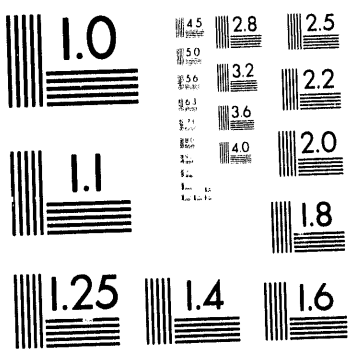
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

1 of 1

Proceedings of the VAPP-CONPAR 90 Joint Conference on Vector and Parallel Processing
Sep 90 Zurich, Switzerland

A Decoupled Data-Driven Architecture with Vectors and Macro Actors*

Paraskevas Evripidou and Jean-Luc Gaudiot†

USC/Information Sciences Institute, 4676 Admiralty Way
Marina del Rey, California 90292-6695

†Department of Electrical Engineering, University of Southern California
Los Angeles, California 90089-0781

Abstract

This paper presents the implementation of scientific programs on a decoupled data-driven architecture with vectors and macro actors. This hybrid multiprocessor combines the dynamic data-flow principles of execution with the control-flow of the von Neumann model of execution. The two major ideas utilized by the decoupled model are: vector and macro actors with variable resolution, and asynchronous execution of graph and computation operations. The compiler generates graphs with various-sized actors in order to match the characteristics of the computation. For instance, vector actors are proposed for many aspects of scientific computing while lower resolution (complier-generated collection of scalar actors) or higher resolution (scalar actors) is used for unvectorizable programs. A block-scheduling technique for extracting more parallelism from sequential constructs is incorporated in the decoupled architecture. In addition a graph-level priority-scheduling mechanism is implemented that improves resource utilization and yields higher performance. A graph unit executes all graph operations and a computation unit executes all computation operations. The independence of the two main units of the machine allows the efficient pipelined execution of macro actors with diverse granularity characteristics.

1 Introduction

The *Decoupled Data-Driven Architecture with Vectors and Macro Actors* is a hybrid data-driven/control-driven architecture with decoupled graph and computation units [1]. It provides an efficient way, both in cost and performance, to implement the data-flow principles of execution at the macro level. Two major ideas are utilized by this architecture: vector and macro actors with variable resolution, and decoupling of graph and computation operations. The variable-resolution model retains the dynamic data-flow model at the coarse-grain level and utilizes the vectorized control-flow model at the fine-grain level. Vectorization and vector processing in general provides a very efficient

*This material is based upon work supported in part by the U.S. Department of Energy, Office of Energy Research, under Grant No. DE-FG03-87ER25043 and in part by the Defense Advanced Research Projects Agency under NASA Cooperative Agreement NCC-2-539 and RADC Contract F30602-88-C-0135.

platform for scientific computing. The variable-resolution graphs expand on this principle by having actors of varying degrees of granularity. At the fine level, scalar actors are supported, while vectors of variable length make up the intermediate level of granularity. The vector size can be determined at compile time or at execution time with the use of special-purpose actors. *Compound Macro Actors* (CMA) are supported at the coarser level of granularity. These are compiler created collections of scalar and/or vector actors. However, increasing the level of granularity decreases the amount of parallelism. Thus there is a tradeoff involved in determining the level of resolution. The variable-resolution model has the ability to adjust the granularity to match program characteristics for best performance.

Iterative methods for solving linear systems, an integral part of scientific computing, are inherently highly sequential. However, in data-driven machines, scheduling the iterative part of the algorithms in blocks and looking ahead across several iterations can exploit a lot of parallelism. Block-scheduling techniques are incorporated into the variable-resolution graphs.

The decoupled model of execution separates (decouples) each actor into two parts: the graph portion and the computation portion. The computation portion of each actor is a collection of conventional instructions (load/store, add, etc). The graph portion contains information about the executability of the actor and its consumers. Thus, a decoupled data-driven graph can be viewed as a conventional program with a data-dependency graph superimposed on it. In the decoupled architecture, the graph unit executes all graph operations (determination of executability) and the computation unit executes all computation operations (code fetching and execution). The two units execute in an asynchronous manner, i.e., the computation unit does not have to execute the computation portions of actors in the same order as the graph unit executes the graph portions. Communication between the two units is provided via queues. Under steady-state conditions, provided the queues are not empty, the effective pipeline cycle of the processor is reduced to that of the computation unit. This can yield a considerable boost in performance.

The fundamental principle of data-driven machines, *execution upon data availability*, gives no special treatment to the critical path. This results in inefficient resource utilization and loss of performance in loop-based algorithms with dependencies across iterations. A graph-level priority-scheduling mechanism has been incorporated in the decoupled architecture that improves resource utilization and yields higher performance.

