International Journal of Parallel Programming, Vol. 31, No. 2, April 2003 (© 2003)

Non-Strict Execution in Parallel and Distributed Computing

Alfredo Cristobal-Salas,[1] Andrei Tchernykh,[1]
Jean-Luc Gaudiot,[2] and Wen-Yen Lin[3]

Received February 22, 2002; revised August 14, 2002; accepted October 25, 2002

This paper surveys and demonstrates the power of non-strict evaluation in applications executed on distributed architectures. We present the design, implementation, and experimental evaluation of single assignment, incomplete data structures in a distributed memory architecture and Abstract Network Machine (ANM). Incremental Structures (IS), Incremental Structure Software Cache (ISSC), and Dynamic Incremental Structures (DIS) provide non-strict data access and fully asynchronous operations that make them highly suited for the exploitation of fine-grain parallelism in distributed memory systems. We focus on split-phase memory operations and non-strict information processing under a distributed address space to improve the overall system performance. A novel technique of optimization at the communication level is proposed and described. We use partial evaluation of local and remote memory accesses not only to remove much of the excess overhead of message passing, but also to reduce the number of messages when some information about the input or part of the input is known. We show that split-phase transactions of IS, together with the ability of deferring reads, allow partial evaluation of distributed programs without losing determinacy. Our experimental evaluation indicates that commodity PC clusters with both IS and a caching mechanism, ISSC, are more robust. The system can deliver speedup for both regular and irregular applications. We also show that partial evaluation of memory accesses decreases

[1] CICESE Research Center, Ensenada, BC, Mexico. E-mail: {asalas,chernykh}@cicese.mx
[2] UCI Parallel Systems & Computer Architectures Lab, Department of Electrical and Computer Engineering, University of California, Irvine, California 92697. E-mail: gaudiot@uci.edu
[3] TIA Mobile, Inc., Los Angeles, California. E-mail: wenyenl@tiamobile.com

0885-7458/03/0400-0077/0 © 2003 Plenum Publishing Corporation

the traffic in the interconnection network and improves the performance of MPI IS and MPI ISSC applications.

KEY WORDS: Incremental structures; software cache; message passing; partial evaluation; non-strict information processing.

1. INTRODUCTION

While conventional (and sequential) information processing usually implies complete production of data before its consumption, the asynchronous execution present in most concurrent systems often requires that this restriction be lifted. This means that data structures as well as the execution model must be completely re-thought in the context of distributed control and storage. Indeed, asynchronous evaluation is facilitated when structure production and consumption can be allowed to proceed with a looser synchronization than conventionally understood. This can take place at the highest level (language), at the intermediate execution level (evaluation model), or at the lowest level in the representation of the communicating data structures themselves.

Indeed, one source of inefficiency in current memory sharing mechanisms is in the fact that the various primitives impose too rigid a control structure on the computations responsible for filling in the components of the data structure. Imperative languages do not suffer from this drawback because the allocation of a data structure (variable declaration) is decoupled from the filling in of that data structure (assignment). However, imperative languages, with their unrestricted assignments, complicate parallelism detection because of timing and determinacy issues. Non-strict structures, for example Incremental Structures (ISs), are an attempt at regaining this flexibility without losing determinacy. Some of such structures are well known and have been experimentally evaluated for a variety of multithreaded shared memory systems. The problem is to assess how suited they are for the exploitation of parallelism in distributed memory systems which use the latency tolerance properties of MPI. We focus here on split-phase memory operations and non-strict information processing under a distributed address space to demonstrate their impact on the overall distributed system performance.

Partial evaluation is another example of incomplete information processing. It allows the partial execution of a program when only some of its input data are available, and generation an often faster residual program that contains the operations which are dependent on the unknown parameters. Such a reduction of operations is a source of program optimization. The next question then becomes, would it be possible to use partial evaluation,

not only to remove much of the excess overhead of program and message passing, but also to reduce the number of messages? The answer is effectively *yes*. In this paper, a novel technique of optimization at the communication level is proposed and described. It is based on the partial evaluation of local and remote memory accesses when some information about the input or part of the input is known. To our knowledge no similar approaches have been reported in the literature.

The incompleteness of IS is considered at the level of elements. Dynamic I-Structures (DISs)[1] differ from I-Structures in that they have an unrestricted number of elements. This may increase the degree of incompleteness. In this paper, we show that DISs lead to fine-grain parallel computation where more optimization can be done by partial evaluation. They are also useful when solving problems where complicated dynamic data structures (such as lists, trees, etc.) are involved and hand or compiler detection of parallelism is difficult.

The paper is organized as follows: In Section 2, we concentrate on giving the essential ideas relevant to the concept of incompleteness on three levels (language, evaluation model, and data structures). Section 3 introduces distributed I-Structure and I-Structure Software Caches (ISSC) implementation. The concept of their partial evaluation is described in Section 3.1, and experimental evaluation is presented in Section 3.2. In Section 4, we present some experimental results related with the DISs. Finally, some related work and conclusion are discussed.

2. NON-STRICT INFORMATION PROCESSING

2.1. Strict and Non-Strict Languages

There is much debate regarding the relative merits of strict and non-strict languages:[2] Incomplete information processing can be a useful characteristic of languages since they can be so easily targeted to parallel execution. In strict languages, the arguments of a function are always evaluated before it can be invoked. As a result, if any expression in the body of a function does not need the value of the arguments, or depends on any part of the arguments, the whole function cannot be evaluated until all argument values are actually evaluated. Strictness imposes also the following requirement: complete evaluation of all components before a structure is build. This has the inherent side-effect of severely restricting coarse and fine-grain producer-consumer pipelining. Conversely, in a non-strict language, the arguments to a function are not necessarily evaluated before their values are actually required. Hence, evaluating the expression may still terminate properly, even if the evaluation of arguments does not

(when the values of the incomplete parameters are not used in the expression). For this reason, non-strict functions are also called *lazy functions*, and are said to evaluate their arguments *lazily*, or *by need*.[3–6] *Lenient* semantics are also non-strict:[7, 8] functions with lenient semantics may be invoked before their arguments are evaluated and structures may be built before their individual components are evaluated, or while their evaluation is undertaken. Lenient semantics offer the significant advantage of eliminating the need to select reducible expressions by initiating eager computations (it should be noted that this is different from speculative execution which would be the evaluation of both branches of a conditional and the discarding of one when the value of the conditional expression has been evaluated).

Non-strict functions are extremely useful in a variety of contexts: their main advantage is that they free programmers from many concerns such as evaluation order, parallelism and global synchronization. Computationally expensive values may be passed as arguments to functions and need not be computed if they are not needed. Data structures of indefinite length (e.g., streams) are a case in point.

2.2. Model of Execution: Partial Evaluation

Partial Evaluation (PE)[9] is another example of incomplete or non-strict information processing. PE is an automatic program transformation technique which allows the partial execution of a program when only some of its input data are available (static), and specializes it by pre-computing parts of the program that depend on specific parameter settings. The residual program is often faster that the original one. Adopting the notation of Mogensen and Sestoft,[10] we use p for the program text, and $[[\,p\,]]$ for the function computed by p. Consider a p which requires two inputs, x_1 and x_2. When the specific values d_1 and d_2 are given for the two inputs, we can execute the program, producing a result. Let us denote by $[[\,p\,]]_L\,[d_1, d_2]$ the result of running p with input values d_1 and d_2 on an L-machine. When only one input value d_1 is given, we cannot run p, but we can partially evaluate it, producing a version p_{d_1} of p specialized for the case where $x_1 = d_1$. A partial evaluator is a program *peval*, which performs a partial evaluation. Hence, given a program p and a d_1, it produces a residual program p_{d_1}, with one input such that $[[\,peval\,]]_L\,[p, d_1] = p_{d_1}$. It satisfies $[[\,p_{d_1}\,]]_T\,[d_2] = [[\,p\,]]_S\,[d_1, d_2]$ for all d_2 or $[[\,peval\,]]_L\,[p, d_1]_T\,[d_2] = [[\,p\,]]_S\,[d_1, d_2]$, where L is the language in which *peval* is written, S is the language of the source program, and T is the language of the target program.

