Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences

Cheng-Zhong Xu, *Member*, *IEEE Computer Society*, and Vipin Chaudhary, *Member*, *IEEE*

Abstract—This paper presents a time stamp algorithm for runtime parallelization of general DOACROSS loops that have indirect access patterns. The algorithm follows the INSPECTOR/EXECUTOR scheme and exploits parallelism at a fine-grained memory reference level. It features a parallel inspector and improves upon previous algorithms of the same generality by exploiting parallelism among consecutive reads of the same memory element. Two variants of the algorithm are considered: One allows partially concurrent reads (PCR) and the other allows fully concurrent reads (FCR). Analyses of their time complexities derive a necessary condition with respect to the iteration workload for runtime parallelization. Experimental results for a Gaussian elimination loop, as well as an extensive set of synthetic loops on a 12-way SMP server, show that the time stamp algorithms outperform iteration-level parallelization techniques in most test cases and gain speedups over sequential execution for loops that have heavy iteration workloads. The PCR algorithm performs best because it makes a better trade-off between maximizing the parallelism and minimizing the analysis overhead. For loops with light or unknown iteration loads, an alternative speculative runtime parallelization technique is preferred.

Index Terms—Compiler, parallelizing compiler, runtime support, inspector-executor, doacross loop, dynamic dependence.

---

1 INTRODUCTION

AUTOMATIC parallelization is a key enabling technique for parallel computing. Of particular importance is loop parallelization. Loops are often classified as one of two types: static or dynamic, with respect to the determinism of array subscript expressions. A static loop, as shown in Fig. 1a, is characteristic of statically known coefficients and deterministic index functions in all subscript expressions. Its loop-carried dependence can be analyzed at compile time. By contrast, the subscript expressions in dynamic loops, like Fig. 1b, are nondeterministic due to the presence of input-dependent indirect access arrays u and v. Since cross-iteration dependences in dynamic loops are unknown at compile time, parallelization of such loops has to be complemented by runtime techniques.

Previous studies on parallelizing compilers were mostly targeted at static loops. This paper focuses on runtime parallelization of dynamic loops. Dynamic loops appear frequently in scientific and engineering applications. For example, in molecular dynamics [18], a calculation of nonbounded forces between atoms is a dynamic loop, like Fig. 2a, because atoms within a certain cutoff are accessed via the indirect array *partners*. Another example of dynamic loops occurs in computational fluid dynamics (CFD). A CFD simulation on unstructured grids is essentially a PDE solver on sparse matrices [21]. Under an edge-oriented compressed data representation of sparse matrices, the solver is a loop with indirect access to grid points, as shown in Fig. 2b. Although the statically unknown cross-iteration dependences of the two example loops can be broken by techniques like parallel induction, runtime parallelization techniques are highly demanded for general dynamic loops. An empirical study by Shen et al. [33] on array subscript expressions of more than 1,000 scientific and engineering routines (100,000 code lines) showed that nearly half of the array references were nonlinear subscript functions and that about 15 percent of the nonlinear one-dimensional array references were due to the presence of indirect access arrays. Dynamic loops are also popular in symbolic applications due to the commonplace of pointer data structures.

Like compile-time analyses for static loops, runtime parallelization of a dynamic loop is for detecting cross-iteration dependences and exploiting appropriate degrees of parallelism for concurrent execution of the loop. Since the process of dependence detection and parallelism exploitation incurs nonnegligible runtime overhead, it is the additional objective of minimizing the overhead that makes runtime parallelization harder than compile-time analyses.

There are two major runtime parallelization approaches: Inspector [9], [22], [30], [32] and *speculative execution* [14], [27], [30], [39]. With the Inspector scheme, a loop under consideration is transformed at compile time into a pair of inspector and executor routines. At runtime, the inspector examines loop-carried dependences and the executor performs the actual loop operations in parallel, based on the dependence information exploited by the inspector. In the speculative execution scheme, the target loop is first

• *The authors are with the Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI 48202.*
E-mail: {czxu, vchaud}@ece.eng.wayne.edu.

Manuscript received 25 Aug. 1998; revised 7 Dec. 1999; accepted 15 Dec. 2000.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 107309.

1045-9219/01/$10.00 © 2001 IEEE

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

```
for (i=0; i<N-1; i=i+1)
   X[i] = F (X[i+1], ...)
endfor
```

(a)

```
for (i=0; i<N; i=i+1)
   X[u[i]] = F(X[v[i]], ... )
endfor
```

(b)

Fig. 1. Generic loops exhibiting static and dynamic dependencies. (a) Static loop. (b) Dynamic loop.

handled as a DOALL loop regardless of its inherent degree of parallelism. If a subsequent dependence test at runtime finds that the loop is not fully parallel, the whole computation is then rolled backed and executed sequentially. Speculative execution yields good results when the loop is in fact executable concurrently. However, it is unable to deal with DOACROSS loops that contain cross-iteration dependences. Due to the popularity of DOACROSS loops, this paper follows the Inspector/Executor scheme with the objective of designing efficient Inspector algorithms for a good trad-eoff between the degree of parallelism and the runtime parallelization overhead.

There were studies of the Inspector scheme that focused on parallel dependence analyses [9], [22], [30], [32], synchronization optimization on shared memory multiprocessors [7], and communication scheduling on distributed memory systems [32]. There were also studies on parallelization techniques for special loops without certain types of cross-iteration dependences [23], [32]. In this paper, we propose time stamp Inspector algorithms for general dynamic loops that contain any type of cross-iteration dependence. It detects cross-iteration dependences in parallel and exploits parallelism at a fine-grained memory reference level. A challenge with parallel Inspector algorithms is the treatment of consecutive reads of a memory location in different iterations. Consider a loop derived from the solution of equation $UX = B$, where U is an upper triangle matrix. Its iteration space is shown in Fig. 3. While the loop is static, Gaussian elimination on sparse matrices often leads to dynamic loops. For a parallel inspector, parallelism detection between consecutive reads of the same element (e.g., X[N-1]) is nontrivial by any means because the reads, subject to the same flow and/or antidependences, may be handled by different processors. As far as we know, there are no parallel approaches that are capable of exploiting parallelism between the consecutive reads. The time stamp algorithm improves upon previous parallel approaches of the same generality by exploiting various degrees of parallelism and, particularly, allowing consecutive reads in different iterations to proceed concurrently. Two variants of the algorithm are evaluated: One allows partially concurrent reads (PCR) and the other allows fully concurrent reads (FCR).

The time stamp algorithms exploit parallelism at a fine-grained memory reference level. An alternative is the iteration-level technique. It assumes a loop iteration as the basic scheduling and execution unit. It decomposes the iteration space into a sequence of subsets, called wavefronts. Iterations within the same wavefront can be run in parallel. Dependences between wavefronts are enforced by barrier synchronization. The iteration-level technique reduces the runtime scheduling overhead while sacrificing some parallelism. Loop parallelization at the iteration level is a common practice in compile-time analyses. Static loops that have nonuniform cross-iteration dependences are often transformed, based on convex hull theories, into a sequence of DOALL loops [19], [35]. For dynamic loops, wavefront transform techniques may not lead to good performance.

We evaluated the runtime techniques for the parallelization of a Gaussian elimination loop, as well as a set of synthetic loops, on a SUN Enterprise Server E4000 with 12 processors. The experimental results show that the time stamp algorithms work well for loops that contain heavy workload at each iteration. However, for loops with light iteration workloads, speculative runtime parallelization techniques are preferred. Of the time stamp algorithms, the PCR algorithm yields substantial improvement over the others because it makes a better trade-off between maximizing the parallelism and minimizing the analysis overhead. We also showed that the time stamp algorithms outperform the iteration-level I approach in most test cases.

The rest of this paper is organized as follows: Section 2 provides an overview of the technique and a brief review of related work. Sections 3 and 4 present the time stamp algorithms that exploit varying amounts of parallelism for consecutive reads of the same memory element. Section 5 describes implementation details of the algorithms and detailed analyses of their complexities in space and time. Section 6 presents the evaluation results. Section 7 concludes

```
ATOM X[NumAtoms], Y[NumAtoms];
ATOM *partners[NumAtoms];

for (i=0; i<NumAtoms; i=i+1)
      while (j is_in partners[i] )
            Y[i] += force(X[i], X[j]);
            Y[j] += force(X[i], X[j]);
endfor
```

(a)

```
NODE X[NumNodes], Y[NumNodes];
struct {
      NODE LeftNode, RightNode;
} edge[ NumEdges];

for (i=0; i<NumEdges; i=i+1)
      n1 = edge[i].LeftNode;
      n2 = edge[i].RightNode;
      Y[n1] += f(X[n1], X[n2]);
      Y[n2] += g(X[n1], X[n2]);
endfor
```

(b)

Fig. 2. Examples of dynamic loops from engineering applications. (a) A molecular dynamics loop. (b) A computational fluid dynamics loop.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

w X[N-1]	r X[N-1]	r X[N-1]	...	r X[N-1]
	w X[N-2]	r X[N-2]	...	r X[N-2]
		w X[N-3]	...	r X[N-3]
			...	
				r X[1]
				w X[0]

Fig. 3. The iteration space of the Gaussian elimination method to solve $UX = B$.

the presentation with remarks on limitations of the Inspector/Executor runtime parallelization algorithms.