In Section 2, the construction of variable-resolution graphs for scientific applications is introduced. The basic framework of the decoupled architecture and the graph-level priority scheduling is presented in Section 3. Concluding remarks and future research issues are presented in Section 4.

2 Variable-Resolution Graphs for Scientific Applications

Variable-resolution graphs (VR-graphs) are composed of actors of varying resolution (amount of computation per actor). They retain the dynamic data-flow principles of execution at the coarser level (macro actor). Vectorized control-flow is employed at the

```
function Jac(a: array; x ,b  vectors
                          returns vector )

  for i in 1,n
   returns array of
   ( b[i] -  for j in 1,n
          returns value of sum a[i,j] * x[j]when j=~i
         end for ) / a[i,i]
  end for
end function
```

(a)

```
function  V-Jac(a: array; x ,b  vectors
                          returns vector )

 for i in 1,n
   ip := Vmask(i, Vmul( a[i],x));
   sm:= Vsum(ip)
 returns array of Vsub( b[i] - sm) / a[i,i]
 end for
end function
```

(b)

Figure 1: Jacobi algorithm expressed in (a) SISAL and (b) V-SISAL

fine level (within a macro actor). The actors supported by the VR-graphs can be grouped into three major classes:

- **Scalar actors.**
- **Vector and Macro actors.**
- **Compound Macro Actors (CMAs):** Compiler-generated collections of scalar and/or vector instructions.

VR-graphs exploit the locality present in vectorizable applications with the use of vector macro actors. Also, sequential constructs like index generation for loops are implemented by CMAs. Thus, the dynamic data-flow overhead (tag creation and processing) is not applied on a per-instruction basis but is instead applied on a macro-actor basis. Let the average processing requirements of an instruction be t_p, and let t_o denote the average overhead per instruction in machine cycles. Processing a vector of size n with fine-grain resolution requires $n \times (t_c + t_o)$ cycles, where the coarser grain (vector macro actors) requires $nt_c + t_o$. Thus, the per-instruction overhead is reduced by a factor of n if $t_o >> t_p$.

As in the original U-interpreter [5], the token format for the VR-graphs is $V_{[c.s.i]}$ where V is the data value, which can be scalar or a vector, and $_{[c.s.i]}$ is the tag. The first part of the tag "c" is the context identifier, "s" is the destination address, and "i" is the iteration identifier (used for loops). Portions or the whole of the tag might be omitted in the examples presented in this paper if no ambiguity arises.

2.1 An Example Vectorized Jacobi Algorithm

Iterative algorithms are very powerful tools for solving linear systems and are particularly efficient in solving large sparse systems, frequently encountered in the solution of Partial Differential Equations. The Jacobi algorithm for solving linear systems expressed in both SISAL [6] and V-SISAL [7] (SISAL 1.2 with our vector extensions) is depicted in Figure 1. The VR-graph of the vector Jacobi algorithm is depicted in Figure 2.a.

The general form of a vector instruction is *Vadd(N,***a**,**b***)*, where **a** and **b** are vectors of length N. The vector load instruction, *Vgather(a,b,c,* **v***)*, provides the flexibility of

operating on selected elements of a vector, where a is the first value to be processed, b is the maximum range, and c is the stride or offset. The *indices* actor also receives as input a 3-tuple $< a, b, c >$ and creates the indices of the loop (Do i=a,b,c). The actor *MVgather(i,a,b,c,* **a***)* selects a vector from matrix **a**. The actor *Vmask(i,***a***)* masks out the i^{th} value from the vector **a**. In Figure 2.a the actor *Vmul* performs the multiplications of the i^{th} row of matrix **A** with vector **X**. The *Vmask* actor masks out the i^{th} value of the resulting product in compliance with the algorithm. The summation part of the algorithm is performed by the the *Vsum* actor; the rest of the computation is then performed in a fine-grain fashion. The resolution of the vectorized Jacobi can be either decreased or increased to better match the target machine's characteristics. Partitioning Figure 2 into CMAs decreases the resolution, which in turn reduces the amount of overhead. Chaining can also be achieved by grouping vector actors into CMAs. An increase in granularity increases the amount of parallelism.