Partial evaluation enables the construction of general highly parameterized software systems without sacrificing efficiency. Specialization turns

the general system into an efficient one specialized for specific parameter setting. It is similar in concept to, but in several ways stronger than, a highly optimizing compiler. Specialization can be done automatically or with human intervention. Partial evaluation may be considered a generalization of the usual evaluation.[11] Introduction of such a generalization appears to be very fruitful and allows the study of various programming concepts from a new point of view. Although simple in concept, partial evaluation has been used in such areas as compilation, scientific computing, meta-programming, computer graphic, pattern matching and others.[12–18] Note that such a form of computation is based on three different mechanisms: evaluation, partial evaluation, and residual program code generation. Partial evaluators exist for several languages: functional, logical, C, Fortran and etc. Two systems for C could be mentioned: C-mix[19] and the TEMPO system.[20] Tempo can perform source-to-source transformation at compile time as well as binary code generation at run time.

Ershov[11] introduced the concept of *program transformation* by a *transformation machine* for partial evaluation. Transformations, i.e., substitution of some syntactic or semantic construction of a program by another one (possibly simplified) are considered the instruction set of such a machine. The instructions not only perform some program evaluation, but also change (simplify) the program itself. If the initial input data are completely determined, then a sequence of transformations reduces the program to the single operator of the result output. If a part of the input data is not available (*dynamic*), then, at some stage, no further transformation can be applied and the program becomes a residual program which is ready to continue the evaluation as soon as dynamic inputs are available. Thus, in the transformation machine, there is no difference between evaluation and partial evaluation. Moreover, the process of transformations appears to be parallel and non-deterministic, since under some circumstances, the possible transformations can be performed in an arbitrary order. The process of transformation is, in fact, similar to the process of reduction.[21, 22]

The theoretical and application results obtained in the partial evaluation field are mostly associated with the fundamental properties of *sequential computations*. We will show here that they can also be included in *parallel computations* in a natural way. The parallel partial evaluation based on the DISs is implemented in the Abstract Network Machine (ANM),[1] where the incompleteness of the information is not restricted only to the dynamic part of program inputs. Relations with unknown or dynamic arity, incomplete mutually supplemented relations, non-strict functions, or incomplete structures with unknown elements and/or an unknown number of elements can be represented and manipulated. ANM is an abstract

model of transformational machine and does not distinguish between the three mechanisms: evaluation, partial evaluation, and residualizing. All of them are merely the result of a single transformation mechanism named network unification which is a modification of a graph reduction mechanism for fine grain parallel computation and processing of DISs.

2.3. Data Structures

The most common source of parallelism in many programming languages is in regular data structures such as array or vectors. Parallelism can be expressed for instance via a *forall* expression which takes all the elements of the array as input and produces another array or its reduction as the output. Several variations of data structures can be mentioned: strict arrays, arrays, tuples, I-Structures, Q-Structures, M-Structures, J-Structures, L-Structure Arrays, and DI-Structures, KL1 incomplete structures (see Table I).

The *strict array* is a conventional sequence of values. It can be used only after all the element bindings have been completed. That is, the binding of the array cannot take effect until all its components are bound to ground values (constants or static values). One way to look at it is that the array is created, filled, and until the producer uses it further, no one else has access to it. Under certain circumstances, such as modifying the array without losing determinacy, such a restriction is very useful. However, the producer of the array and its consumer work in a strict sequential fashion with coarse grain synchronization.

An *array* structure is a sequence of values. The number of its components is also known at run time. Index selectors are integer computable constants. The array is a non-strict structure because each element can be used without waiting for all element bindings to have completed. However, the producer of the array element and its consumer still work in a strict sequential fashion with fine grain synchronization at the level of individual elements. Different elements can be evaluated in parallel. All components, if bounded, are ground values.

Tuples structures[23] are also non-strict. They are sequences of values and created by specifying all of their components. The structure can be propagated even before the individual component bindings take place. Index selectors are integer constants specified at compile time. The non-strictness of tuples facilitates the construction of the structure and the computation of all its elements in parallel.

I-Structures. In a multiprocessor system, arbitrary (chaotic) writing and reading may occur because of parallel computations. When processors

Table I. Data Structures

	SA	A	T	I	J	L	DI	Q	M
Non-strict semantics		•	•	•	•	•	•	•	•
Single assignment semantics				•			•	•	
Sequence of components	•	•	•	•	•	•		•	•
Created by specifying all the components			•						
Number of components is known at run time	•	•		•	•	•	•	•	
Number of components can be incremented at run time							•		
All components if bounded are static values	•	•	•	•	•	•		•	•
Propagated before the individual components bindings take place (fine grain synchronization)		•	•	•	•	•	•	•	•
Access by index (index selector is computable constant)	•	•	•	•	•	•		•	•
Associative access by component name							•		
Index selector is specified at compile time			•						
Producer of the structure and its consumer work strictly sequentially (coarse grain synchronization)	•								
Producer of the component and its consumer work strictly sequentially (fine grain synchronization)		•	•		•	•			•
Producer of the component and its consumer work independently (a fetch request may arrive before the corresponding store completes)				•			•	•	
Asynchronous reading	•	•	•	•	•		•	•	
Split-phase transaction component access (deferred read)				•			•	•	
Multiple/parallel reading of the same component	•	•	•	•	•		•	•	
Emptying reading (consumption)						•			•
Ability to test a component state				•	•	•		•	•
Initialization to an empty state				•	•			•	•
Initialization to a full state						•			

communicate, they need to synchronize to ensure that valid data is used and to avoid race conditions. If some required data is not available, the processor must either wait for it to arrive (using coarse or fine grain synchronization), or switch to another thread, and occasionally poll for the arrival of the data or wait for an interrupt announcing the arrival of the data. ISs[24–28] are an attempt at solving this problem. A request for IS fetch may arrive at an individual IS element before the corresponding IS store completes. The number of components is known at run time and the run-time system should be capable of testing for a ground value (full location), an empty location, or one or several deferred reads.

Whether an I-fetch is deferred or not depends upon scheduling, and hence on the algorithm and machine configuration. ISs provide non-strict data access, fully asynchronous operations, and split-phase memory accesses. ISs facilitate the discovery of parallelism where timing sequences and determinacy issues would otherwise complicate its detection. In addition, they allow a regain of flexibility without losing determinacy. ISs can be extended to share data among several threads. The structure memory may contain multiple data or multiple continuations at the associated queued as a communication channel between threads. These extended I-Structures are called *Q-structures.*[29, 30]

M-Structures. Introducing a state into non-strict functional languages is an active area of research (e.g., monads[31] are mutable abstract data types), but such approaches come mostly at the cost of imposing some sequentially in order to preserve well-defined functional semantics. M-structures[25, 26, 32, 33] demonstrate an alternative approach, which does not come at the cost of forced sequentially. M-Structures are imperative data structures with implicit synchronization supporting atomic updates. A state bit is associated with every M-Structure element indicating whether it is empty or full. Correctness is defined in terms of the history of values held in an M-Structure cell. Serializability ensures that M-Structures updates appear atomic and sequential.[34]

J-Structures. Fine-grain data-level synchronization is expressed using data structures with accessors that implicitly synchronize them. In Refs. 35 and 36, these structures are called J-Structures and L-Structures. A J-Structure is a data structure for producer-consumer style synchronization inspired by I-Structures. A J-Structure is like an array, but each element has an additional state: full or empty. A reader of an element waits until the element's state is full before returning the value. A writer of a J-Structure element writes a value, set the state to full, and releases any waiting readers. The difference between J-Structures and I-Structures is that J-Structure elements can be reset to an empty state. As with M-Structures, it is possible to implement pair-wise producer-consumer synchronization using L-Structures: producers use synchronizing writes and consumers use locking reads. This can be viewed as an optimization of J-Structures which avoids having to reset elements when there is only a single consumer for each value produced.

L-Structures are arrays that support three operations: a locking read, a non-locking peek, and a synchronizing write.[35] A *locking read* waits until an element is full before emptying it and returning the value. A *peek* also waits until the element is full, but then returns the value without emptying

the element. A *synchronizing write* stores a value into an empty element, and sets it to full, releasing any waiting processes. The L-Structure allows mutually exclusive access to each of its elements. L-Structures are different from M-Structures in that they allow multiple non-locking readers. L-Structures are similar to J-Structures. The main differences are that L-Structure elements are initialized to full with some initial value, and that an L-Structure read of an element sets the associated full/empty bit to empty.

