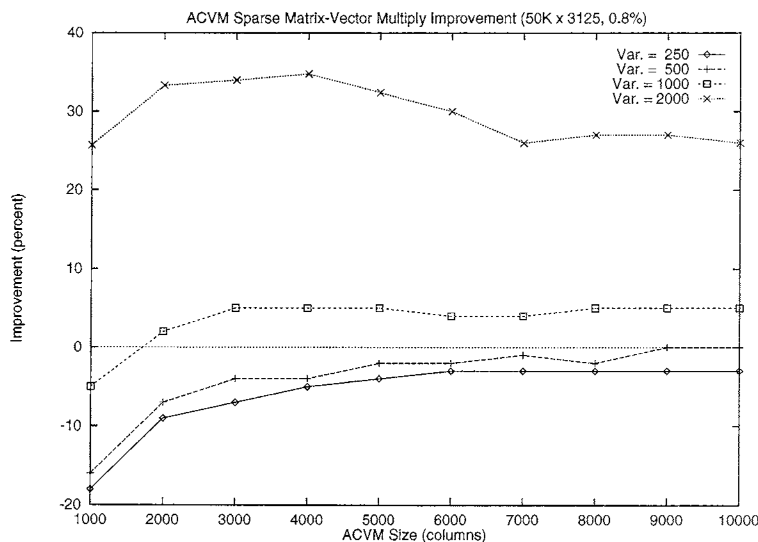

Figure 17: ACVM Performance Improvement vs. ACVM Size and Distribution on an SP2 Node for Sparse Matrix-Vector Multiplication

ACVM, but the overhead required for the run-time component of the ACVM model increases as the number of ACVMs increase.

Figure 24 shows the effect of reference density on performance. If the number of references per grid cell is too small then the there is not enough reuse of data in the cache to amortize the cost of bringing in the cache line from memory. The figure shows that for a reference density of less than one there is a degradation of approximately 18% from the optimal time. A reference density of 1.7 improves this to only a 5% degradation. A density ratio of nearly 7 provides the needed cache reuse to maintain optimum performance.

7 Conclusions

The ACVM model presented here provides a viable framework for optimizing the use of the cache and memory hierarchy for non-dense, dynamic codes. As described in this paper, there are two components that are integrated to form the model. The first is conventional static or dynamic cache analysis used for dense codes to determine the correct size for each virtual machine. The second is a method to keep the references local to each ACVM. It is this run-time component, along with data distribution among virtual processor, that suggests compiler directives that are similar to real processor data distribution techniques found in HPF. As described, the ACVM `SEND` method can be directly integrated into a parallel program by considering reference request moves between real processors as a special case of moving requests between ACVMs. Moreover,

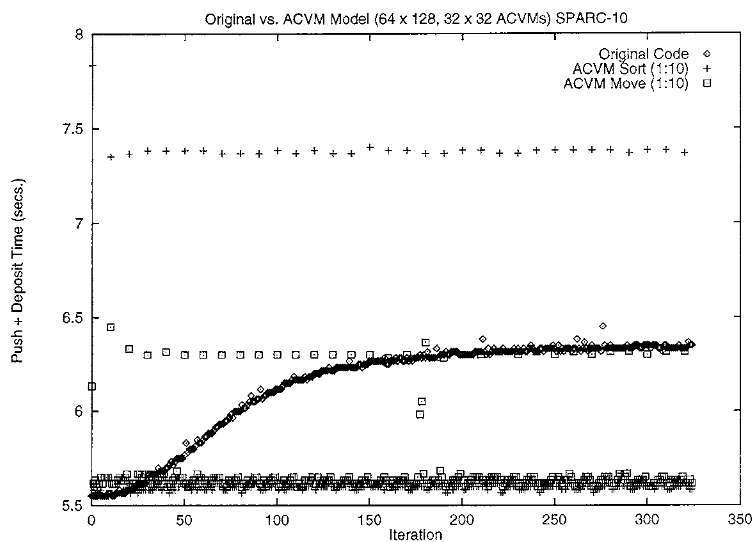

Figure 18: Performance Improvement for SunSPARC-10 Processor

the definitions of strict and non-strict ACVM run-times provide for amortizing the cost of the fine-grain operations required over some number of program iterations.