2 Background and Related Work

Loop parallelization focuses on cross-iteration dependence analyses and exploitation of parallelism. Since a loop tends to be run over data arrays, loop-carried dependences can be characterized by subscript expressions in array references. Each subscript expression is a function of loop index variables. For example, two iterations, (i_1, j_1) and (i_2, j_2), of a double loop, which contain references to the same array element, are dependent and their dependent distance is $(i_2 - i_1, j_2 - j_1)$. One of the principal objectives of loop parallelization is to discover distance vectors to characterize all cross-iteration dependences. Finding dependences has proven to be equivalent to the NP-Complete problem of finding integer solutions to systems of Diophantine equations. Past studies were mostly targeted at loops with simple linear subscript expressions; see [3] for a survey of the early methods for solving linear Diophantine equations and [2] for related techniques in a general context of optimizing compilers. For nonlinear subscript expressions, the compiler employs approximate methods to test data dependences [4] or simply assumes that the loop statements are fully dependent on one another.

Note that a subscript expression is not necessarily deterministic at compile time. Its coefficients, or the function, may be input-dependent. When the subscript expressions are nondeterministic, cross-iteration dependent information has to be exploited at runtime. Inspector/Executor is an important runtime parallelization technique. Its basic idea is to have the compiler generate an inspector and an executor for a loop to be parallelized at runtime. The inspector identifies cross-iteration dependences and produces a parallel execution schedule. The executor uses this schedule to perform the actual operations of the loop. The Inspector/Executor scheme provides a runtime parallelization framework and leaves strategies for dependence analysis and scheduling unspecified. The scheme can also be restructured to decouple the scheduling function from the inspector and merge it with the executor. The scheduling function can even be extracted to serve as a stand-alone routine between the inspector and the executor. There are many runtime parallelization algorithms belonging to the Inspector/Executor scheme. They differ from each other mainly in their structures and strategies used in each routine, in addition to the type of target loops considered.

Pioneering work on using the Inspector/Executor scheme for runtime parallelization is due to Saltz et al. [32]. For loops without output dependences (i.e., the indexing function used in the assignments of the loop body is an identity function), they proposed an effective iteration-level Inspector/Executor scheme. Its inspector partitions the set of iterations into a number of subsets, called wavefronts, that maintain cross-iteration flow dependences. Iterations within the same wavefront can be executed concurrently, but those in different wavefronts must be processed in order. The executor of the scheme enforces antidependences during the execution of iterations in the same wavefront. Saltz et al. applied the technique to loops like Fig. 2b [18]. The basic scheme was generalized by Leung and Zahorjan to general loops that contain any cross-iteration dependences [23]. In their algorithm, the inspector generates a wavefront-based schedule and maintains output dependences and antidependences as well as flow dependences. The executor simply performs the loop operations according to the wavefronts of iterations. Fig. 4 shows the inspector and executor transformed from the loop in Fig. 1b. In the figure, $wf[i]$ stores the wavefront level of iteration i. $lr[i]$ and $lw[i]$ record the last wavefront that reads and writes $X[i]$, respectively. wf, lr, and lw are initially set to zero.

Note that the inspector in the above scheme is sequential. It requires time commensurate with that of a serial loop execution. Parallelization of the inspector loop was investigated by Saltz et al. [32] and Leung and Zahorjan [22]. Their techniques are applicable to loops without anti or output dependences. Rauchwerger et al. presented a parallel inspector algorithm for a general form of loops [30]. They extracted the scheduling function and presented an inspector/scheduler/executor scheme. Both their inspector and scheduler can be run in parallel.

Iteration-level Inspector/Executor schemes assume a loop iteration as the basic scheduling unit in the inspector and the basic synchronization object in the executor. By contrast, reference-level parallelization techniques exploit parallelism at a fine-grained memory reference level. They assume a memory reference as the basic unit of scheduling and synchronization. Processors running the executor are assigned iterations in a wrapped manner and each spin

```
for (i=0; i<N; i++)
    w = max(lr[u[i]], lw[u[i]], lw[v[i]], ...) + 1;
    lw[u[i]] = wf[i] = w;
    lr[v[i]] = max(lr[v[i]], w);
endfor
```

(a)

```
for (w = 1; w <= NumWavefront; w++)
    forall i such that wf[i] = w
        X[u[i]] = F(X[v[i]], ...)
    endfor
endfor
```

(b)

Fig. 4. An iteration level Inspector/Executor shceme. (a) Inspector. (b) Executor.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

waits as needed for operations that are necessary for its execution. Zhu and Yew [40] proposed a scheme that integrates the functions of dependence analysis and scheduling into a single executor. Midkiff and Padua [25] improved upon the algorithm by allowing concurrent reads of the same array element by several iterations. Even though the integrated scheme allows concurrent analysis of cross-iteration dependences, tight coupling of the dependence analysis and the executor incurs high synchronization overhead in the executor. Recently, Chen et al. [9] advanced the technique by decoupling the function of the dependence analysis from the executor. Separation of the inspector and executor not only reduces synchronization overhead in the executor, but also provides the possibility of reusing the dependence information developed in the inspector across multiple invocations of the same loop. Their inspector can proceed in parallel. But, it is unable to exploit parallelism between consecutive reads of the same array element.

The iteration and reference levels are two competing techniques for runtime loop parallelization. Reference-level techniques overlap dependent iterations. They can also enforce nondeterministic control dependences. Consider the loop and index arrays shown in Fig. 5. The first two iterations can be either independent (when $exp(0)$ is false and $exp(1)$ is true), flow dependent (when $exp(0)$ is true and $exp(1)$ is false), antidependent (when both $exp(0)$ and $exp(1)$ are true), or output dependent (when both $exp(0)$ and $exp(1)$ are false). The nondeterministic cross-iteration dependences are due to control dependences between statements in the loop body. Control dependences can be converted into data dependences by an *if-conversion* technique at compile time [1]. The compile-time technique, however, may not be useful for loop-carried dependence analyses at runtime. In iteration-level techniques, loops with conditional cross-iteration dependences must be handled sequentially. In contrast, reference-level techniques can handle this class of loops easily. At runtime, the executor, upon testing a branch condition, may set all operands in the untaken branch available so as to release processors waiting for those operands.

An alternative to the Inspector/Executor approach is the speculative execution scheme. In this scheme, the target loop is first handled as a parallel loop regardless of its inherent degree of parallelism. If a subsequent dependence test at runtime finds that the loop is not fully parallelizable, the whole computation is then rolled back and executed sequentially. The idea of speculative execution is not new. It has long been an important technique for exploitation of branch-related instruction-level parallelism [28]. The technique was also deployed in the design of multithreading architectures [20], [27]. Recently, Rauchwerger et al. applied the idea, together with dynamic privatization and reduction recognization, for runtime loop parallelization [31], [39]. The technique has proven to be effective for DOALL loops and loops that are convertible to DOALL due to array privatization and reduction parallelization. The speculative parallelization relies on runtime dependence testing. A related technique is the linearization test [13], [15]. It checks whether the

```
for (i=0; i<N; i++)                          u1 = [5, 7, ...]
    if ( exp(i) )                            u2 = [3, 3, ...]
        X[u1[i]] = F(X[v1[i]], ...)          v1 = [7, 2, ...]
    else                                     v2 = [4, 5, ...]
        X[u2[i]] = F(X[v2[i]], ...)
endfor
```

Fig. 5. An example of loops with conditional cross-iteration dependences.

array subscript expressions are linear functions of loop indices and decides if the loop carries cross-iteration dependences. Inspector/Executor parallelization techniques complement speculative execution of general dynamic DOACROSS loops. Advantages of parallelizing DOACROSS loops were shown in [6], [7], [12], [17], [24]. Readers are referred to [29] for a recent comprehensive survey of the Inspector/Executor and speculative runtime parallelization techniques.

Finally, we note that objectives of loop parallelization are to exploit parallelism based on cross-iteration dependences, partition the loop computation and data across processors, and orchestrate the parallel execution of the loop. Besides techniques for dependence analysis and parallelism exploitation, there are strategies that are dedicated to scheduling DOACROSS loops for data locality, load balancing, and minimizing synchronization overhead [8], [16], [38]. Scheduling and synchronization optimization techniques complement runtime loop parallelization algorithms.

3 Time Stamp Algorithm Allowing Partially Concurrent Reads

This section presents a parallel reference-level INSPECTOR/EXECUTOR algorithm (PCR, for short) along the lines of the work by Chen et al. [9]. We refer to the original algorithm as CTY. The PCR algorithm allows partially concurrent reads without incurring extra overhead in the inspector. The next section presents a new algorithm (FCR, for short) that allows fully concurrent reads.

Consider the general form of the loops in Fig. 1b. It defines a two-dimensional iteration-reference space. The inspector of the algorithm examines the array element references within a loop and constructs a dependence chain for each element in the iteration-reference space. We index dependence chains by the array element references. That is, dependence chain k represents a chain of references that accesses the kth array element. Each reference in a dependence chain is assigned a stamp indicating its earliest activation time relative to the other references in the chain. A reference can be activated if and only if the preceding references are finished. The executor schedules activation of the references of a chain through a logical clock. At a given time, only those references whose stamps are equal to or less than the clock are allowed to proceed. Dependence chains are associated with clocks ticking at different speeds.