DI-Structures[37] also have non-strict semantics. A DI-Structure is an infinite set of components. It can be created by specifying its components statically or at run time. In either case, the final number of components is unknown, hence there is no ability to test for this number, and elements are selected by names (associative access). A structure is available (propagated) when none or not all of its element bindings have been completed. A fetch request may arrive at a DIS element before the corresponding DIS element appears. The request allocates the element of the structure if it does not exist and the continuation vector of the original request will be redirected for this element. Hence, DISs can be gradually made more complete. Producers of the whole DIS or producers of individual DIS elements and their consumers work in a pipeline fashion with unordered pipelining similar to that of ISs. In contrast to IS, however, the DIS runtime system is not capable of testing for the presence (availability) or status of the elements (value, an empty location, or deferred reads). Fetch and store requests are implemented as a network unification.[38]

KL1 incomplete structures. Incomplete data structures also play an important role in various logical programming techniques.[39] They are the source of flexible programming styles such as, for instance, the KL1 concurrent logic programming language based on Guarded Horn Clauses.[40, 41] Elements of the structure which do not have their value yet determined are written as variables. For example, an incomplete list whose first element is a symbolic atom and the rest of the list is underdetermined can be represented and manipulated. Variables in such structures can be shared among goals. Their values can be determined by goals other than the goal which has the whole data structure. Thus, incomplete structures can be gradually made more complete by some other goals. The incomplete message mechanism in KL1 is another example of the use of incomplete data structures. The message can be sent with the argument left undefined, to allow increment and decrement by an arbitrary amount by adding arguments to messages. Such message with undefined arguments is called an *incomplete message*. Using this style, a single stream can be used for both-way communication.

3. I-STRUCTURE AND I-STRUCTURE SOFTWARE CACHES

An IS has the property that an IS fetch request may arrive at an IS element before the corresponding IS store completes. An IS provides presence bits for each element. When an I-fetch occurs before an I-store, the deferred request is queued on a linked list of that particular IS element. When the I-store finally occurs, the system responds to the deferred reads by distributing the written value to the requests which have been received in the meantime. Whether an I-fetch is deferred or not depends upon scheduling, and ultimately on algorithm and machine configuration, but not upon synchronization issues.

Non-Blocking threads and MPI execution models have been proposed as an effective means to overlap computation and communication in distributed memory systems without any hardware support. Split-phase operations of IS are also used to enable the tolerance of request latencies by a decoupling between the initiators and the receivers of communication/synchronization transactions. However, the data locality of the distributed data is not exploited in MPI and Non-Blocking multithreading models; moreover, each request also incurs the cost of communication interface overhead.

The I-Structure Software Cache (ISSC)[42] provides a software caching mechanism for ISs to reduce the communication latency by caching the split-phase transactions, so that the system would combine the benefits of both latency tolerance and latency reduction in order to provide the capability to adapt to the unpredictable communication characteristics of MPP systems.

As we have already described, I-Structures are part of a single assignment memory system where multiple updates of a data element are not permitted, hence any copy of the data elements in a local cache will not be updated. Therefore, the cache coherence is already embedded in the concept of IS memory systems. It makes the design of a cache much simpler than for others non-strict structures without having to be concerned with the cache coherence problem. A remote memory request is sent out to the remote host only if the requested data is not available in the local ISSC. A write-through policy is adopted for the I-store operations. In the original ISSC design, a "cache advance" scheme was implemented. This means that a cache line is allocated upon a remote IS read miss while the following misses are queued in this pre-allocated cache line without forwarding any further requests to the actual location of the IS.

The cache system works as an interface between the user applications and the network interface. It intercepts all the remote read operations. Spatial locality is explored through a block request mechanism. Instead of

requesting a single element of the structure, an entire block of the data (cache line), including the requested element, is requested to the node that hosts the IS. Therefore, spatial data locality can also be exploited.

Simulation and experimental results which demonstrate the impact of the ISSC runtime system on the non-blocking multithreaded systems have been presented in Refs. 42–44. The performance of the four benchmarks with three versions: the original EARTH system, the EARTH system with IS support and the EARTH system with IS and ISSC support are compared for Matrix Multiplication, Conjugate Gradient, Hopfield and Sparse Matrix Multiplications. It was shown that ISSC increases the system utilization and improves the overall system performance by a factor of up to 95%. The cache advance scheme of ISSC also provides the adaptability to the unpredictable communication characteristics in the system. This makes ISSC achieve the same performance without being affected by variations in the communication latency.[45] When the cost to execute remote operations is very low, the difference in speedup between the versions with and without ISSC is not extremely high. However, a significant speedup of the version with ISSC over the version without ISSC are achieved in high network interface overhead environments simulated in the EARTH system. It is shown that when the communication overhead is greater than 100 μs is, the system with ISSC support runs almost 10 times faster than the other two versions in the benchmarks.

3.1. Partial Evaluation of Distributed IS and ISSC

IS and ISSC memory systems are well known and have been experimentally evaluated[45] for a variety of shared memory computers. In this section, we present experimental results which demonstrate the impact of the ISSC on distributed systems: we describe distributed MPI implementations of IS and ISSC on NUMA S2MP SGI Origin 2000 and on PC clusters. We show how they reduce the communication overhead in communication-intensive MPI applications. We also present optimization techniques based on partial evaluation.

The idea of using partial evaluation techniques for distributed applications has been considered in the recent past. Indeed, Sperber *et al.*[18] presented a distributed partial evaluation model that considers distributing the work of creating the specializations to distinct computational agents called specialization servers. Each specialization server can perform work on an arbitrary specialization, given its static configuration. As specialization produces more and more static configurations, it is necessary to distribute the work among the specialization servers present in the scenario.

Ogawa *et al.*[46] defined the OMPI (Optimizing MPI) system that removes much of the excess overhead by employing partial evaluation techniques and exploiting static information of MPI calls. Template functions are also utilized for further optimization. Dynamic caching optimization is used. In this technique, each MPI function would have its own cache that holds the arguments of its previous calls, and when there is a cache hit, the following optimizations become possible in order to eliminate the overhead: (1) Error checking could be eliminated. (2) Parameter checks for other dynamic optimizations could also be eliminated. (3) Message buffers could be reused. Another way of optimizing at the communication level has been proposed by Eicken[47] by using a simple communication mechanism named Active Messages. Each message contains its head and the address of a user-level handler, which is executed upon message arrival with the message body as argument. Under Active Messages, the network is viewed as a pipeline operating at a rate determined by the communication overhead and with a latency which is directly related to the message length and the network depth. The sender launches the message into the network and continues computing; the receiver is notified or interrupted upon message arrival and initiates the handler. This model minimizes the software overhead in message-passing machines. The efficiency of this model is due to the elimination of buffering beyond the network transport requirements.

In this paper, we describe a technique of optimization at the communication level based on the partial evaluation of local and remote memory accesses when some program inputs are known.

Split-phase transactions (*send-a-request, receive-a-request, send-a-value* and *receive-a-value*) together with the ability of deferring reads of IS elements allow partial program evaluation with ISs without losing determinacy. A *send-a-request* transaction specifies the information being requested and the process (memory) that owns the IS element and executes an MPI_Send instruction. In a *receive-a-request* transaction, the owner of the IS executes an MPI_Receive instruction, processes the request, checks the status of the IS element, and either *sends-a-value* to the requester by an MPI_Send or stores the request as a deferred read. When the data reaches the IS element, the owner of the element processes the pending reads and *send-a-value* by executing a series of MPI_Send instructions. A *receive-a-value* transaction completes an MPI_Receive and writes the value to the local memory of a requester.

Send-a-request and *receive-a-request* transactions can be completed if static information such as the number of IS elements in the input matrices is available. In that case, the programs can be partially evaluated even if the data bindings of the input matrices are not performed. In the residual program, only *send-a-value* and *received-a-value* transactions are left.

In the next section, we show that the cache system can reduce considerably the saturation of the interconnection network for MPI applications, and that partial evaluation can reduce the communication overhead even more.

3.2. Experimental Results

Experimental results are presented for two machines configurations:

- An SGI Origin 2000 with 10 MIPS R10000 processors running at 195 MHz with 1280 MB of main memory and a network bandwidth of 800 MBs/sec with a hypercube topology.
- A PC Cluster with 4 nodes, 8 processors Pentium III in a point-to-point interconnection by 10/100 Fast Ethernet, and 512 MB of memory in each node.