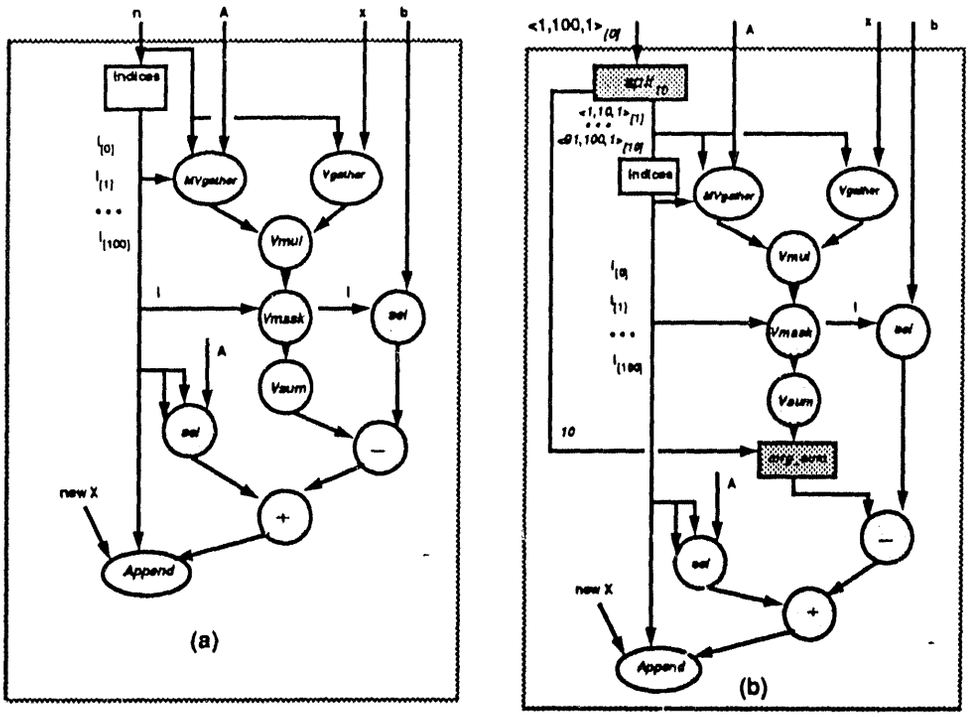

Figure 2: Jacobi algorithm with (a) vector macro actors and (b) variable-resolution actors

Conventional vector supercomputers are generally very efficient when operating on large vectors. However, the proposed architecture is targeting a large number of processors with an emphasis on parallelism. Therefore, it is desirable that large vectors be distributed throughout the machine. The VR-graphs support the runtime adjustment of granularity with the use of the *split* and *merge* actors. The *split* actor $splt_k$ takes as input the total range $< a.b.c >$ and splits it into n parts of size k, each with a unique tag. Thus n partial threads of computation are created, executed independently, and merged at the end. Merging of the partial threads is performed by the *merge* actors. These are divided into two subclasses: reduction and non-reduction. The reduction *merge* actors are used when

the merge operation occurs after a reduction operation (vector-to-scalar), for example, *Vsum* is coupled with *mrg-sum*. A non-reduction *merge* actor is used when the overall thread of computation is non-reduction (vector-to-scalar-producer), for example, *Vstore* is coupled with *mrg-store*. In non-reduction mode, the *merge* actors process subvectors as they arrive. For example, the *mrg-store* actor will receive n subvectors, which it must store in the structure store. The tags of the subvectors allow the *mrg-sum* actor to determine their positions in the vector. The value k can either be determined at compile time or be evaluated at execution time as a function of the data-set size and machine load.

The use of *split* and *merge* actors is demonstrated with the use of the vectorized Jacobi example, which uses a reduction merge actor *mrg-sum* as depicted in Figure 2.b. The $splt_{10}$ actor gets as input the range $< 1, 100, 1 >$ and splits it into $n = 10$ subranges ($< 1, 10, 1 >_{[1]} \ldots < 91, 100, 1 >_{[10]}$) of 10 elements each. These subranges are tagged uniquely. Thus during execution, $n = 10$ instances of the *Vgather*, *MVgather*, *Vmul*, *Vmask*, *Vsum*, and *mrg-sum* actors will be activated. The last actor in this thread of computation is the *Vsum* actor, which generates 10 partial results. The *mrg-sum* actor performs the summation of the partial results and sends a scalar value to the "-" actor. The remainder of the graph is composed of scalar actors ("-", "+", *sel*, and *append*).