Acknowledgments The authors thank Steve Karmesin of the California Institute of Technology, Aeint de Boer of UCLA, and Paulette C. Liewer of JPL/CIT and Charles Norton of Rensselaer Polytechnic Institute for comments on the run-time performance of PIC codes and its relationship to cache size.

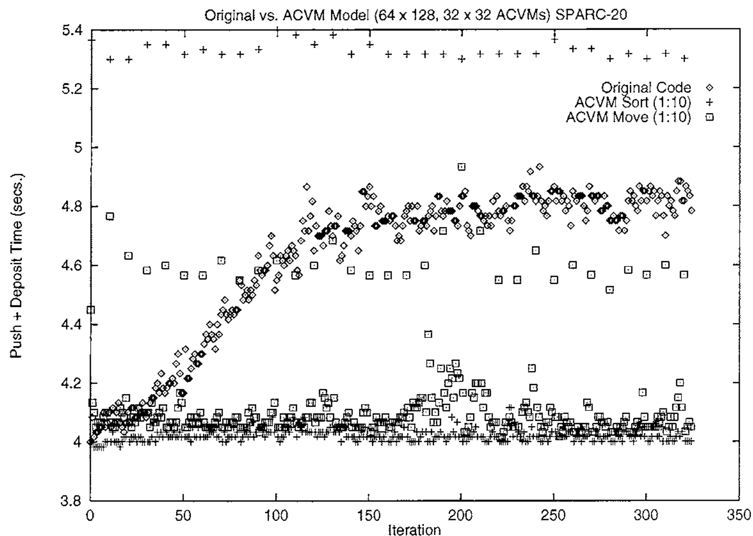

Figure 19: Performance Improvement for SunSPARC-20 Processor

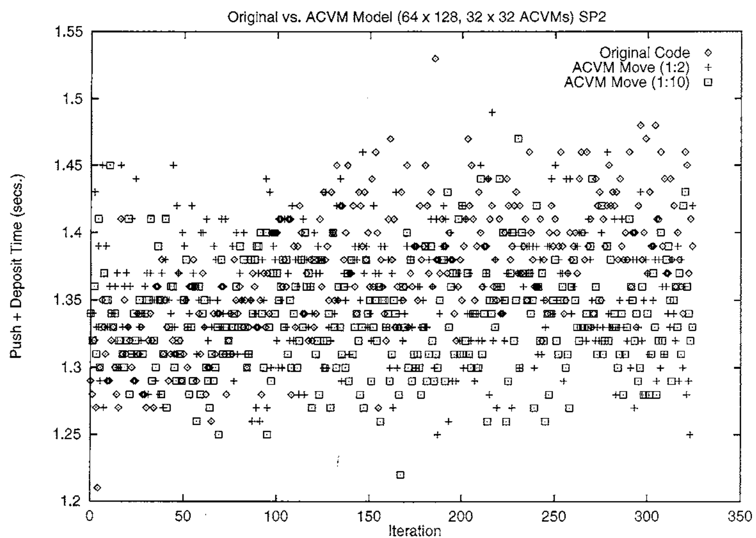

Figure 20: Performance Improvement for SP2 Node

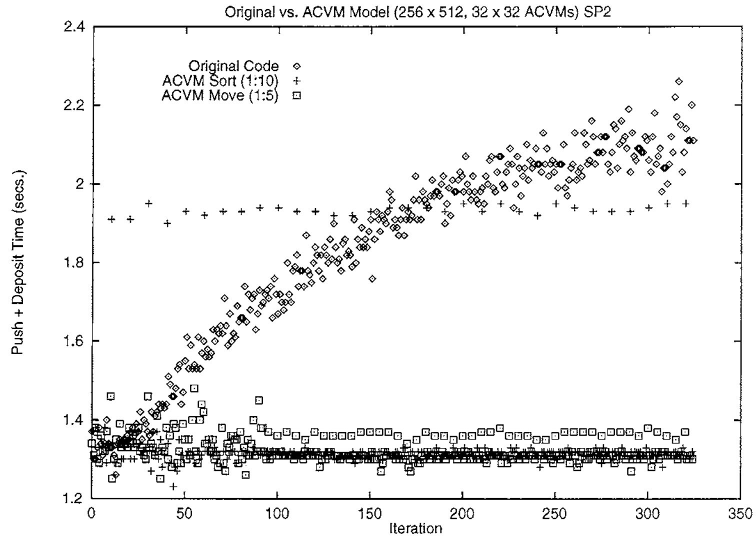