As benchmarks, we use the conventional dense matrix multiplication (MM) and the conjugate gradient (CG) algorithms with well-known characteristics (behavior) when executed on single and multiple processors. In the MM benchmark, each 128×128 matrix is implemented as an IS with double precision values. The MM has excellent temporal locality: two input matrices are constantly referenced during the computation. Data locality is reduced when processing elements are added due to the initial data distribution. The steepest descent method, also known as the gradient method, is the simplest example of gradient-based methods. This loose coupling algorithm solves a system of 256 equations with 256 variables. Matrices are also represented by ISs.

The problem is chosen as sufficiently small so that it can easily fit into a single PC and yet display a speed up on a multiprocessor. Indeed, the larger a problem, the better the performance improvements. This is particularly true when the memory requirements of the application cannot be satisfied in a single processor machine but can be distributed across multiple processors. Further, increasing the size of the problem transforms the application into a computationally intensive application which has obvious parallelization advantages. Our objective here is to demonstrate that our approach will yield speed-ups, even for small problems.

The performance of the benchmarks with four different MPI implementations (written in C) have been compared:

1. *IS.* In this implementation no optimization technique is applied. Remote requests are managed by the IS memory system.
2. *IS_Residual.* A residual program differs from the original IS program in that some communications of the IS program are

performed during the partial evaluation step, hence, they are not executed in the residual program and do not contribute to the execution time. All matrices elements requests (*send-a-request*, local and remote) are performed. Also, all *received-a-request* transactions are completed by setting the IS elements to a deferred state. The residual program contains only deferred memory requests. Hence, it executes only *send-a-value* and *receive-a-value* transactions.

3. *ISSC.* The ISSC system is used.
4. *ISSC_Residual.* Both optimization techniques ISSC and partial evaluation are applied. The residual program differs from the original *ISSC* program in that all matrices elements requests (local and remote) are performed during the partial evaluation step; hence, they are not executed in the residual program. The residual program must only bind the IS elements and complete the deferred reads. Hence, it executes only *send-a-value* and *receive-a-value* transactions.

To evaluate the impact on performance of partial evaluation techniques, a time optimization S_p^0 and message optimization M_p^0 coefficients are calculated. $S_p^0 = T_0/T_r$ is the ratio of the time T_0 taken by the original program over the time T_r taken by the residual one where some operations are eliminated by partial evaluation. It shows the degree of reduction of the execution time of the residual program over the original one. $M_p^0 = M_0/M_r$ is the ratio between the total number of messages M_0 sent by the original program and the total number of messages M_r produced in the residual program. It shows the degree of message elimination by partial evaluation. These coefficients show two different aspects of program improvement by partial evaluation. In general, the larger the ratios, the better the optimization.

3.2.1. Message Reduction

Table II shows the number of messages of *IS, IS_Residual, ISSC*, and *ISSC_Residual* programs for MM and CG benchmarks obtained for different numbers of processors.

For both programs, the total number of messages is increased when more processing elements are added. It is a result of the distribution of the data between processors. We can also see the impact of ISSC in the reduction of the number of messages in the system. The number of messages during the execution of MM is reduced significantly by a factor of more than 16 compared to the original quantity of *IS* program without *ISSC* support, while the number of messages during the execution of CG is only decreased by a factor of 1.9.

Table II. Number of Messages in IS, IS Residual, ISSC, and ISSC Residual Programs for the MM and CG Benchmarks with Variable Numbers of Processors

Program	Implementation	Number of messages		
		2 PEs	4 PEs	8 PEs
MM	IS	2097152	3145728	3670016
	IS-Residual	1048576	1572864	1835008
	ISSC (CB = 1)	32768	98304	229376
	ISSC-Residual (CB = 1)	16384	49152	114688
CG	IS	134144	205824	250880
	IS-Residual	67072	102912	125440
	ISSC (CB = 1)	67584	104448	129024
	ISSC-Residual (CB = 1)	33792	52224	64512

This difference is because the cache allows for the processing of more than 90% of all requests locally in MM and only 48% in CG. Hit ratio of the MM and CG programs for different numbers of processes and different cache block sizes is shown in Fig. 1.

From Table II, the impact of partial evaluation in the reduction of messages can be also seen. The number of messages of the residual programs *IS_Residual* and *ISSC_Residual* for both, MM and CG, is reduced by a factor of two compared to the original *IS* and *ISSC* programs respectively.

Figure 2 shows M_p^0 versus the cache block size for eight processors. Increasing the cache block size decreases the number of *request-a-value* transactions (request a block), and keeps the number of *receive-a-value* transactions constant, which results in decreasing M_p^0 from 2 to 1.25 with an increasing cache block size to 8. All elements of the block should not be

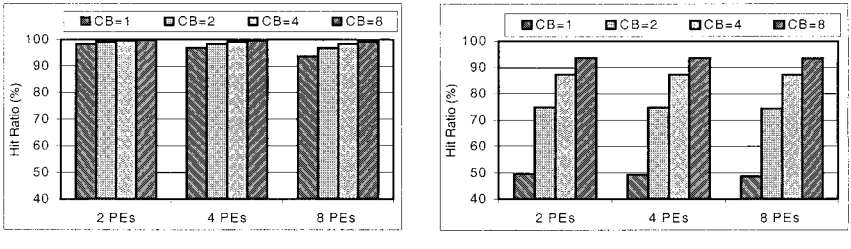


Fig. 1. Hit ratio of the MM (a) and CG (b) programs for different number of processes and different cache block sizes.

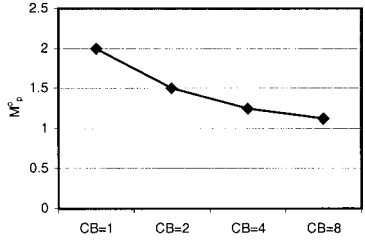


Fig. 2. Message optimization (M_p^0) of ISSC by partial evaluation varying the size of cache block for 8 PEs.

sent in one message in order to keep the property of asynchronous bounding of IS elements.

We can conclude that more than 95% of the messages in MM and more than 75% in CG can be reduced using our ISSC. The rest of messages are reduced by a factor of from 1.2 to 2 (depending on the block size) by partial evaluation. This considerable reduction decreases the traffic in the interconnection network and improves the overall system performance.

3.2.2. Time Reduction

Figure 3 shows the relative speedup of MM and CG with *ISSC* as compared with programs without ISSC in both the SGI Origin 2000 and the cluster of PCs with varying numbers of processors and cache block size. For more than one processor, a significant reduction of the number of messages in MM with *ISSC* makes it more than four times faster in the SGI Origin 2000 (Fig. 3a) and more than ten times faster in the Cluster (Fig. 3c) than the corresponding IS program. Fewer messages for CG programs result in a lower speedup of the program with ISSC. CG with ISSC is about twice faster than without ISSC in both computers (Figs. 3b and 3d).

Figure 4 shows the relative speedup obtained by parallelization of programs with ISSC on different numbers of processors for MM and CG. Note that the degradation from ideal speedup is not significant for MM because ISSC efficiently exploits temporal and spatial data locality of the algorithm, while CG has low re-use rate of data which results in its lower speedup.

Figure 5 shows S_p^0 obtained for our benchmarks with and without the cache system, varying the number of processors. Program partial evaluation reduces the execution time of MM with ISSC by a factor more than four in the SGI Origin 2000 (Fig. 5a) and 2.5 in the Cluster (Fig. 5c).

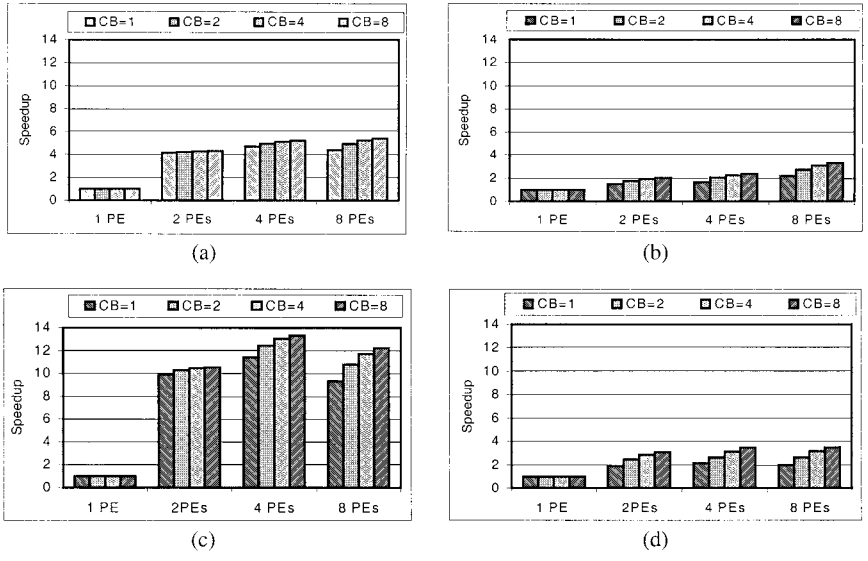


Fig. 3. Speedup of ISSC over IS, for MM (a), (c) and CG (b), (d) benchmark programs varying the number of processors and the cache block size. Results are presented for SGI Origin 2000 (a) and (b) and for a cluster of PCs (c) and (d).