Note that we assume the loop has the same number of array element references (i.e., loop depth) at each iteration. We use a vector, rw, to indicate each reference as a read or a write. For example, the loop in Fig. 1b has an rw vector of $[\mathrm{READ}, \mathrm{WRITE}]$. The proposed algorithms are readily

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

applicable to loops that contain control dependences. For example, in Fig. 5, we can set the loop depth to four and set $rw = [\text{READ}, \text{WRITE}, \text{READ}, \text{WRITE}]$. A corresponding indirect access array can be constructed at runtime based on the values of u_1, u_2, v_1, v_2, and $exp(i)$. The proposed algorithms can also be applied to loops that have different numbers of references in iterations. For a loop with iteration space, as in Fig. 3, we normalize the loop by inserting "void" references into its indirect access array so that $rw = [\text{READ}, \text{READ}, \ldots, \text{READ}, \text{WRITE}]$.

3.1 Serial Inspector and Parallel Executor

We assume reference stamps are discrete integers. The stamps are stored in a two-dimensional array $stamp$. Let (i, j) indicate the jth reference within the ith iteration. $stamp[i][j]$ represents the stamp of reference (i, j). $rw[j]$ indicates that the jth reference within an iteration is a read or a write. Stamping rules of the inspector algorithm are defined as follows:

S0. References at the beginning of dependence chains are assigned to one.

S1. For a reference (i, j) to array element k, assuming its immediate predecessor in the dependence chain k is reference (m, n),

S1.1.
if $rw[j] == \text{WRITE}$, then $stamp[i][j] = s$, where s is the reference order in the chain;
S1.2.
if $rw[j] == \text{READ}$, then

$$stamp[i][j] = \begin{cases} stamp[m][n] & \text{if } rw[n] == \text{READ}, \\ stamp[m][n] + 1 & \text{if } rw[n] == \text{WRITE}. \end{cases}$$

Assume the indirect arrays in the loop of Fig. 1b are

$$u = [15, 5, 5, 14, 10, 14, 12, 11, 3, 12, 4, 8, 3, 10, 10, 3]$$
$$v = [3, 13, 10, 15, 0, 8, 10, 10, 1, 10, 10, 15, 3, 15, 11, 0].$$

Applying the above stamping rules to the target loop, we obtain the stamped dependence chains labeled by array elements, as shown in Fig. 6. For example, the references (2, 0), (4, 1), (6, 0), (7, 0), (9, 0), (10, 0), (13, 1), and (14, 1) form a dependence chain 10 because they are all associated with array element 10. Reference (2, 0) heads the chain and, hence, set $stamp[2][0] = 1$.

From Fig. 6, it can be seen that the stamp difference between any two consecutive references in a chain is one, except for pairs of write-after-read and read-after-read references. In pairs of read-after-read, both reads have an equivalent stamp. In pairs of write-after-read, their difference is always the number of consecutive reads.

In the executor, we define a logical clock for each dependence chain. Let $time[k]$ represent the current clock time of chain k. We set up the following clocking rules in the executor corresponding to the inspector's stamping rules:

C1. Initially, for each dependence chain k, set $time[k] = 1$.

C2. A reference (i, j) in dependence chain k is activated if $stamp[i][j] \le time[k]$.

C3. Upon completion of the reference, $time[k] = time[k] + 1$.

Access \ Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Read	3 (1)	13 (1)	**10** (1)	15 (2)	0 (1)	8 (1)	**10** (3)	**10** (3)	1 (1)	**10** (3)	**10** (3)	15 (2)	3 (3)	15 (2)	11 (2)	0 (1)	v
Write	15 (1)	5 (1)	5 (2)	14 (1)	**10** (2)	14 (2)	12 (1)	11 (1)	3 (2)	12 (2)	4 (1)	8 (2)	3 (4)	**10** (7)	**10** (8)	3 (5)	u

Fig. 6. Sequentially constructed dependence chains labeled by an array element. The numbers in parentheses are the stamps of the references.

It is easy to show that the above Inspector/Executor algorithm enforces all dependences and allows concurrent reads. Look at the dependence chain associated with array element 10. From the stamps of its references, it is clear that the four reads (6, 0), (7, 0), (9, 0), and (10, 0) are able to proceed simultaneously once the write (4, 1) is done. The write operation (13, 1) cannot be activated until all four reads are finished.

3.2 Parallel Inspector and Parallel Executor

The above algorithm builds the stamp table sequentially. Building dependence chains that reflect all types of dependences is a time-consuming process. It requires an examination of all references at least once in the loop. A basic parallel strategy is to partition the entire iteration space into a number of blocks. Each block, comprised of a number of consecutive iterations, is assigned to a different processor. Each processor establishes its local dependence chains by examining the references in its local block. Processors then exchange information about their local dependence chains and connect them into complete chains.

To apply the algorithm to the construction of the stamp table in parallel, one key issue is to stamp the references in a dependence chain across blocks so as to enforce all dependences and simultaneously allow independent references to be performed in parallel. Since no processors (except the first) have knowledge about the references in preceding blocks, they are unable to stamp their local references in a dependence chain without the assignment of its head. Suppose there are four processors that cooperatively build a stamp table, like Fig. 6. Each processor examines four consecutive iterations. We label processors participating in the parallelization by their block indices. For example, consider the references in block 3 to array element 3. Since processor 3 does not know whether there are dependent references in blocks 0 through 2 and what their stamps are, it is unable to stamp local references (12, 0), (12, 1), and (15, 1).

To allow processors to continue with the examination of other references in their local blocks in parallel, Chen et al. [9] proposed a *conservative* approach for an inspector to assign a conservative number to the second reference of a local chain and leave the first to be decided in a subsequent global analysis. Using this conservative approach, processor 3 temporarily assigns 24 plus 1 to the reference (12, 0), assuming all 24 accesses in preceding blocks (from 0 to 2) are in the same dependence chain. This results in a stamp of 26 in the subsequent reference (12, 1), as shown in Fig. 7. The negative sign of a stamp indicates the stamp is temporarily recorded for the calculation of subsequent references' stamps. Consider a reference (i, j)

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

Access \ Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Read	3 (1)		10 (1)	15 (2)			10 (10)	10 (10)		10 (-17)	10 (-17)	15 (-17)	3 (-25)	15 (-25)			v
Write	15 (1)				10 (-9)				3 (-17)				3 (26)	10 (-25)	10 (26)	3 (27)	u

Fig. 7. A partially stamped dependence chain labeled by array elements. Numbers in parentheses are stamps of references.

Access \ Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Read	3 (1)		10 (1)	15 (2)			10 (10)	10 (10)		10 (12)	10 (12)	15 (3)	3 (18)	15 (18)			v
Write	15 (1)				10 (2)				3 (2)				3 (26)	10 (19)	10 (26)	3 (27)	u

Fig. 8. A fully stamped dependence chain labeled by array elements. Numbers in parentheses are stamps of refereneces.

in block r. Let $g(r)$ denote the total number of references in blocks 0 through $r-1$ and $h(r)$ denote the number of references of the same dependence chain in block r. For example, in Fig. 7, $g(0)=0$, $g(1)=8$, $g(2)=16$, and $g(3)=24$. For the dependence chain associated with 10 in block 3, $h(3)=2$.

Borrowing the idea of the conservative approach, we set up one more stamping rule that assigns the same stamp to all references in a local *read group*. A read group comprises consecutive reads of the same array element. If all reads in the group are in block r, the group is local to block r.

S2. For a reference (i,j) in block r, $r>0$,

S2.1.
if the reference heads a local dependence chain, $stamp[i][j] = -(g(r)+1)$;
S2.2.
if the head reference is a read, for a subsequent reference (m,n) in the same read group, $stamp[m][n] = stamp[i][j]$.

Applying the above stamping rule, together with the rule for the serial inspector on local dependence chains, we obtain partially stamped dependence chains as presented in Fig. 7. There are three partially stamped dependence chains associated with array elements 3, 10, and 15 in Fig. 7. The dependence chains of other elements are omitted for clarity.

Using the conservative approach, most of the stamp table can be constructed in parallel. Upon completion of the local analysis, processors communicate with each other to determine the stamps of undecided references in the stamp table. Processor 2 sets $stamp[8][1]$ to 2 after communicating with processor 0 (processor 1 marks no reference to the same location). At the same time, processor 3 communicates with processor 2, but gets an undecided stamp on the reference (8, 1) and, hence, assigns another conservative number, 17 plus 1, to reference (12, 0), assuming all accesses in blocks 0 and 1 are in the same dependence chain. The extra one is due to the total number of dependent references in block 2. Note that the communications from processor 3 to processor 2 and from processor 2 to processor 1 are run in parallel. Processor 2 cannot provide processor 3 with any information, except the number of references in the local block, until the end of the communication with processor 0.

Generally, processors communicate in the global analysis phase to determine their undecided references using the following rules: For an undecided reference (i,j) in block r, assuming its immediate predecessor (m,n), if it exists, is in block r'.