2.2 Block Scheduling

The vectorized Jacobi with the variable-resolution actors exploits locality through vectorization and thus reduces the overhead per operation, and the variable resolution exploits parallelism. However, the sequential nature of the algorithm (perform one iteration, check stopping criterion, and activate the next iteration until required accuracy is achieved), does place a limit on the amount of parallelism exploited. However, iterative algorithms such as the Jacobi can become more efficient for parallel execution if some amount of look-ahead is employed: a parallel loop (Forall $i = 1, n$) is inserted inside the sequential loop (repeat-until) [8]. This allows the dynamic data-flow graph to simultaneously unravel a block of n iterations (block scheduling) instead of merely one. Figure 3 provides a qualitative picture of the basic principles of sequential and parallel scheduling of iterative algorithms. The basic form of the modified vector Jacobi implementation is shown in Figure 3.c.i. Figure 3.c.ii shows the traditional implementation of the Jacobi algorithm. The function *evaluate_block_size* is used to give the initial block size. The decision will be based on the nature of the problem and the convergence rate of the algorithm. The function *evaluate_new_block_size* produces a new block size by estimating the number of iterations needed to achieve the required accuracy.

Block scheduling with look-ahead of the vectorized Jacobi implementation has the following advantages:

- It exploits pipelining across various iterations: as soon as the subvector $x_1^{(k)}$ has been calculated, the next iteration $k + 1$ can be initiated without waiting for the entire production of the vector $x^{(k)}$.
- It reduces the algorithm overhead by drastically reducing the number of evaluations of the convergence criterion.
- It neutralizes the effect of "overhead/synchronization" actors by making them available early in the computation so they can be executed by otherwise idle processors.

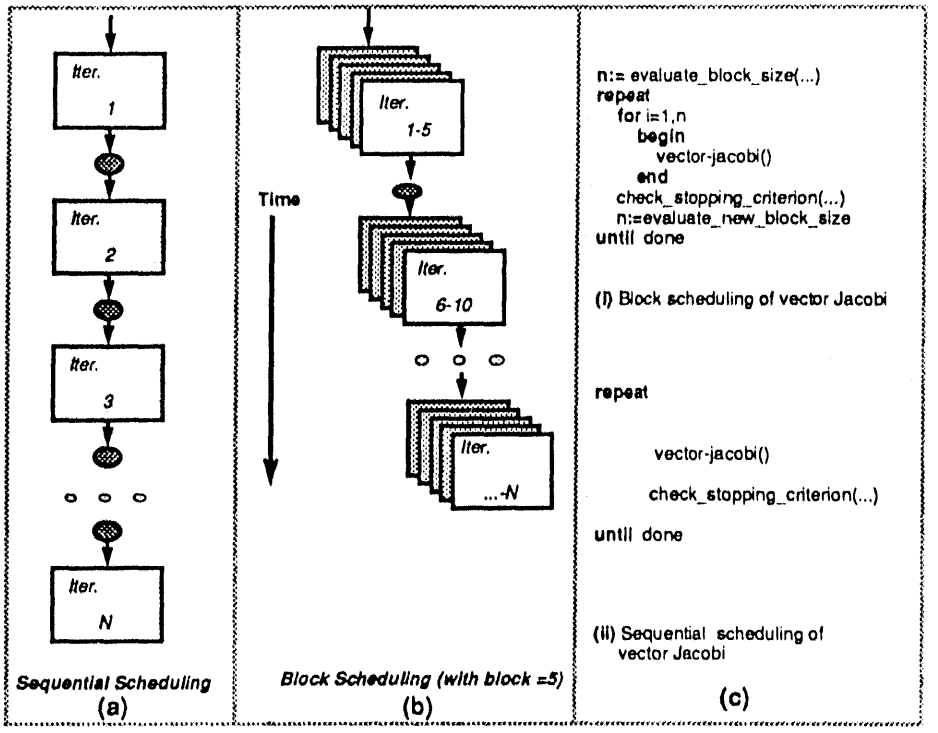

Figure 3: Sequential scheduling vs. Block scheduling

3 Decoupled Architecture

The actors of the VR-graphs have very diverse computational needs. For instance, a scalar add requires only a few cycles, while a vector add needs time proportional to the number of elements in the vectors. Furthermore, the size of the vector is not fixed and can vary widely even within the same graph. On the other hand, the processing time requirements of the graph portion of the operations (token matching and formatting) are about the same for the various levels of resolution. This could create an imbalance in the processor pipeline. Therefore, the synchronous processor pipeline is not an efficient approach for executing VR-graphs. The graph pipeline stages (match and token formatting/routing) and computational stages (fetch and execute) should be decoupled to accommodate imbalance among the execution times of the pipeline stages. The Decoupled Graph/Computation (DGC) model [7, 1] decouples the processor pipeline into two pipelines that communicate via queues. The operations performed by a decoupled data-flow computer are classified into:

Graph operations: All operations involved in tag creation and processing are classified as graph operations (sometimes referred to as graph overhead). These include tagged-token matching, token formatting, and routing. Graph operations can be summarized as the operations involved in *determining executability by data availability.*

Computational operations: These include conventional instructions such as load, store, and ALU operations.