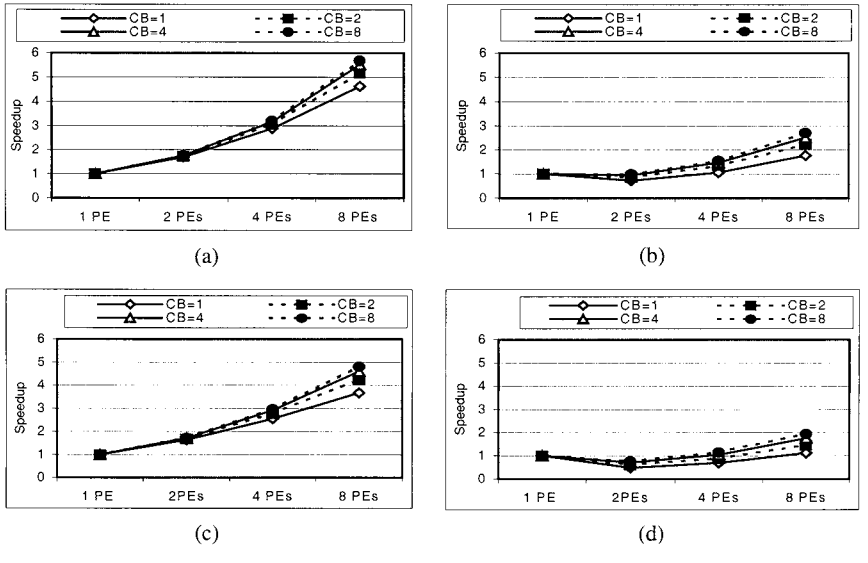
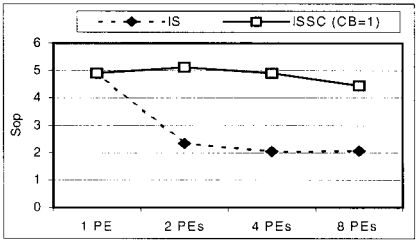
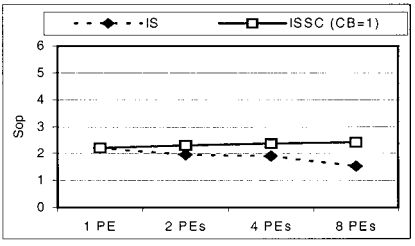


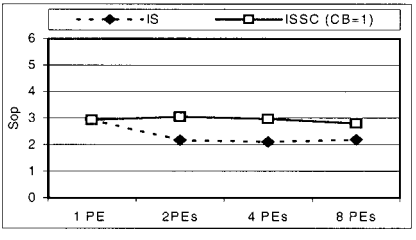
Fig. 4. Relative speedup of ISSC versus the number of PEs and the cache block size for MM (a) and for CG (b) in SGI Origin 2000, and for MM (c) and for CG (d) in Cluster.



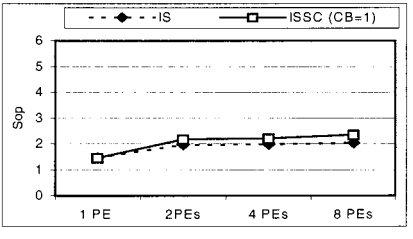
(a)



(b)



(c)



(d)

Fig. 5. S_p^0 for MM (a), (c) and CG (b) and (d) benchmark programs with and without cache system versus the number of processors. Results are presented for SGI Origin 2000 (a) and (b) and for a cluster of PCs (c) and (d).

There are small changes when the number of processors is made to vary. A lower S_p^0 is obtained for MM with IS. It is about two, which is directly due to the reduction in the number of messages by a factor of two (see Table II). The execution time of CG with IS has been also reduced by a factor of two in both the ISSC and IS implementations (Figs. 5b and 5d). Due to the low temporal data locality of the CG algorithm, ISSC does not significantly reduce the execution time.

The impact of a varying number of processors and block size on S_p^0 is shown in Fig. 6. When the block size is increased, the number of messages that can be eliminated with partial evaluation is reduced (see Fig. 2). The reduction is less for larger blocks. Increasing the number of processors also results in less reduction for any block size. This is directly due to the increase in communication overhead.

The reduction of the number of messages is more significant for systems with large latency such as clusters of PCs with commodity NIC. We show that a portion of the remote or local memory accesses can be partially evaluated in the programs by using the split-phase feature of I-Structures and deferred read mechanisms. These observations mean that ISSC is highly

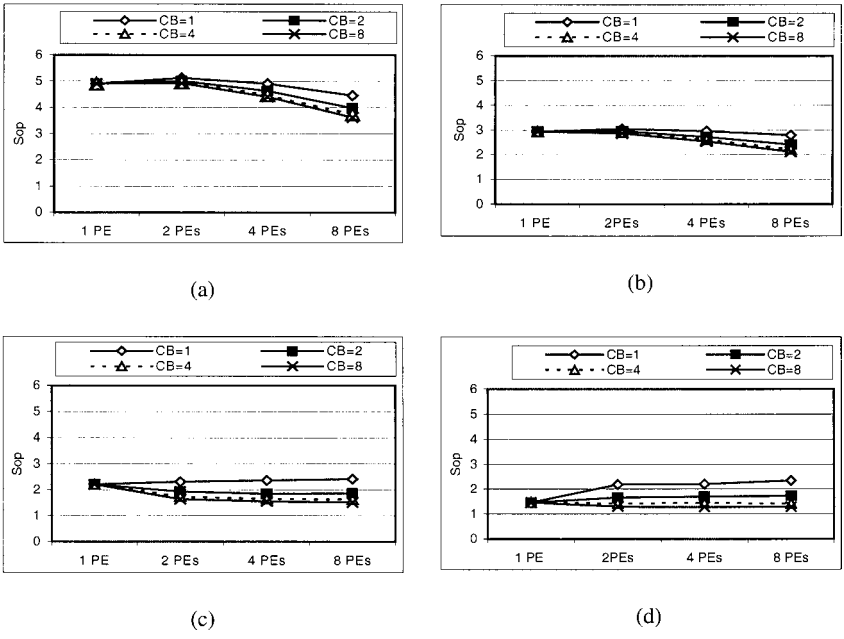


Fig. 6. S_p^0 of ISSC by partial evaluation for MM (a) and (c) and CG (b) and (d) varying the number of processors and the size of cache block. Results are presented for SGI Origin 2000 (a) and (b) and a cluster of PCs (c) and (d).

suited for concurrent environments where readings and writings can be executed in any order and for optimization by partial evaluation.

4. DI-STRUCTURES

The incompleteness of IS is considered on the level of elements. In this section, we show that a higher degree of incompleteness on the level of whole data structures leads to fine-grain parallel computation where more optimization is possible by partial evaluation. Dynamic I-Structures (DISs) differ from I-Structures in that they have an unrestricted number of elements which may increase the degree of incompleteness. The total number of components is unknown at compile and run time while the selectors are the names of elements (associative access). A fetch request (such as in the IS) may arrive at a DIS element before the corresponding DIS element appears. In contrast to what would happen in an IS system, it does not cause a deferred read, but creates instead the element of the structure if it does not yet exist. Hence, DISs can be gradually made more complete by

successive reads. Also, in contrast to the IS, the DIS run-time system is not capable of testing for the presence (availability) or the status of the elements (value, empty location, or deferred read). Fetch and store requests are implemented as a DISs unification (network-unification), which is a modification of a reduction (substitution) in a graph reduction mechanism for fine grain parallel computation and exploiting the non-strict features of the DIS.[38, 48] These non-strict features make DISs highly suited for the exploitation of fine-grain parallelism and partial evaluation.