G1. If $rw[j] == \mathrm{WRITE}$,

$$stamp[i][j] = \begin{cases} g(r') + h(r') + 1 & \text{if } stamp[m][n] < 0, \\ stamp[m][n] + 1 & \text{otherwise.} \end{cases}$$

G2. If $rw[j] == \mathrm{READ}$, then

G2.1.
if $rw[n] == \mathrm{WRITE}$, then

$$stamp[i][j] = \begin{cases} g(r') + h(r') + 1 & \text{if } stamp[m][n] < 0, \\ stamp[m][n] + 1 & \text{otherwise.} \end{cases}$$

G2.2.
if $rw[n] == \mathrm{READ}$, then

$$stamp[i][j] = \begin{cases} stamp[m][n] & \text{if } r = r', \\ g(r') + h(r') + 1 & \text{if } r \neq r' \text{ and } stamp[m][n] < 0, \\ stamp[m][n] + s & \text{if } r \neq r' \text{ and } stamp[m][n] > 0, \end{cases}$$

where s is the group size of reference (m,n).

Fig. 8 shows the complete dependence chains associated with array elements 3, 10, and 15. Because $g(1)=8$ and $h(1)=3$, $stamp[9][0]$ is assigned to 12 by rule G2.2 (the second case).

Accordingly, the third clocking rule of the executor in Section 3.1 is modified as follows:

C3. Upon the completion of the reference (i,j) of a dependence chain k in block r,

$$time[k] = \begin{cases} time[k] + 1, & \text{if } stamp[i][j] \geq g(r), \\ g(r) + 2, & \text{otherwise.} \end{cases}$$

For example, consider references in the dependence chain associated with element 10 in Fig. 8. Execution of the reference (2, 0) increases $time[10]$ by one because $g(0)=0$ and, hence, triggers the reference (4, 1). Activation of reference (4, 1) will set $time[10]$ to 10 because $g(1)=8$. There are two concurrent reads (6, 0) and (7, 0) in block 1 and two concurrent reads (9, 0) and (10, 0) in block 2. One of the reads in block 2 will set $time[10]$ to 18 because $g(2)=16$ and the other increments $time[10]$ to 19. The write (13, 1) is then triggered. The last is the write (14, 1).

Theorem 3.1. *The PCR algorithm, specified by local inspection rules S1-S2, global inspection rules G1-G2, and clocking rules C1-C3, exploits parallelism among consecutive reads in the same block.*

Proof. A group of reads which are consecutive in a local dependence chain are assigned the same stamp according to inspection rules S1.2, S2.2, and G2. They will be activated simultaneously by their immediate predecessor write following the execution rules from C1 to C3. Since their immediate successor write is stamped by counting the total number of references, including reads, as in rule

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

G1, the write will not be activated until all predecessor reads are complete. Conclusively, the PCR algorithm does allow consecutive reads in a block to be activated simultaneously. □

Note that this parallel inspector algorithm only allows consecutive reads in the same block to be performed in parallel. Read references in different blocks must be performed sequentially even though they are consecutive in the final dependence chains. For example, in the dependence chain associated with element 10 in Fig. 8, the reads (9, 0) and (10, 0) are activated after reads (6, 0) and (7, 0). We are able to assign reads (9, 0) and (10, 0) the same stamp as reads (6,0) and (7,0), and assign the reference (13, 1) a stamp of 14. However, this dependence chain will destroy the antidependences from (6, 0) and (7, 0) to (14, 0) in the executor if reference (9, 0) or (10, 0) starts earlier than one of the reads in block 1.

4 Time Stamp Algorithm Allowing Fully Concurrent Reads

This section presents the FCR algorithm that allows fully concurrent reads. The basic idea is to use rational numbers to represent clock time in the synchronization of references in a dependence chain. Write operations and read groups can each be regarded as a macroreference. For a write reference or the first read in a read group in a dependence chain, the inspector stamps the reference with the total number of macroreferences ahead. Other references in a read group are assigned the same stamp as the first read in the group. Correspondingly, in the executor, the clock of a dependence chain is incremented by one time unit on a write reference and by a fraction of a time unit on a read operation. The magnitude of an increment on a read operation is the reciprocal of its read group size.

Fig. 9 presents sequentially stamped dependence chains. In addition to the stamp, each read reference is also associated with an extra integer recording its read group size. Since neither a reference stamp nor read group size would be larger than $N \times D$, where N is the number of iterations and D is the number of references in an iteration, the variable for read group size can be combined with the variable for stamp in implementations. For simplicity of presentation, they are declared as two separate integers. Look at the dependence chain associated with element 10. The reference (4, 1) simultaneously triggers four subsequent reads: (6, 0), (7, 0), (9, 0), and (10, 0). Activation of each of these reads increments the clock time by 1/4. After all of them are finished, the clock time reaches 4, which in turn activates the reference (13, 1). Following are the details of the algorithm.

4.1 Parallel Inspector

As in the PCR algorithm, the inspector first partitions the iteration space of a target loop into a number of blocks. Each block is assigned to a different processor. Each processor first stamps local references except the head macroreferences at the beginning of the local dependence chains. References next to head macroreferences are stamped with conservative numbers using the conservative

Access \ Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Read	**3**	13	**10**	15	0	8	**10**	**10**	1	**10**	**10**	15	**3**	15	11	0	v
	(1,1)	(1,1)	(1,1)	(2,3)	(1,2)	(1,1)	(3,4)	(3,4)	(1,1)	(3,4)	(3,4)	(2,3)	(3,1)	(2,3)	(2,1)	(1,2)	
Write	15	5	5	14	**10**	14	12	11	**3**	12	4	8	**3**	**10**	**10**	3	u
	(1)	(1)	(2)	(1)	(2)	(2)	(1)	(1)	(2)	(2)	(1)	(2)	(4)	(4)	(5)	(5)	

Fig. 9. Sequentially constructed dependence chains labeled by array elements. Numbers in parentheses are stamps of references.

approach as in the PCR algorithm. Processors then communicate with each other to merge consecutive read groups and stamp undecided macroreferences in the time table.

The base of the algorithm is a three dimensional stamp table: $stamp$. $stamp[i][j][0]$ records the stamp of reference (i, j) in the iteration-reference space. $stamp[i][j][1]$ stores the size of a read group to which reference (i, j) belongs. If reference (i, j) is a write, $stamp[i][j][1]$ is not used. As mentioned above, the variables for the read group size and the stamp can be combined to reduce the three-dimensional table to a two-dimensional array. The local inspector at each processor passes once through its iteration-reference block, forms read groups and assigns appropriate stamps onto its *inner* references. Inner references refer to those whose stamps can be determined locally using conservative approaches. All write operations, except for the heads of local dependence chains, are inner references.

Read chains are formed as a doubly linked list in the stamp table. For a read reference (i, j), we use $stamp[i][j][0]$ and $stamp[i][j][1]$ to store addresses of its predecessor and successor, respectively. To save memory space for the stamp table, we represent each address (i, j) in a single integer $i * D + j$, where D is the number of references in an iteration. To distinguish reference pointers from stamps in the table, we use negative numbers to represent read group links. A read group is closed if it is bracketed by two literals, L-CLOSURE and R-CLOSURE, at two ends. Whenever a read reference (i, j) is unable to determine the address of its predecessor or successor, it marks the address "u" (undecided) using the minimal integer (MININT) at $stamp[i][j][0]$ or $stamp[i][j][1]$. A read group is open if one or two ends are labeled with address "u."

Following are the rules to stamp a reference (i, j) of a dependence chain in block r. Assume reference (m, n) is its immediate predecessor, if one exists.

S1. If reference (i, j) leads a local dependence chain, then

$$stamp[i][j][0] = \begin{cases} 1 & \text{if } r = 0, \\ u & \text{otherwise} \end{cases}$$

S2. If $rw[j] == \text{READ}$ and $rw[n] == \text{WRITE}$, then

$$stamp[i][j][0] = \text{L-CLOSURE};$$

S3. If $rw[j] == \text{READ}$ and $rw[n] == \text{READ}$, then

$$stamp[m][n][1] = -(i * D + j)$$

and

$$stamp[i][j][0] = -(m * D + n),$$

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

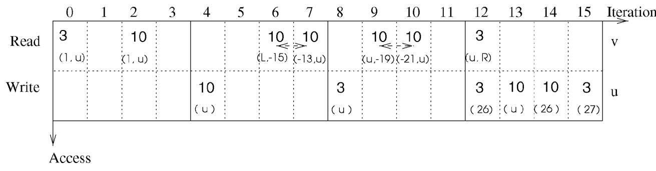

Fig. 10. A partially stamped dependence chain constructed in parallel.

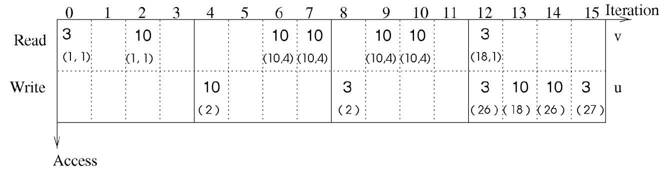

Fig. 11. A fully stamped dependence chain constructed in parallel.

connecting these two reads together;

S4. If $rw[j] == \text{WRITE}$ and $rw[n] == \text{READ}$, then

$$stamp[i][j][0] = g(r) + 2$$

and

$$stamp[m][n][1] = \text{R-CLOSURE};$$

If the read group is closed at both ends, then, for each reference (p, q) in the group, $stamp[p][q][0] = x + 1$ and $stamp[p][q][1] = y$, where x is the stamp of the predecessor write and y is the read group size; and

S5. If $rw[j] == \text{WRITE}$ and $rw[n] == \text{WRITE}$, then

$$stamp[i][j][0] = \begin{cases} g(r) + 2 & \text{if } stamp[m][n][0] = u, \\ stamp[m][n][0] + 1 & \text{otherwise.} \end{cases}$$

Fig. 10 presents a partially stamped dependence chain associated with elements 10 and 3. From Fig. 10, it can be seen that block 0 contains a single read group (2, 0), block 1 contains a right-open group (6, 0)(7, 0), and block 2 has a group (9, 0)(10, 0) that is open at both ends.