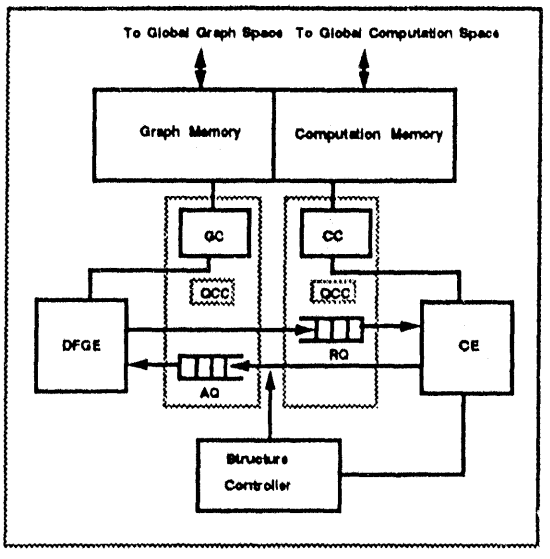

Figure 4: A processing element with decoupled graph computation units

Graph operations are performed by the Data-Flow Graph Engine (**DFGE**) and computational operations are performed by the Computation Engine (**CE**). The two engines communicate via two queues. The Ready Queue (RQ) holds the actors deemed executable by the DFGE. The Acknowledge Queue (AQ) holds the actors executed by the CE.

The basic structure of a decoupled processing element is depicted in Figure 4. The CE is a RISC-like processor. It executes the computation operations of each actor in a control-flow mode of execution (program counter). The interface to the decoupled architecture is provided by the addition of a new instruction (Ret) that acknowledges the execution of the current actor by placing its address in the AQ and then uses the top of the RQ to fetch the first instruction of the next actor to be executed. The DFGE determines which actors are ready to be executed by updating the dynamic data-flow graph. It places the dynamic identification (actor's name and specific context $< ID, context >$) of ready actors in the RQ. The DFGE receives the Ack signals through the AQ and updates the data-flow graph. The Ack signals are used to notify actors that their operand(s) is (are) ready. Thus, the operand matching and the token formatting and routing stages of the data-flow model are reduced to one stage. However, the Ack signals are not pointers to the data produced; this information is embedded into the internal graph of the actor templates.

3.1 Mode of operation

The DGC architecture uses dynamic data-frames for implementing the data-driven principles of execution. Each dynamic instance of an actor reads its inputs form and writes its output into unique memory locations. The overall mode of operation of the decoupled architecture is described with the aid of the the inner product example depicted in Figure 5.