Stepanov *et al.*[37] implemented DISs for their simulated Abstract Network Machine (ANM). In their implementation, a DIS memory system named *Associative Network* is used. The memory is represented as a directed graph of a special kind with labeled edges. Three types of vertexes namely atoms, DISs, and empty objects are distinguished. Each element of a DIS is a pair $[A\ X]$, where A is the element's name and X is the element's value (some other vertex). There is a limitation on the DIS construction: it cannot contain two elements with the same name. A computational process of the ANM consists of asynchronous local transformations of the network with a single operation: network-unification. The ANM performs a partial evaluation when the information given is incomplete and synthesizes a residual parallel program from the residual directed graph obtained after finishing the transformations.[49]

In this section, we provide experimental results to demonstrate the impact of partial evaluation on the performance of parallel programs as well as optimization-related issues. One of our premises is that more incompleteness (by data structures) will foster more opportunities for optimization by partial evaluation. Our choice of ANM is dictated by our desire to "abstract" ourselves from "practical" overhead considerations and produce observations on the quality of optimization which are strictly independent of the ultimate computing platform.

We show that processing non-strict information in the ANM (a) exposes fine-grain parallelism of programs which are free of any explicit description of a parallel or sequential control structure; (b) optimizes programs by considerable reduction of calculations; and (c) reveals in some cases an inherent parallelism irrespective of the style of description of an algorithm and sequence of declarations in a code.

Experimental results are presented for ANM v.2.1[38] for unit execution time operations. As a benchmark, we use the well-known vector Insertion Sort and list Quick Sort algorithms. The programs are in the PARS[50] declarative language. Two implementations of Insertion Sort algorithm are presented: *InsSort1* is based on a recursive algorithm that is to take an element of a vector, sort recursively all the rest elements of the vector, and insert the element to the sorted vector. *InsSort2* is based on a

similar recursive algorithm that is to take each element of the input vector and insert it to the sorted vector that initially is empty.

The performance of the benchmarks with 6 different implementations are compared: *InsSort1, InsSort1_Residual, InsSort2, InsSort2_Residual, Qsort,* and *Qsort_Residual.*

Time of the sequential execution T_1, time of execution on p processors T_p, the speedup S_p, the efficiency E_p, and the cost C_p are calculated. To estimate the impact of partial evaluation on the quality of the residual programs generated, the following optimization coefficients are calculated: code size, number of operations, execution time, maximum number of processors, and execution cost. Coefficients are the ratios of the respective parameters measured for an original program over the parameters obtained for a residual program. In general, the larger the ratios the better the optimization.

Table III summarizes the result for the *InsSort1, InsSort2*, and *Qs* programs. It shows the different characteristics of the programs evaluation when all the input data are given (Table III, Original). It also shows the characteristics of the process of partial evaluation when only N is given (Table III, Partial evaluation). As a result of this process, the residual program is synthesized. The results of the evaluation of this residual program are given in Table III (Residual), and the optimization coefficients obtained are shown in Table III (Ratio).

We see that partial evaluation has eliminated useless calculations significantly, reduced overhead and made computations more effective. The number of operations is decreased in residual programs as compared with the original ones by a factor of 24.39 for *InsSort1*, and 31.62 for *InsSort2*. The program speedup is increased by a factor of 2.25 and 2.68 respectively, while the cost is decreased by 61.20 and 33.60 times respectively. Also, we see that the equivalent residual parallel programs with the same number of operations and degree of parallelism are synthesized as a result of partial evaluation of *InsSort1* and *InsSort2* programs (Table III, Residual: *InsSort1, InsSort2*).

For these benchmarks, a hidden inherent parallelism has been revealed, irrespective of the style of an algorithm description and the sequence of declarations in program codes. A reduction in the amount of computational resources can also be obtained by this technique. The maximum number of processors used for evaluation of the residual program is reduced when compared to the original program by a factor of 27.20 for *InsSort1*, and 12.55 times for *InsSort2*. Table III (Partial evaluation) shows that the process of partial evaluation is also parallel.

InsSort1 and InsSort2 are the programs, which are special, in that they are, for the most part, data-independent, meaning that the parallelism of

Table III. Parallelization and Performance Optimization

Program		InsSort1 ($N = 24$)	InsSort2 ($N = 24$)	QSort ($L = 24$)
Original	Size	2700 Byte	3887 Byte	2904 Byte
	T_1	27287	35370	1671
	T_p	126	150	101
	P	1496	690	37
	S_p	216.56	235.80	16.54
	E_p	14.48	34.17	44.72
	C_p	188496	103500	3737
Partial evaluation (N is available)	T_1	26182	34265	256
	T_p	76	150	77
	P	1495	682	11
	S_p	344.50	228.43	3.32
	E_p	23.04	33.49	30.22
	C_p	113620	102300	847
Residual	Size	123 Byte	122 Byte	13,768 Byte
	T_1	1119	1119	1441
	T_p	56	56	69
	P	55	55	47
	S_p	19.98	19.98	20.88
	E_p	36.33	36.33	44.43
	C_p	3080	3080	3243
R size		21.95	31.86	0.21
A operation		24.39	31.61	1.16
T time		2.25	2.68	1.46
I processor		27.20	12.55	0.79
O cost		61.20	33.60	1.15

operations to be evaluated is independent of the actual values being manipulated. The size of the problem is available in our experiments; hence the programs are well optimized by partial evaluation.

For another type of problems which are data-dependent, meaning that most operations to be evaluated are dependent of the actual values being manipulated, the partial evaluation does not give significant results. Table III shows the results of parallel evaluation and the optimization of such data-dependent list quick sort algorithm *Qsort*. It shows that partial evaluation cannot eliminate useless computations in *Qsort* to any great degree. The program is specialized to the value of list length, but the computation depends on pivot values that are not available. The number of operations is decreased only by a factor of 1.16, speedup is increased by a factor of 1.46, and the cost is decreased 1.15 times. Nevertheless, the

program parallelism automatically exposed is high when the all input data are given. In our example, when the list length equals 24 and the number of processor elements is 47, the speedup of the residual program is 20.88, with an efficiency of 44.43 percent. The number of operations, the running time, and the cost of the residual program are 1441, 69, and 3243, respectively (Table III, *QSort*, Residual).

5. RELATED WORK

Dennis and Gao[51] proposed a cache management scheme for their abstract shared-memory computer system with a data-flow program execution model. In that design, the size of a cache line is a single IS element. Therefore, spatial locality is not exploited and no deferred read-sharing problem occurs in their design. Moreover, no implementation and or performance evaluation had been done. Culler *et al.*[52] implemented the idea of IS caching in a software manner on the *Id90* compiler for their *Threaded Abstract Machine* (TAM) implemented on the CM-5. In the implementation, the unit of a cache block is also a single IS data element. Only temporal data locality had been exploited. With a cache block size of a single IS data element, no deferred read-sharing problem occurs. This made the design comparatively easier, like cache replacement, deferred read handling, etc. A similar work was presented in Ref. 53 where a multithreaded architecture SMALL that is capable of exploiting both coarse-grain parallelism and fine-grain instruction level parallelism in a program was proposed. To reduce both network traffic and the latency in accessing remote locations, to make the system scalable, a distributed data structure cache DS-Cache was designed. Coherence in caches is maintained by using a special scheme for ISs, and software cache coherence mechanism for regular data structures. Lin *et al.*[28] shown that spatial data locality does play an important role in the performance improvement. Moreover, temporal data locality could be easily utilized by the programmer or the compiler without implementing the IS caching. IS software cache (ISSC) to cache IS elements was implemented in the EARTH Threaded-C language on the MANNA machine.[28, 54, 55] Hyong-Shik Kirn[56] proposed an organization and an operation scheme of an IS cache in frame-based multithreading. With IS caches, the performance impact was found to be threefold: reduction of the average latency, increase of quantum size, and enhancement of frame parallelism. Among them, the enhancement of frame parallelism seems most important. Kavi *et al.*[57] proposed a design of cache memories for the explicit token store (ETS) model of data-flow systems with IS memory system. Basically, a write-back cache was adopted in the design. However, to guarantee the service of pending requests on IS

memory; the data elements were written back to IS when there was any pending request. To implement this, extra storage in the local caches was needed and extra message passing on the network had to be introduced.