The local inspector stamps all inner references of each block. A subsequent global inspector merges consecutive read groups that are open to each other. Two consecutive read groups that open to each other at one end and close at the other end are merged into a closed read group. Two consecutive open read groups are merged into a larger open read group. Fig. 11 is a fully stamped chain evolved from Fig. 10. Since the read group (6, 0)(7, 0) is open to the group (9, 0)(10, 0), they are merged into a right-open bigger group. The global inspector will close the group due to the presence of write (13, 1).

The global inspector treats read groups as macroreads and assigns stamps to undecided references in parallel. Although they require communication with each other to stamp the undecided references, no reference stamps will be updated by more than one processor. Specifically, for an undecided reference (i, j) in block r,

G1. If $rw[j] == \text{WRITE}$ and its immediate predecessor (m, n) is in block r', then

$$stamp[i][j][0] = \begin{cases} stamp[m][n][0] + 1 & \text{if } rw[n] == \text{WRITE}, \\ g(r') + 2 & \text{otherwise,} \end{cases}$$

G2. If $rw[j] == \text{READ}$ and the read is the last of a read group, then, for each read (p, q) within the group, $stamp[p][q][1]$ is assigned to the read group size and

$$stamp[p][q][0] = \begin{cases} g(r') + h(r') + 1 & \text{if } stamp[m][n][0] = u, \\ stamp[m][n][0] + 1 & \text{otherwise,} \end{cases}$$

where r' is the block index of the first read in the group and (m, n) is the immediate predecessor of the first read.

Note that a right closed read group implies the left end is closed as well because the processor examines reads in a group from left to right. When a processor encounters a right-open read group, it needs to check information from the following blocks to determine whether this read is the last in the current read group. For example, in Fig. 11, write (13, 1) is assigned 18, by rule G1 (case two), because $g(2) = 16$. Reads in the group {(6, 0)(7, 0)(9, 0)(10, 0)} are all assigned 10, by rule G2, because the first read of the group (6, 0) is located in block 1 and $g(1) = 8$ and $h(1) = 1$.

4.2 The Executor

The executor uses a logical clock to synchronize the memory access operations in a dependence chain. The clock time can be represented by rational numbers. As in the PCR algorithm, we index dependence chains by the indices of data elements. Let $time[k]$ represent the current clock time of chain k. We set up the following clocking rules corresponding to the inspector's stamping algorithm:

C1. Initially, $time[k] = 1$ for each clock k.

C2. Upon the completion of reference (i, j) in block r,

C2.1.
If $rw[j] == \text{WRITE}$, then

$$time[k] = \begin{cases} time[k] + 1 & \text{if } stamp[i][j][0] \geq g(r), \\ g(r) + 2 & \text{otherwise.} \end{cases}$$

C2.2.
If $rw[j] == \text{READ}$, then

$$time[k] = \begin{cases} time[k] + 1.0/stamp[i][j][1] & \text{if } stamp[i][j][0] \geq g(r), \\ g(r) + 1 + 1.0/stamp[i][j][1] + f & \text{otherwise,} \end{cases}$$

where f is the fractional part of $time[k]$.

Clock $time[k]$ ticks at different rates upon the completion of different types of references. In the case of a read reference, the clock is adjusted for the purpose of activating subsequent writes after all read operations are performed while preserving all antidepences dependences. For atomicity of the adjustment of clocks, mutually exclusive access to a clock is required. In our implementation, we use a lock to ensure mutually exclusive references. In order to avoid busy waiting, we use a condition variable to link all threads that are waiting for execution. Waiting threads are woken when the clock is incremented. If the current time reaches

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

the earliest time of a reference, the reference enters the ready state for execution. Otherwise, it goes back to the waiting state.

For example, look at the dependence chain associated with array element 10 in Fig. 11. Execution of reference (2, 0) increments $time[10]$ by one, by rule C2.2 (case one), because $g(1) = 0$ and $stamp[2][0][1] = 1$ (single read group) and, hence, triggers the execution of reference (4, 0). Upon the completion of reference (4, 0), $time[10]$ is set to 10 by rule C2.1 because $g(2) = 8$ and $stamp[4][1][0] = 2$. The subsequent four reads are then triggered. Without loss of generality, suppose the four references are performed in the following order: (6, 0), (9, 0), (7, 0) and (10, 0). Activation of reference (6, 0) increments $time[10]$ by 1/4 by rule C2.2 (case one) because $g(1) = 8$ and $stamp[6][0][0] = 10$. Execution of reference (9, 0) sets $time[10]$ to 17.5 by rule C2.2 because $g(2) = 16$. The subsequent two reads increase $time[10]$ to 18. Upon completion of all the reads, their subsequent write (13, 1) is activated. The purpose of the real number clock time is to record the number of activated reads in a group.

Theorem 4.1. *The FCR algorithm, specified by local inspection rules S1-S5, global inspection rules G1-G2, and clocking rules C1-C2, exploits parallelism among consecutive reads of the same memory location.*

Proof. A group of reads that are consecutive in a dependence chain are all assigned to the same stamp according to rule S3 and the merge operation in the global analysis. They will be activated simultaneously by their immediate predecessor write. Since their immediate successor write is stamped by treating the read group as a single read, the write will not be activated until all those reads finish, according to rule C2.2. Conclusively, the FCR algorithms do allow all consecutive reads to be activated at the same time. □

Finally, we note that using rational numbers to represent clock time is critical to the FCR algorithm. Consider a read group and its immediate successor write. The write may be located in a block different from the last read of the group or in the same block, as illustrated by the dependence chain of element 10 in Fig. 11. The first case can be handled by using integer clock time if the write is stamped with the size of its predecessor read group and each read increments the clock by one. However, this stamping and scheduling rule does not work for the second case. Since the write does not lead a local dependence chain, it should be assigned a conservative stamp in the local analysis so that its subsequent references can be stamped. The conservative stamping rule in Section 3.2 is in no way reflective of the size of a cross-block read group. Clocking by rational numbers, in contrast, allows a write to treat its preceding read group as a single macroreference regardless of their relative locations.

5 Space and Time Complexity

Runtime parallelization incurs nonnegligible runtime overhead. This section presents our reference implementations of the time stamp algorithms and detailed analyses of their complexities in both space and time.

Consider the dynamic loop in Fig. 1b. Assume the operator $\mathcal{F}$ in the loop body takes t_0 time. It follows that the loop sequential execution time $T_{SEQ} = Nt_0$, where N is the loop size. Parallel complexity of the time stamp algorithms is dependent on their implementation details. We assume the algorithms are programmed in a single-program multiple-data (SPMD) paradigm and as multithreaded codes. The threads proceed in three phases: *local inspector*, *global inspector*, and *executor*. The phases are separated by barrier synchronization. The barriers can be implemented in a blocking or busy-waiting scheme [11]. We assume blocking barriers so as to support a varying number of threads in the inspector and executor phases on a given set of processors.

For the dynamic loop in Fig. 1b, it is the compile-time analyzer that generates loop structural information as inputs to the local inspector. The loop structural information includes loop size (i.e., the number of iterations), loop depth (i.e., the number of array element references of an iteration), and a vector of D read/write indicators of references in each iteration, where D is the loop depth. During the local inspector phase, the threads jointly construct a global stamp table based on the loop structure information and an input-dependent indirect access array. Both the global stamp table and the indirect access array require $O(ND)$ space.

In the local inspector phase, each thread is assigned a block of nonoverlapping consecutive iterations to construct the corresponding block of the stamp table. Our implementation was based on two auxiliary arrays, *head* and *tail*:

```
struct {
    int index;      /* loc. of the 1st ref. in stamp table */
    int tag;        /* READ/WRITE indicator of the ref. */
} head [N]

struct {
    int current;        /* stamp of the most recent ref. */
    int index;          /* loc. of the ref. in stamp table */
    int tag;            /* READ/WRITE indicator */
    int groupsize;      /* read grp sz; used only in FCR */
} tail [N]
```

The *head* array records the first reference of each local dependence chain. The *tail* keeps the immediate predecessor of each reference) and, finally, the last reference of each local dependence chain. For example, in the construction of the stamp table of Fig. 10, thread 2 maintains a pair of head and tail arrays as follows:

$$\begin{aligned} head[3] &= (17, \text{WRITE}), \\ tail[3] &= (\text{UNDECIDED}, 17, \text{WRITE}, 0); \\ head[10] &= (18, \text{READ}), \\ tail[10] &= (\text{UNDECIDED}, 19, \text{READ}, 2). \end{aligned}$$

Since the arrays are to be maintained by each thread, the local inspector needs an extra $O(NP)$ auxiliary space for P threads. It leads to a space complexity of $O(ND + NP)$.