Figure 21: Performance Improvement for Larger Problem SP2 Node

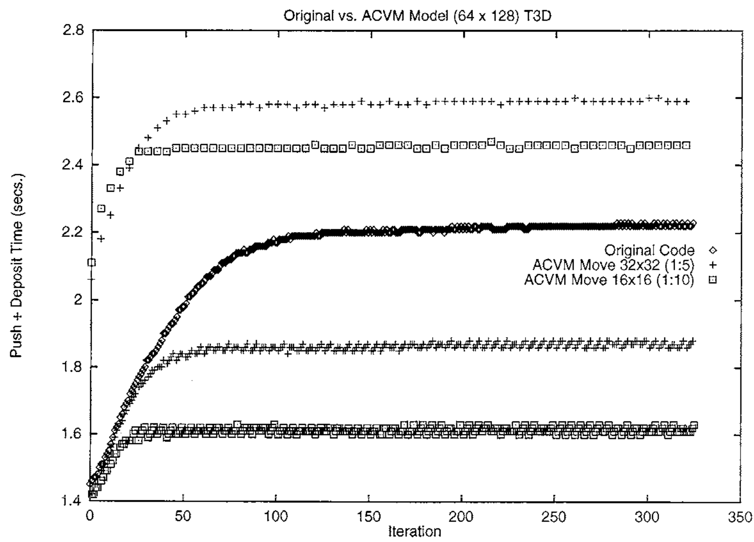

Figure 22: Performance Improvement for T3D Node

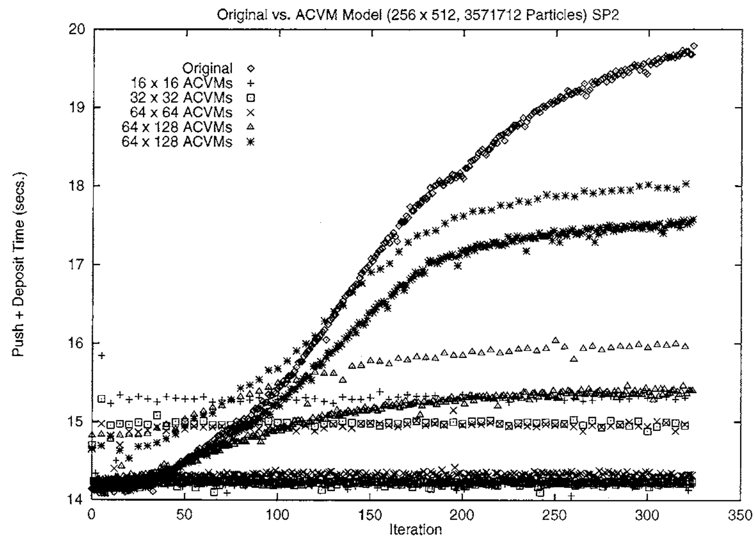

Figure 23: Impact of ACVM Size Selection on SP2 Performance

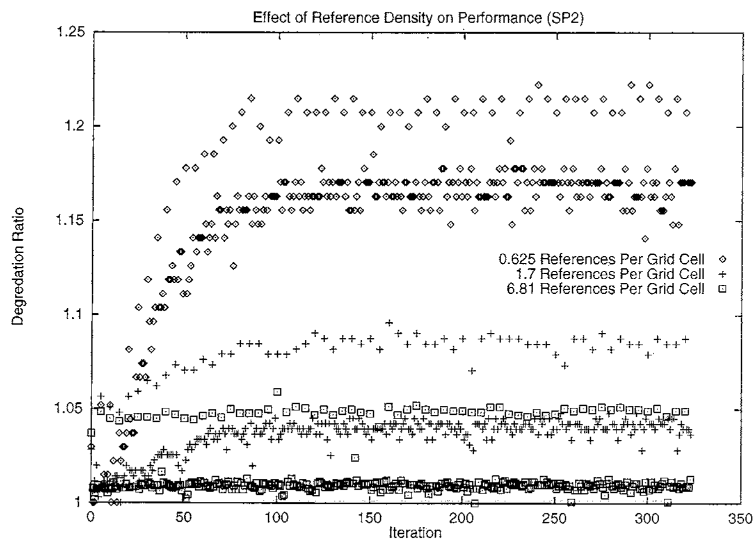

Figure 24: Impact of Reference Density on SP2 Performance

References

[1] Gagan Agrawal, Alan Sussman, and Joel Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applicatons. *IEEE Transactions on Parallel and Distributed Computing*, 1995.