Sperber *et al.*[18] presented a distributed partial evaluation model in the form of specialization servers. Each server can perform work on specialization, given its static configuration. Ogawa *et al.*[46] designed an Optimizing MPI system that removed much of the excess overhead by employing partial evaluation techniques.

The philosophy underlying the ANM project was closely coupled with the data-flow model, ISs and languages like Id, pH. The transformation machine was considered by Ershov.[11] Several authors considered graph reduction machines and mechanisms for implementation of functional languages, lazy evaluation, lambda calculus.[21, 58, 59, 60] Using partial evaluation for the automatic program parallelization matched the goal of the partial evaluation-based compiler for the Supercomputer Toolkit.[61] The central focus of this study was to expose and to find a way to exploit extremely fine-grained parallelism. Traditionally, parallelization techniques include the control-flow and data-flow analysis, vectorizing, static and dynamic scheduling. Tchernykh *et al.*[49] presented an extraction of the implicit program parallelism technique base on on-line partial evaluation.

6. CONCLUSIONS AND FUTURE WORK

In this paper, the design and implementation of distributed ISSC for PC clusters and SGI Origin 2000 have been presented. We have shown that in a distributed memory systems, the split-phase memory access schemes of MPI and IS allow for an overlap of long communication latencies and useful computations. In addition, ISSC provides a software caching mechanism to further reduce the communication latency by caching the split-phase transactions. Therefore, the system would combine the benefits of latency tolerance and latency reduction. It is more significant for systems with large latency such as PC clusters with commodity NICs. By exploiting both the temporal and the spatial global data locality, the number of network packets decreases dramatically. ISSC not only helps the system by exploiting the data locality in different types of applications, but it also reduces the number of network packets in the network. The distributed ISSC memory system significantly reduces the network latency and makes the system more scalable. Experimental results show the speedup of the distributed ISSC runtime system over IS for different number of processors and cache block sizes. A significant reduction of the messages made ISSC programs more than four times faster corresponding IS programs.

Plain versions (non IS-based) of the programs for small problem sizes display better performance than IS because of the IS access overhead. We have shown that, even though ISSC incurs an additional overhead for its operations, by taking advantage of the global data locality in applications and with the amount of communication interface overhead saved by ISSC, the ISSC improves system performance in the PC Cluster platforms.

We have presented an optimization technique based on partial evaluation to decrease the overhead and improve performance of IS and ISSC programs. Experiments have shown that if we take the total number of messages in the original program as 100%, then introducing ISSC reduces up to 90% of messages in some cases. The amount of messages that is left (10%) can be reduced twice by partial evaluation, so only 5% of original messages are left. Experimental results also show that the total execution time can be reduced by a factor of 5 with ISSC and partial evaluation together. We have shown than, in distributed memory systems, it was possible to reduce the traffic of the interconnection network and improve the performance of the entire system by these techniques.

We have also demonstrated that additional non-strict features of DISs would lead to the parallel computation where more optimization can be done by partial evaluation. Application of DISs and partial evaluation techniques for optimization are effective in numerically-oriented scientific programs, since the programs tend to be mostly data-independent. They are also useful when solving problems where complicated dynamic data structures (such as lists, trees, etc.) are involved and hand or compiler detection of parallelism is difficult.

Our main results can be summarized as follows:

1. The addition of ISSC to MPI results in increased robustness to latency variation and the speedup obtained with ISSC increases for larger numbers of processors.
2. The ISSC significantly reduces the amount of traffic in the network.
3. The performance of the programs with ISSC is improved for all benchmarks for PC cluster and SGI Origin 2000.
4. Partial evaluation techniques reduce the number of MPI messages in the system and increase the speedup of the programs.

ACKNOWLEDGMENTS

The authors are pleased to acknowledge the anonymous referees whose valuable remarks and comments helped improve the paper. Part of this work was supported by CONACYT (Consejo Nacional de Ciencia y

Tecnología de México) under Grant #32989-A, and by the National Science Foundation under Grants #CCR-0073527 and #INT-0223647.

REFERENCES

1. A. M. Stepanov, *Parallel Computation on Associative Networks*, Preprint, AS USSR. Institute of Precise Mechanics and Computer Technology, Vol. 2, Moscow, p. 53 (1991) (In Russian).
2. S. Wray and J. Fairbaim, Non-Strict Languages—Programming and Implementation, *Comput. J.* **32**(2):l42–151 (1989).
3. Y.-H. Wei and J.-L. Gaudiot, Lazy Evaluation of FP Programs: A Data-Flow Approach, *Proc. of the Int'l. Conf. on Fifth Generation Computer Systems* (1988).
4. G. Tremblay and G. R. Gao, The Impact of Laziness on Parallelism and the Limits of Strictness Analysis. In *Proceedings High Performance Functional Computing*, W. Bohm and J. T. Feo (eds.), pp. 119–133 (April 1995).
5. R. Bird, *Introduction to Functional Programming using Haskell*, 2nd edn., Prentice Hall Press, 460 pp. (1998).
6. R. Nikhil and M. Arvind, *Implicit Parallel Programming in pH*, Morgan Kaufmann Publishers, p. 400 (2001).
7. M. Amamiya and R. Hasegawa, Data-Flow Computing and Eager and Lazy Evaluations, *Computing* **2**(2):105–129 (1984).
8. M. Amamiya, R. Hasegawa, and H. Mikami, List Processing with a Data-Flow Machine, *Lecture Notes in Comput. Sci.* **147**:165–190 (1983).
9. N. Jones, An Introduction to Partial Evaluation, *ACM Comput. Surv.* **28**(3):480–503 (1996).
10. T. Mogensen and P. Sestoft, *Partial Evaluation*, An Article for Encyclopedia of Computer Science and Technology, FTP version (1996).
11. A. P. Ershov, Mixed Computation: Potential Applications and Problems for Study, *Theoret. Comput. Sci.* **18** (1982).
12. I. Bjorner, A. Ershov, and N. Jones, *Partial Evaluation and Mixed Computation*, North-Holland (1988).
13. C. Consel and O. Danvy, Partial Evaluation of Pattern Matching in String, *Inform. Process. Lett.* **30**(2):79–86 (1989).
14. C. Consel and O. Danvy, Static and Dynamic Semantic Processing, *ACM Symposium on Principles of Programming Languages*, pp. 14–23 (1991).
15. J. Jørgensen, Generating a Compiler for a Lazy Language by Partial Evaluation, *ACM Symposium on Principles of Programming Languages*, pp. 258–268 (1992).
16. N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice–Hall (1993).
17. P. Sesyoft and H. Sondergaard (eds.), *Special Issue on Partial Evaluation and Semantic-Based Program Manipulation (PEPM'94)* (Lisp and Symbolic Computation, Vol. 8, No. 3) (1995).
18. M. Sperber, H. Klaeren, and P. Thiemann, Distributed Partial Evaluation. In *PASCO'97: Second Int'l. Symposium on Parallel Symbolic Computation*, Erich Kaltofen (ed.), pp. 80–87, Maui, Hawaii, World Scientific Publishing Company (1997).
19. http://www.diku.dk/research-groups/topps/activities/cmix/
20. http://compose.labri.fr/prototypes/tempo/
21. P. Kumar, J. P. Gupta, and S. C. Winter, CTDNET III—An Eager Reduction Model with Laziness Features. In *Abstract Machine Models for Highly Parallel Computers*, J. R. Davy and P. M. Dew (eds.), pp. 103–117 (1995).