Threads in the local inspector work on nonoverlapped blocks of the stamp table and on their own auxiliary

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

head and *tail* arrays. They are able to run in parallel. In the local inspector, each thread parses the references in its local block once. Let t_1 denote the cost of parsing a reference. It takes a constant $O(1)$ time according to the stamping rules, as defined in the local inspector of the algorithms.

According to rule S4 of the FCR algorithm in Section 4.1, each locally closed read group needs an extra parse to stamp the references according to the group size. The extra time cost is determined by the distribution of the read groups and the group sizes. Let α denote the percentage of references that belong to locally closed read groups. Assume such groups are uniformly distributed across the loop iteration space. Then, the time complexity of the FCR local analysis is

$$T_{LA}^{FCR} = \frac{(1+\alpha)NDt_1}{P}. \quad (1)$$

In the global analysis phase, the threads determine the undecided stamps through communicating their read-only *head* and *tail* arrays. That is, thread i reads the *tail* data of thread $(i-1)$ and the *head* data of thread $(i+1)$. Although threads of the FCR global inspector may need to stamp references in nonlocal blocks, they can still be run in parallel without synchronization. This is because no table elements are to be updated by more than one thread. Time complexity of the global analysis includes two parts: 1) the cost of merging consecutive read groups which are open to each other and 2) the cost of stamping undecided references in the merged read groups. The cost of stamping an undecided reference is equivalent to $t_1 = O(1)$. Merging open read groups is based on the auxiliary arrays *head* and *tail*. Its cost is determined by the distribution of open read groups and the number of references in the groups. Let β denote the percentage of the references that belong to open read groups. Assume the open read groups are uniformly distributed across iteration spaces. We obtain the time complexity of the FCR global analysis as

$$T_{GA}^{FCR} = \frac{\beta NDt_1}{P}. \quad (2)$$

In the phase of executor, the loop iterations can be assigned to threads in either static or dynamic ways. The threads are synchronized based on mutex locks and condition variables associated with array elements. To exploit fine-grained reference level parallelism, each array element is associated with a pair of mutex lock and condition variables. To access an element, a thread first acquires a corresponding mutex lock. After obtaining the lock, it checks with the corresponding logical clock to see if the reference is ready for execution. After accessing the element, the thread updates the clock time and broadcasts the new time to threads that are blocked on this element. If multiple threads are then blocked on the element, they will compete again for the mutual exclusion lock after their releases.

During this phase, the time stamp algorithms require $O(N)$ space for synchronization variables and $O(N)$ space for logical clocks in addition to the $O(ND)$ space for the global stamp table. Since the auxiliary arrays *head* and *tail* are no longer needed in this phase, the space complexity of the algorithms in this phase is reduced to $O(ND)$.

The threads execute the target loop in parallel based on the dependence information contained in the stamp table. Let γ denote the parallel efficiency of the executor and t_2 represent the average cost for synchronized access to an array element. Then, the time complexity of the executor becomes

$$T_{EX} = \frac{N(Dt_2 + t_0)}{\gamma P}. \quad (3)$$

Ignoring the cost for barrier synchronization, we add T_{LA}, T_{GA}, and T_{EX} together and obtain the time complexity of the FCR algorithm as

$$T = \frac{(1+\alpha+\beta)NDt_1}{P} + \frac{N(Dt_2 + t_0)}{\gamma P}. \quad (4)$$

Recall that α and β represent the percentages of references in locally closed and open-read groups, respectively, and that γ represents parallel efficiency of loop execution. Evidently, γ is determined by α and β. The FCR algorithm aims to maximize γ at a cost of one more parsing of $(\alpha+\beta)$ percentage references in the loop iteration space. At the other extreme, the CTY algorithm excludes parallelism between consecutive reads in execution by setting $\alpha = 0$ and $\beta = 0$. The PCR algorithm exploits parallelism between references in locally closed read groups without incurring extra overhead, according to rule S2 in Section 3.2. However, references in open read groups, such as (9, 0)(10, 0) in Fig. 7, need an additional parse for stamping in the phase of global analysis. Hence, the time complexity of the PCR algorithm is of T with $\alpha = 0$.

In comparison with T_{SEQ}, we obtain a *necessary condition* for performance improvements due to runtime parallelization:

$$t_0 \geq \frac{((1+\alpha+\beta)\gamma t_1 + t_2)D}{\gamma P - 1}. \quad (5)$$

This inequality reveals that runtime parallelization algorithms will not yield speedup at all for loops that have very light computational requirements in each iteration, even in the ideal case of $\gamma = 100$ percent. We refer to the computational requirements t_0 in each iteration as *iteration workload*.

We note that the above time complexity analysis assumes read groups are uniformly distributed in the iteration space. A nonuniform distribution would lead to a severe load imbalance between threads and a higher cost for the global inspector.

6 Performance Evaluation

In this section, we evaluate the performance of the reference-level parallelization algorithms. For comparison, we include the results from an iteration-level algorithm due to Rauchwerger et al. (RAP, for short) [30]. The algorithm breaks down parallelization functions into three routines: *inspector*, *scheduler*, and *executor*. The inspector examines the memory references in a loop and constructs

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

```
X[N-1] = B[N-1]/U[N-1][0];
for (i=N-2; i≥0 ; i=i-1)
      sum = 0.0;
      for (j=1; j<numEntry(i); j++)
            k = colPtr[i][N-j-1];
            sum += U[i][j] * X[k];
      endfor
      delay();
      X[i] = (B[i]-sum)/U[i][0];
endfor
```

(a)

```
for (i=1; i<N ; i=i+1)
      sum = 0.0;
      for (j=1; j<N; j++)
            k = colPtr[i][N-j-1];
            if (k > 0)    // valid read/write
                  sum += U[i][N-j] * X[k];
      endfor
      delay();
      X[N-i-1] = (B[N-i-1]-sum)/U[i][0];
endfor
```

(b)

Fig. 12. A code fragment of the Gaussian elimination method for sparse matrices. (a) Original loop. (b) Normalized loop.

a reference-level dependence chain for each data element accessed in the loop. The scheduler then derives more restrictive iteration-level dependence relations from the reference-level dependence chains. Finally, the executor restructures the loop into a sequence of wavefronts, separated by barrier synchronizations. Iterations in the same wavefront are executed in parallel while iterations in different wavefronts are executed in order.

We applied both the reference-level and iteration-level parallelization algorithms to a Gaussian elimination loop and a set of synthetic loops on a Sun 12-way SMP server E4500. The machine is configured with 12 250MHz UltraSPARC processors and 1.5 GB of memory. Each processor module has an external 1 MB cache.

Performance of a runtime parallelization algorithm is dependent on a number of factors. Of foremost importance is iteration workload. Since any Inspector/Executor algorithm needs to parse array references at least once for the exploitation of cross-iteration dependences and each reference is rendered by a sequence of synchronization operations, (5) shows that time stamp algorithms would not yield speedup for loops that have very light computational requirements in each iteration. A number of other key factors include loop structure, array reference pattern, and loop distribution strategy. In the following, we examine the impact of the factors on the performance of the parallelization algorithms.

6.1 Impact of Iteration Workload

First, we consider a code fragment, as shown in Fig. 12a. It is derived from a Gaussian elimination solver with pivoting over sparse matrices. The matrix is first decomposed into a product of lower triangle matrix L and upper triangle matrix U. The code fragment shows the subsequent reduction process of solving equation $UX = B$. Due to its sparsity, the upper triangle matrix U is represented in an Ellpack-Itpack compact format [21]. Since the format contains only nonzero elements, an indirect matrix $colPtr$ is used to store the column indices of corresponding elements in matrix U. To examine the impact of iteration workload, a `delay` function is inserted into the loop body. Since vector B in the equation is often input-dependent and needs to be calculated at runtime, the loop delay partially reflects the computational requirements for the vector.

Notice that, unlike the loop model in Fig. 1b, the loop in Fig. 12a has a different number of references to array X at each iteration and the loop proceeds with a decreasing index i. To parallelize the loop using the proposed techniques, we first normalize it by reversing the index change and by inserting "void" references to the loop, as we illustrated in Section 3.1. The indirect access array $colPtr$ is updated accordingly to reflect the validity of the references. Fig. 12b presents the normalized loop.

We tested the parallelization algorithms on two representative sparse matrices: ARC130 and OLM500, from the Harwell-Boeing collection [26]. ARC130 is a 130×130 matrix with 304 nonzero entries derived by laser modeling. Its nonzero entries are distributed across the whole graph (average nonzero elements per column is 8 and standard deviation is 18). OLM500 is a 500×500 matrix with 1,996 entries from the Olmstead model for the flow of a layer of viscoelastic fluid headed from below. Its nonzero elements are located in a narrow band around the diagonal (average nonzeros per column is 4 and standard deviation is 0.089).

Fig. 13 plots the execution time of the code fragment with various iteration loads ranging from 0 to 600 microseconds. Due to the presence of cross-iteration dependences, any speculative parallelization technique would execute the loop in serial. Therefore, the sequential execution time (SEQ) represents a lower bound for speculative algorithms. Fig. 13a shows that the iteration-level parallelization technique (RAP) lags far behind the reference-level parallelization algorithms. This is because the loop is mostly sequential across iterations due to the loop-carried flow dependence. The RAP algorithm establishes a parallel execution schedule with 76 wavefront levels out of 130 iterations. In terms of the average degree of parallelism of each wavefront, the 12-way multiprocessor system was severely underutilized. Fig. 13a also shows that the total execution time, due to the iteration-level parallelization algorithm, is insensitive to the iteration workload. This is because the barrier synchronization overhead in the executor stage dominates the time.