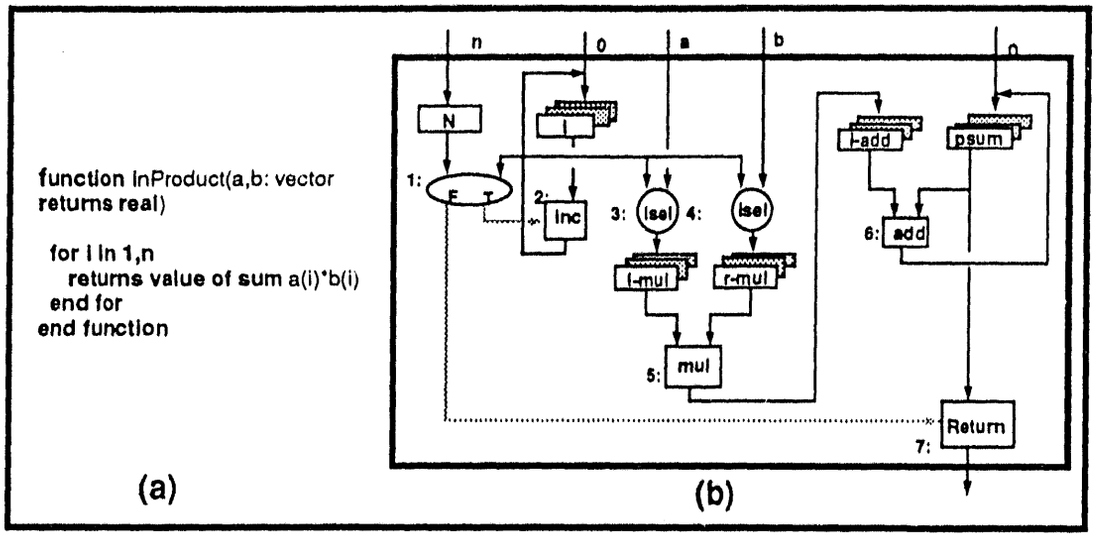

Figure 5: The vector inner product example (a) SISAL code and (b) D^3-graph

Applications programs written in SISAL are first compiled into VR-graphs and then into decoupled data-driven graphs (D^3-graph). The decoupled data-driven graph of the inner product example is shown in Figure 5. The shaded lines represent data dependencies with no actual movement of data; the solid lines represent actual data movement. This graph has one static value (n) and five dynamic values (i, *l-mul*, *r-mul*, *l-add*, *psum*). The static values remain constant throughout the execution of the block, as opposed to the dynamic values, for which a new value is created at each iteration. Figure 6 depicts the contents of graph and computation memories for the inner product example. Each actor is represented by an actor template, which is made up of a graph and a computation sub-template (stored in the graph and computation memories, respectively). The graph sub-template contains the following: (1) the address of the corresponding computation code; (2) the actor number within the block; and (3) the consumer list. The computation sub-template is a thread of computation. The code shown in Figure 6.a is in a generic 3-address RISC assembly form: *Label : opcode source1[, source2], destination*

The basic component of the D^3-graphs is the *context-block*. The *context-blocks* mark the boundaries of a loop that changes the iteration context part of the tag. The inner product example of Figure 5 and 6) represents one *context-block*. For each invocation of the *context-block* a new graph and computation base address must be assigned. Thus the context in a *context-block* is synonymous with the iteration number. The computation data-frame is composed of all the dynamic values of the block, and the graph data-frame is composed of the status-words (number of tokens required for the actor to fire) of the actors in the block.

The contents of the Ready Queue is the 2-tuple actor < *actor address, context* >. The CE uses the actor address to fetch the first instruction of the actor and also loads the new context in the context register. The dynamic values belonging to certain context are accessed by using the context register (Rc) as index register. The instruction:

load i,Rc,R1

causes register R1 to load the dynamic instantiation of the value i that belongs to the context pointed to by the Rc.

The first actor of Figure 6 is the implementation of the *switch* actor. In the decoupled data-driven graph, the *switch* actor does not transfer any values but instead activates the "true" or the "false" block according to the evaluation of its predicate. The "Ret" instruction is not sufficient to implement the *switch* actor because it gives no information about the evaluation of the predicate. Two more instructions are necessary for that: *RetT*, which causes the DFGE to update the status of the consumers of the "true" block, and *RetF*, which activates the consumers of the "false" block.

Split-phase transactions are used for structure synchronization. Actors 3 & 4 of Figures 5 and 6 represent the I-structure [9] select operation. The CE executes thses actors by placing a request to the structure controller (Figure 4). When the requested value is ready the structure controller stores it in the location specified by the CE and notifies the DFGE which in turn updates the status of the consumer actor(s) (actor 5 of Figure 5).

The last actor of each context-block is the *Return* actor, which stores the result of the context-block in the parent's data-frame. It also triggers garbage collection for the data frames used by the context-block. In Figure 6, the RESULT label points to the location where the result of the context-block must be stored.

When a block is activated, the addresses of all ready actors are placed in the Ready Queue. The CE removes one actor address at a time, executes all instructions of that actor, and places the actor address into the AQ. The DFGE removes the address of an "executed" actor from the AQ and reads its consumer list from its graph subtemplate. It then decrements the status word of each consumer. The triplet <*base address, actor number, context*> makes up the address of the status word of each actor. When the status word reaches zero, the actor is deemed executable, and the DFGE places the address of its computation code in the RQ.

Our multiprocessor architecture is cluster based with global virtual address space. A schematic diagram of the architecture for a 256 processor machine is depicted in Figure 7. Two disjoint virtual addressing spaces are used for the graph and computation memories. Each cluster has a shared graph memory and a shared computational memory. All cluster memories are connected via a communication medium that allows inter-cluster reads and writes.

3.2 Priority Scheduling

The basic principle of the data-flow model of execution is *execution upon data availability*. If this, however, is the sole scheduling criterion, the critical path of the graph receives no special treatment. This becomes essential in applications with a high degree of data-dependencies like iterative or direct methods for solving linear systems. It has been shown in [8] that graph-level priority scheduling using the outermost level of the tag as the priority field results in higher performance and more efficient use of machine resources.