22. J. P. Gupta, S. C. Winter, and D. R. Wilson, CTDNet—A Mechanism for the Concurrent Execution of Lambda Graphs, *IEEE Trans. Soft. Eng.* **15**:1357–1367 (1989).
23. J. Jaakko, Tuples and multiple return values in C++, TUCS Technical report No. 249, Turku Centre for Computer Science (March 1999). ISBN 952-12-0401.
24. Arvind, R. S. Nikhil, and K. K. Pingali, I-Structures: Data Structures for Parallel Computing, *ACM Transaction on Programming Languages and Systems* **11**(4):598–632 (Oct. 1989).
25. P. S. Barth, Using Atomic Data Structures for Parallel Simulation. In *Proceedings of the Scalable High Performance Computing Conference, Williamsburg, VA, April 27* (1992).
26. S. Sur and W. Böhm, *Efficient Declarative Programs: Experience in Implementing NAS Benchmark FT*, Technical Report CS-93-128, Colorado State University (October 1993).
27. X. Shen and B. S. Ang, Implementing I-Structures at Cache Coherence Level, *Proceedings on the 5th Annual MIT Student Workshop on Scalable Computing, MIT* (1995).
28. W.-Y. Lin and J.-L. Gaudiot, I-Structure Software Cache—A Split-Phase Transaction Runtime Cache System, *Proceedings of PACT '96 Boston, MA, Oct. 20–23* (1996).
30. M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and S. Sekiguti, Distributed Data Structure in Thread-Based Programming for a Highly Parallel Dataflow Machine, EM-4. *Proc. of ISCA 92 Dataflow Workshop* (1992).
31. P. Wadler, Monads for Functional Programming. In *Advanced Functional Programming*, J. Jeuring and E. Meijer (eds.), Springer Verlag, LNCS 925 (1995).
32. P. S. Barth, R. S. Nikhil, and Arvind, M-Structures: Extending a Parallel, Non-strict.—Computation Structures, *Proceedings on Functional Programming and Computer Architecture, Cambridge, MA, August 28–30* (1991).
33. S. Sur and W. Böhm, Functional, I-Structure, and M-Structure Implementations of NAS Benchmark FT, *Proceedings of the Int'l. Conf. on Parallel Architecture and Compilation Techniques (PACT'94)* (August 1994).
34. I. Attali, D. Caromel, Y.-S. Chen, J.-L. Gaudiot, and A. L. Wendelbom, Enhanced Functional and Irregular Parallelism: Stateful Fucntions and Their Semantics, *Int. J. Parallel Progr.* **29**(4) (August 2001), in press.
35. D. Kranz, B.-H. Lim, A. Agarwal, and D. Yeoung, Low-Cost Support for Fine-Grain Synchronization in Multiprocessors. In *Multithreaded Computer Architecture: A Summary of the State of the Art*, R. A. Iannucci, G. R. Gao, and R. H. Halstead, Jr. (eds.), Kluwer Academic Publishers (1992).
36. A. Agarwal, R. Bianchini, D. Chiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, The MIT Alewife Machine: Architecture and Performance, *Proceedings of the 22nd Annual Int'l. Symposium on Computer Architecture, ISCA '95, June 22–24, Santa Margherita Ligure, Italy*, pp. 2–13 (1995).
37. A. M. Stepanov, A. N. Tchernykh, A. I. Lupenko, and N. G. Tchernykh, Parallel Computation on Associative Network. In *Proceedings of MPCS '96 MFCS'96 Second Int'l. Conf. on Massively Parallel Computing Systems*, IEEE Computer Society Press, pp. 190–197 (1996).
38. A. Tchernykh, A. Stepanov, A. Rodríguez, and I. Scherson, Parallel Computation in Abstract Network Machina, *Revista Iberoamericana de Investigacion "Computacion y Sistemas" v. TV*, No. 4, pp. 143–157 (2000).
39. K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine, *Comput. J.* **33**(6):494–500 (1990).
40. K. Ueda, Guarded Horn Clauses. In *Concurrent Prolog: Collected Papers*, E. Shapiro (ed.), MIT Press, Vol. 1, pp. 140–156 (1987).
41. K. Ueda, Designing a Concurrent Programming Language. In *Proceedings of an Int'l. Conf. organized by the IPSJ to Commemorate the 30th Anniversary (InfoJapan'90)*, Information Processing Society of Japan, pp. 87–94 (October 1990).

42. W.-Y. Lin, J. N. Amaral, J.-L. Gaudiot, and G. R. Gao, *Caching Single-Assignment Structures to Build a Robust Fine-Grain Multi-Threading System*, Technical report, Dept. of E.E.-Systems, University of Southern California (July 1999).
43. W.-Y. Lin, J.-L. Gaudiot, J. N. Amaral, and G. R. Gao, Performance Analysis of the I-Structure Software Cache on Multi-Threading Systems, *19th IEEE Int'l. Performance, Computing and Communication Conference, IPCCC2000, Phoenix, Arizona, Feb. 20–22* (2000).
44. W.-Y. Lin, J. N. Amaral, J.-L. Gaudiot, and G. R. Gao, Caching Single-Assignment Structures to Build a Robust Fine-Grain Multi-Threading System, *Int'l. Parallel and Distributed Processing Symposium, IPDPS2000, Cancun, Mexico May 1–5* (2000).
45. J. N. Amaral, W.-Y. Lin, J.-L. Gaudiot, and G. R. Gao, Exploiting Locality in Single Assignment Data Structures Updated Through Split-Phase Transactions, *Cluster Comput. J.* **4**(4) (October 2001).
46. H. Ogawa and S. Matsuoka, OMPI: Optimizing MPI Programs Using Partial Evaluation, *Proceedings of the 1996 IEEE/ACM Supercomputing Conference, Pittsburgh* (November 1996).
47. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, Active Messages: A Mechnisim for Integrated Communication and Computation, *Proceedings of the 19th Int'l. Symposium on Computer Architecture*, pp. 256–266 (May 1992).
48. A. M. Stepanov, A. N. Tchernykh, A. I. Lupenko, and N. G. Tchernykh, Dynamic Partial Evaluations as Declarative Program Parallelization and Optimization Technique, *Information Technology and Computer Systems* **1**(4):32–41 (1997).
49. A. N. Tchernykh, A. M. Stepanov, A. I. Lupenko, and N. G. Tchernykh, Extraction and Optimization of the Implicit Program Parallelism by Dynamic Partial Evaluation, *pAs'97 The Second Aizu Int'l. Symposium on Parallel Algorithms/Architecture Synthesis*, pp. 332–339, IEEE Computer Society Press (1997).
50. A. Stepanov and A. Lupenko, *Programming for ANM, Institute of Precise Mechanics and Computer Technology RAS*; 3, p. 53, Moscow (1991).
51. J. B. Dennis and G. R. Gao, *On Memory Models and Cache Management for Shared-Memory Multiprocessors*, CSG MEMO 363, Laboratory for Computer Science, MIT (March 1995).
52. D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken, Empirical Study of a Dataflow Language on the CM-5. In *Advanced Topics in Dataflow Computing and Multithreading*, G. R. Gao, L. Bic, and J.-L. Gaudiot (eds.), pp. 187–210, IEEE press (1994).
53. R. Govindarajan, S. Nemawarkar, and P. LeNir, Design and Performance Evaluation of a Multithreaded Architecture. In *Proceedings of the First Int'l. Symposium on High-Performance Computer Architecture, Raliegh*, pp. 298–307 (1995).
54. W.-Y. Lin and J.-L. Gaudiot, Exploiting Global Data Locality in Non-Blocking Multithreading Architectures, *Proceedings of ISPAN '97, Taipei, Taiwan* (December 1997).
55. W.-Y. Lin and J.-L. Gaudiot, The Design of an I-Structure Software Cache System, *Proceedings of MTEAC'98, Las Vegas*, February 1–4.
56. H.-S. Kim, S. Ha, and C. S. Jhon, Performance Impacts of Caching I-Structure Data on Frame-Based Multithreaded Processing, *Proceedings of the High-performance Computing on the Information Superhighway, HPC-Asia '97* (1997).
57. K. M. Kavi, A. R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam, Design of Cache Memories for Multithreaded Dataflow Architecture. In *ISCA 95*, pp. 253–264 (1995).
58. J. Darlington, M. Cripps, T. Field, P. Harrison, and M. Reeve, The Design and Implementation of ALICE: A Parallel Graph Reduction Machine. In *Selected Reprints on Dataflow and Reduction Architectures*, S. S. Trakkan (ed.), IEEE Computer Society Press (1987).

59. J. Peyton, C. Clark, J. Salkild, and M. Hardie, GRID—A High-Performance Architecture for Parallel Graph Reduction, *Processing of 1987 Functional Programming Languages and Computer Architecture Conference*, Springer-Verlag LNCS 274, pp. 98–112 (1987).
60. P. Sesyoft, Deriving a Lazy Abstract Machine, *J. Functioning Programming* **1**(1) (1993).
61. R. Surati and A. Berlin, *Exploiting the Parallelism Exposed by Partial Evaluation*, MIT A.I. Memo No. 1414a (May 1994).