In the case of OLM500, the iteration-level algorithm performs even worse. Its execution time is in the range of 1,050 to 1,090 milliseconds, which is too large to be presented in Fig. 13b together with the results from the reference-level algorithms. Due to the structure of the OLM matrix, no iteration-level parallelism has been found. That is, the RAP algorithm produces a schedule with 500 wavefront levels out of 500 iterations (i.e., sequential execution). The overhead in dependence

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

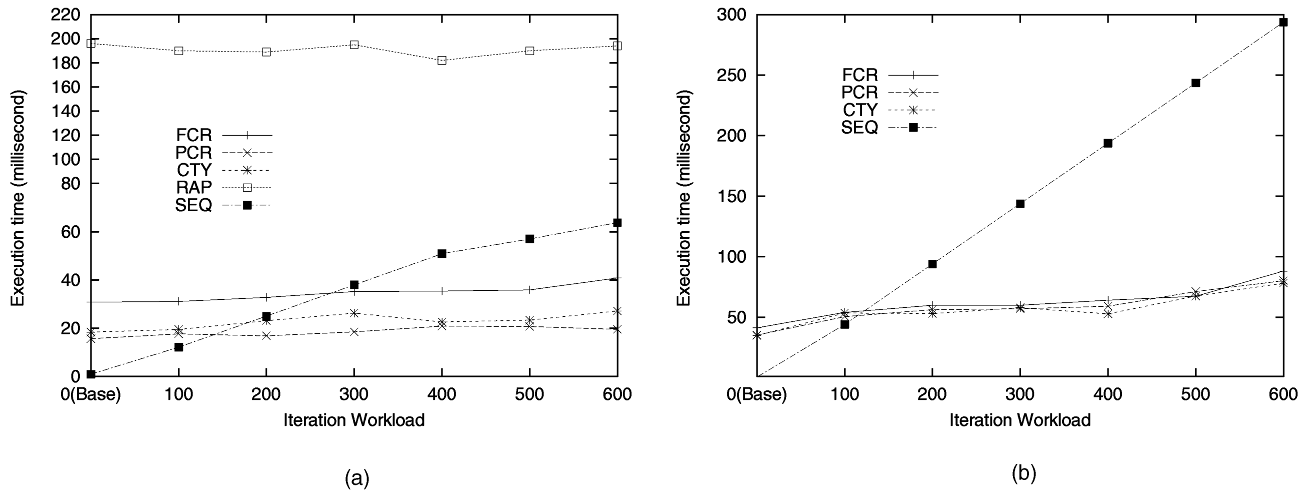

Fig. 13. Execution time of various parallelization algorithms over the loop of Fig. 12b with different delays. (a) ARC130. (b) OLM500.

analysis, scheduling, and barrier synchronization lead to a significant execution time.

In comparison with sequential execution, both figures clearly indicate that such dynamic loops hardly benefit from reference-level parallelization techniques if there is no extra iteration workload associated with array element references. However, these benefits become significant as the iteration load increases. Recall that the cost of time stamp algorithms for the local and global analyses is independent of the workload. The larger the iteration workload is, the smaller the execution time percentage for the analyses. Large iteration workloads can also amortize the synchronization overhead in the executor. This is the reason for the execution time for all reference-level algorithms to rise only slightly as the iteration load increases.

Of the reference-level algorithms, Fig. 13a shows that PCR outperforms CTY by 20 to 25 percent in the case of ARC130. This is because the loop instance contains 2,984 references, out of which 2,804 are formed into 283 local read groups. The average group size is 9.9, which represents a significant parallelism source for a 12-way multiprocessor system. The figure also shows that benefits obtained from more parallelism among consecutive reads, as exploited by the FCR algorithm, are outweighed by its extra overhead. The FCR algorithm establishes a total of 86 read groups across different iteration blocks of the threads. It increases the average read group size from 9.9 to 32.67. The overexposed parallelism might be another source of the performance loss.

In the case of OLM500, Fig. 13b shows that all the reference-level algorithms achieve nearly the same performance. With the input of OLM500, the loop instance contains 1,802 element references. The PCR algorithm forms 492 read groups out of 1,222 reads with an average group size of 2.48. In comparison with the CTY algorithm, benefits from the extra parallelism among small read groups are just outweighed by the PCR overhead. The FCR algorithm establishes a total of 552 read groups with an average group size of 2.35. Although it does not increase the read group size, FCR exploits 10 percent more read groups, which are across blocks. Note that we count two or more consecutive reads as a read group. FCR generates more read groups. This is because some read groups across blocks are due to multiple single reads in different blocks.

To better understand the relative performance of the parallelization algorithms, we break down their execution into a number of principal tasks and present their percentages of overall execution time in Fig. 14. From the figure, it can be seen that the RAP algorithm spends approximately 10 percent of the execution time in the construction of a wavefront-based schedule. The percentage remains unchanged as the iteration workload increases. This is because the total execution time is dominated by the barrier synchronization overhead in the executor. Recall that the PCR algorithm performs nearly the same work as CTY in the local analysis. However, Fig. 14a shows that the PCR algorithm spends a greater percentage of time in its local dependence analysis. This implies a reduction of the time in the executor phase due to the concurrency between consecutive reads. Fig. 14b shows the cost for dependence analysis as a percentage drops and the loop size increases. This is reflected in the increasing speedup over sequential execution, as shown in Fig. 13.

In summary, the experimental results show that both reference-level and iteration-level parallelization techniques benefit from dynamic loops whose iteration workload is heavy enough to amortize the runtime overhead. Although the results on a single application are not enough to reach any conclusion about their relative performances, they imply that the iteration-level algorithms critically depend on the cost of the barrier and are more sensitive to the degree of parallelism available in the loops. Of the reference-level algorithms, the PCR algorithm delivers the best results because it makes a better trade-off between maximizing the parallelism and minimizing the analysis overhead. The subsequent experiments on synthetic loops will verify these observations.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

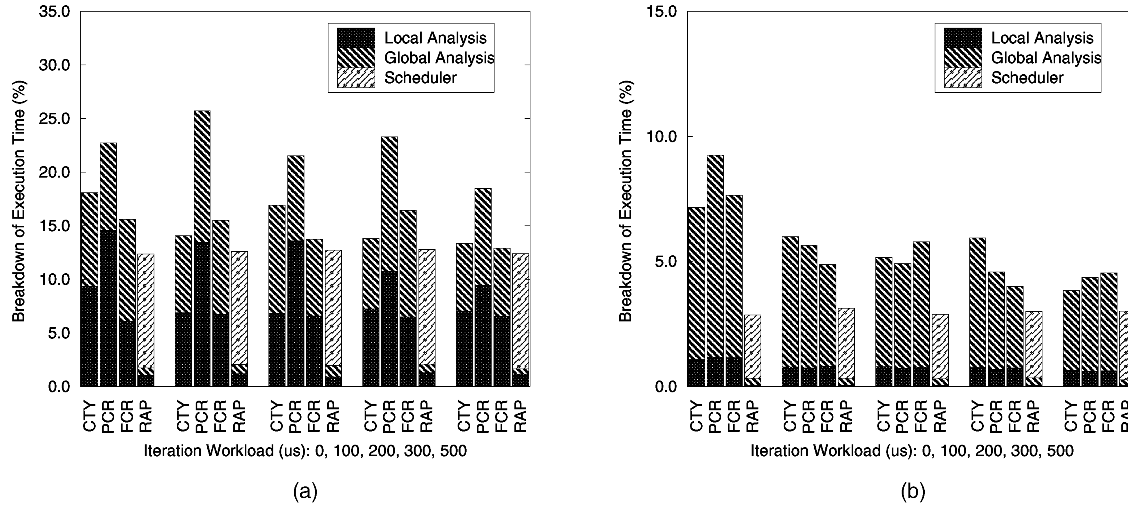

Fig. 14. Breakdown of the execution time by percentage. (a) ARC130. (b) OLM500.

6.2 Impact of Loop Structures and Array Reference Patterns

To examine the effect of loop structures and access patterns, we consider two kinds of synthetic loop structures: Single-Read-Single-Write (SRSW) and Multiple-Reads-Single-Write (MRSW), as shown in Fig. 15. The SRSW loop is comprised of a sequence of interleaved element reads and writes in the loop body. Each iteration executes a *delay()* function to reflect the iteration workload. In the MRSW loop, a write follows a sequence of reads. Such loop models were assumed by other researchers as well [9], [23], [30].

Array reference patterns are defined by the index array $u[i][j]$. Two reference patterns are considered: uniform and nonuniform access patterns. A uniform access pattern (UNIFORM, for short) assumes all array elements have the same probability of being accessed by an element reference. A nonuniform access pattern (NUNIFORM, for short) refers to the pattern where 90 percent of the references are to 10 percent of array elements. Nonuniform access patterns reflect hot spots in array accesses and result in long dependence chains.

The experiments measure the overall runtime execution time for a given loop and an array access pattern. Each data point obtained in the experiments is the average of five runs, using different seeds for the generation of pseudorandom numbers. Since a seed determines a pseudorandom sequence, algorithms are able to be evaluated under the same test instances.