We have adopted a relative-priority mechanism for the decoupled architecture: priority is assigned according to the iteration identifier at the outermost level of the tag. Thus no absolute priority is enforced but rather priority is enforced among actors belonging to the same context-block. Consider, for example, the block-scheduling implementation of

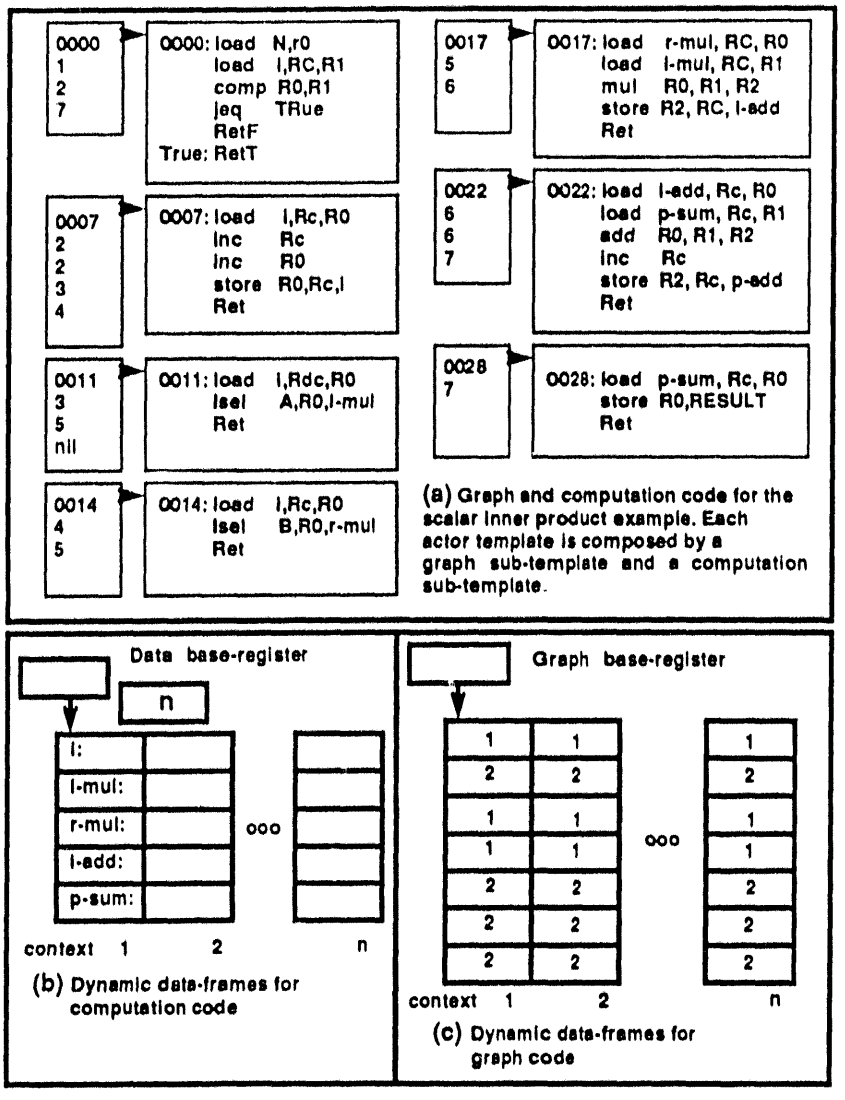

Figure 6: Contents of the graph and computation memories for the inner product exmple

the Jacobi alogorithm that consists of three nested loops (scalar implementation). Figure 8 depicts the evolution of the tag of a token while it traverses the graph, and its final structure at the outermost level of the three nested loops The outermost iteration identifier i (shaded in Figure 8) is used as the priority key (highest priority is given to the lowest iteration number). In the DGC architecture the graph-level priority is implemented by hardware in the Ready Queue. As mentioned earlier, the contents of the Ready Queue are 2-tuples <*actors address, context*> (context is the iteration number). The priority hardware keeps the ready queue sorted according to the context of each entry.