[2] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.

[3] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. Technical Report 90-41, ICASE, May 1990.

[4] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation*. McGraw-Hill, New York, 1981.

[5] S. Carr, K. McKinley, and C W. Tseng. Compiler optimizations for improving data locality. In *ACM Architectural Support for Programming Languages and Operating Systems, San Jose, CA*, October 1994.

[6] Craig M. Chase and Anothony P. Reeves. Data remapping for distributed-memory multicomputers. In *Proceedings of the Scalable High Performance Computing Conference, SHPCC-92, Williamsburg, Virginia*, pages 137–144, April 26–29 1992.

[7] V. K. Decyk. Skeleton pic codes for parallel computers. *Computer Physics Communications*, 87(87), 1995.

[8] Viktor K. Decyk, Steven Roy Karmesin, Aeint de Boer, and Paulett C. Liewer. Optimization of particle-in-cell codes on risc processors. *Computers in Physics*, 1995. Submitted for Publication.

[9] T. Fahringer. Automatic Cache Performance Prediction in a Parallelizing Compiler. In *Proceeding of AICA 1993, Lecce/Italy*, September 1993.

[10] J. Ferrante, V. Sarkar, and W. Thrash. On Estimating and Enhancing Cache Effectiveness. In *Languages and Compilers for Parallel Computing, Fourth Internation Workshop, Santa Clara, CA*. Springer-Verlag, NY, August 1991.

[11] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1), June 1993.

[12] A. Goldberg and J. Hennessy. Performance debugging shared-memory multiprocessor programs with mtool. In *Processings of Supercomputing 91*, 1991.

[13] A. Gupta, M. Martonosi, and T. Anderson. Memspy: Analyzing memory system bottlencks in programs. *Performance Analysis Review*, 20(1), 1992.

[14] Seema Hiranandani, Joel Saltz, Piyush Mehrotra, and Harry Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12(4):415–422, August 1991.

[15] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993.

[16] W. K. Kaplow, B. Szymanski, and W. Maniatty. Impact of Memory Hierarchy on Program Partitioning and Scheduling. In *Proceeding of HICSS-28,Hawaii*, January 1995.

[17] W. K. Kaplow and B. K. Szymanski. Program optimization based on compile-time cache performance prediction. *Parallel Processing Letters*, 1995. To be published.

[18] D. R. Kincaid, J. R. Respess, D. M. Young, and R. G. Grimes. Itpack 2c: A fortran package for solving large sparse linear systems by adaptive accelerated iterative methods. Technical report, University of Texas at Austin, 1992.

[19] M. S. Lam, E. E. Rothberg, and M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. ACM ASPLOS, Santa Clara, CA*, pages 63–74. ACM, NY, April 1991.

[20] C. D. Norton, B. K. Szymanski, and V. K. Decyk. Object-oriented parallel computation for plasma simulation. *Communications of the ACM*, 38(10), October 1995.

[21] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, Houston, Texas, May 1989.

[22] Karen A. Tomko and Santosh G. Abraham. Data and program restructuring of irregular applications for cache-coherent multiprocessors. In *8th ACM International Conference on Supercomputing, Manchester, England.* ACM, July 1995.

[23] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proc. ACM SIGPLAN, Toronto, Canada.* ACM, NY, June 1991.

[24] Louis H. Ziantz, Can C. Ozturan, and Boleslaw K. Szymanski. Run-time optimization of sparse matrix-vector multiplication on simd machines. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE 94 Parallel Architectures and Languages Europe, Athens, Greece*, number 817 in Lecture Notes in Computer Science, pages 313–322. Springer-Verlag, July 1994.