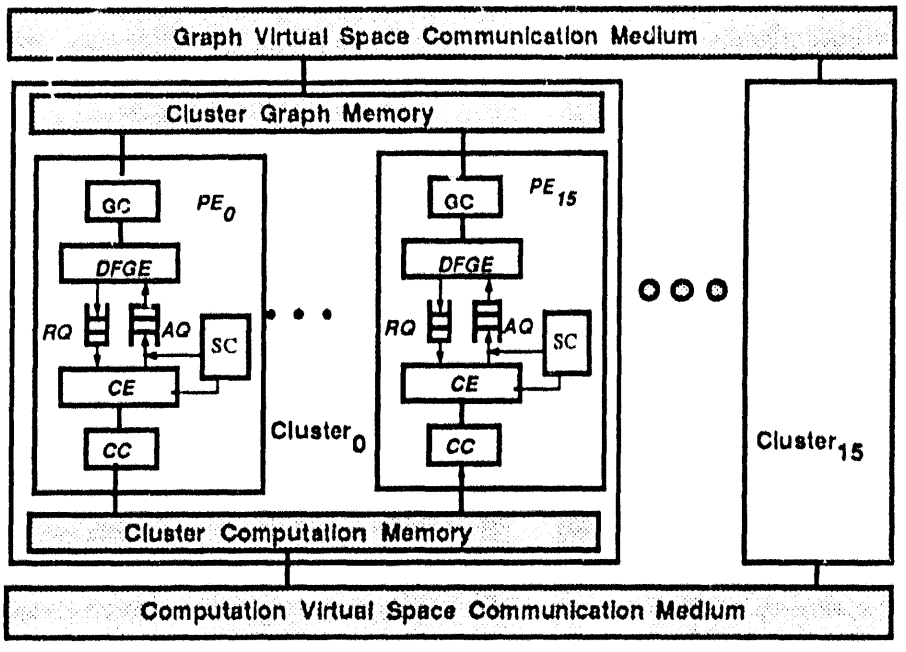

Figure 7: Decoupled cluster based multiprocessor architecture

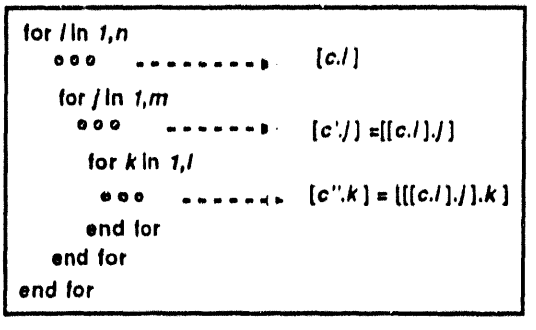

Figure 8: Tag structure for nested loops

4 Concluding Remarks

In this paper, we have presented some implemetation issues of scientific appliactions on a hybrid data-driven/control-driven architecture where the atomicity of computation is variable (micro, vector, and compound actors). The variable-resolution graphs facilitate high performance by exploiting locality through efficient vector operations. For non-vectorizable code, the proposed architecture can benefit from the parallelism provided by the dynamic data-flow principles of execution applied to fine-grain actors and CMAs, while retaining the benefits of vector operations. Furthermore, the performance of iterative algorithms can be enhanced by the block-scheduling techniques, which can be applied regardless the level of granularity. Block scheduling allows maximum pipelining among successive iterations. Furthermore, it allows saving on the amount of computation performed for checking termination criterion by employing look-ahead. The variable-

resolution model allows many degrees of freedom in adjusting computation granularity and thus can match program and machine characteristics for best performance. Graph-level priority scheduling has been incorporated in the decoupled architecture. This yields considerably better resource utilization and faster execution.

In the proposed decoupled model both the graph unit and the computation unit operate asynchronously. Thus under steady-state conditions, if both queues are not empty, the effective pipeline cycles is reduced, which can yield a considerable boost in performance. The hybrid nature of the architecture allows it to adapt and readily uses technology and building blocks of the von Neumann model of execution.

References

[1] P. Evripidou and J.-L. Gaudiot, "A decoupled graph/computation data-driven architecture with variable resolution actors," in *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

[2] P. Treleaven, R. Hopkins, and P. Rautenbach, "Combining data flow and control flow computing," *Computer Journal*, vol. 25, no. 2, 1982.

[3] W. Najjar and J.-L. Gaudiot, "Multi-level execution in data-flow architectures," in *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, IL, pp. 32–39, August 1987.

[4] K. Hiraki, S. Sekiguchi, and T. Shimada, "Efficient vector processing on a Dataflow Supercomputer SIGMA-1," in *Proceedings of Supercomputing '88*, 1988.

[5] Arvind and K. Gostelow, "The U-Interpreter," *IEEE Computer*, pp. 42–49, February 1982.

[6] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee, "SISAL-Streams and Iterations in a Single Assignment Language, Language Reference Manual, Version 1.2," Tech. Rep. TR M-146, University of California - Lawrence Livermore Laboratory, March 1985.

[7] P. Evripidou and J.-L. Gaudiot, "The USC decoupled multilevel data-flow execution model," in *Advanced Topics in Data-Flow Computing*, Prentice Hall, 1990. In press.

[8] P. Evripidou and J.-L. Gaudiot, "Some scheduling techniques for numerical algorithms in a simulated Data-Flow multiprocessor," in *Proceedings of the Parallel Computing 89*, Elsevier Science Publishers B.V., August 1989.

[9] Arvind and R. Thomas, "I-structures: An Efficient Data Type for Functional languages," Tech. Rep. LCS/TM-178, Massachusetts Institute of Technology, Laboratory for Computer Science, June 1980.

DISCLAIMER

DATE
FILMED
12/01/93
END