Fig. 16 plots the parallel execution time of the loops with N ranges from 128 to 8,192, assuming the loop depth $D = 4$ and 100 microseconds of iteration workload. Loops in the RAP scheduler and the executor of the reference-level algorithms are decomposed in a cyclic way. The RAP executor is decomposed in a block way. Generally, a loop iteration space can be assigned to threads statically or dynamically [37], [36]. In [36], we experimented with three simple static assignment strategies: cyclic, block, and block-cyclic and showed that the reference level algorithms preferred cyclic or small block cyclic distributions because such distribution strategies lead to good load balance among processors.

Overall, Fig. 16 shows that both reference-level and iteration-level parallelization algorithms are capable of accelerating the execution of large loops in all test cases. Since the total workload of a loop increases with the loop size, runtime parallelization makes more sense as the loop size increases. This is in agreement with our observations from the preceding experiments.

Fig. 16 reveals the superiority of the reference-level algorithms to the iteration-level algorithm. Their relative performance gaps are extremely wide, particularly in the case of small loops or loops with nonuniform access

```
for (i=0; i<N; i++)
    for (j=0; j<D; j++)
        if (odd(j))
            dummy = X[u[i][j]];
        else
            X[u[i][j]] = dummy;
    endfor
    delay()
endfor
```

(a)

```
for (i=0; i<N; i++)
    for (j=0; j<D-1; j++)
        dummy = X[u[i][j]];
    X[u[i][D-1]] = dummy;
    delay()
endfor
```

(b)

Fig. 15. Single-Read-Single-Write (SRSW) versus Multiple-Reads-Single-Write (MRSW) loop structures. (a) SRSW loop structure. (b) MRSW loop structure.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

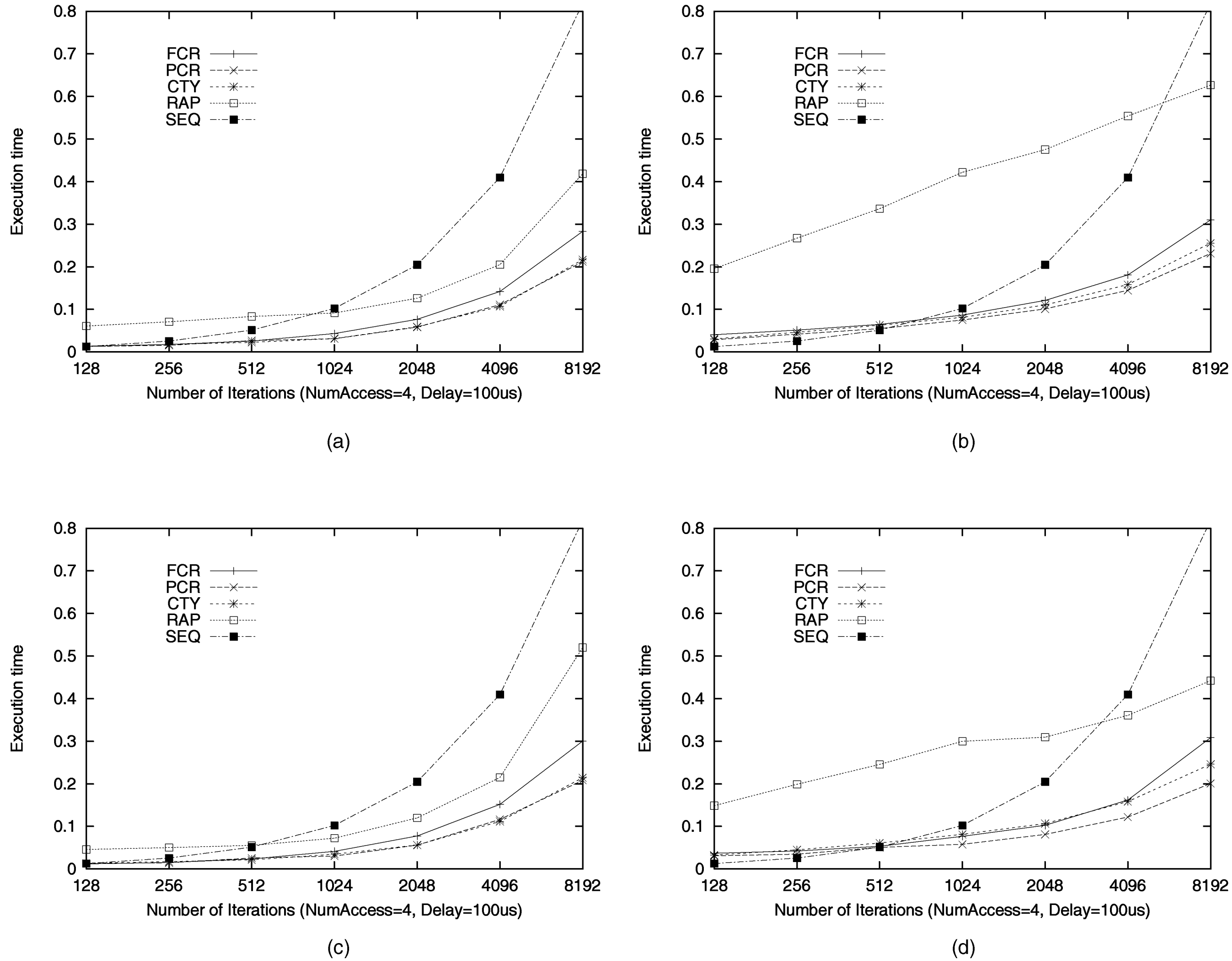

Fig. 16. Execution time of parallelization algorithms, where $D = 4$ and iteration load = 100μs. (a) SRSW and Uniform. (b) SRSW and Nonuniform. (c) MRSW and Uniform. (d) MRSW and Nonuniform.

patterns. Recall that the objective of loop parallelization is to expose enough parallelism to saturate available processors and meanwhile avoid over-exposure of parallelism. Since small loops or loops with nonuniform access patterns tend to have a limited amount of iteration-level parallelism, reference-level fine-grained parallelization techniques help increase the utilization of the available processors. For the same reasons, the reference-level algorithms achieve a higher speedup for small loops with uniform access patterns than those having nonuniform access patterns. The impact of access patterns decreases with loop size because the reference-level parallelism becomes excessive as the loop size increases. For example, the PCR algorithm achieves 1.92 times the speedup over sequential execution for a loop of uniform SRSW and of size 256. For a loop of the same type and of size 8,192, the algorithm obtains 3.86 times the speedup.

Of the reference-level algorithms, the PCR algorithm outperforms CTY for loops having high cross-iteration dependences, as shown in Fig. 16b and 16c, because of the extra parallelism between consecutive reads. Since the FCR algorithm needs to distinguish between read, write, and group consecutive reads of the same array element in the local and global inspector, benefits from exploiting extra parallelism for some loops are outweighed by the analysis overhead. In contrast, the PCR algorithm can obtain almost the same performance as CTY, even for loops with uniform access patterns, because it incurs only slightly more overhead in its local inspector.

From Fig. 16, it can also be seen that the reference-level algorithms are insensitive to loop structures (SRSW versus MRSW). It is because the algorithms treat read and write references equivalently in the executor. Each reference is rendered by a sequence of point-to-point synchronization operations.

By contrast, Fig. 16 depicts the large impact of array access patterns on the performance of the iteration-level algorithm. The figure also shows some effect of loop structure on the algorithm. To help explain the impact, we plot the average degrees of iteration-level parallelism inherent in different loops in Fig. 17. The degree of parallelism refers to the number of iterations in a wavefront. From the figure, it can be seen that the degree of parallelism of a loop is linearly proportional to its size. MRSW loops, with uniform and nonuniform access patterns, exhibit no more than 12 degrees of parallelism

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

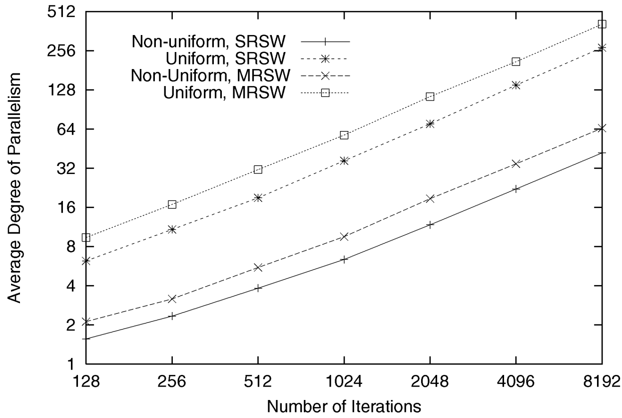

Fig. 17. Average degree of parallelism exploited by the iteration level algorithm.

until their size reaches beyond 256 and 2,048, respectively. This implies that the iteration-level algorithm for loops smaller than 256 or 2,048 would not gain any improvement when it is run on a 12-way multiprocessor system. This is in agreement with the execution time curves of the algorithm in Fig. 16c and 16d. Similar observations can be made for the SRSW loops.

Fig. 17 shows that, in comparison with the SRSW loop, the MRSW loop structure leads to an increase in the degree of parallelism by approximately 50 percent. This is equivalent to a reduction of the wavefront depth (i.e., the maximum wavefront level) of the schedule by half. Since the cost of a barrier is determined by the number of processors and is independent of the parallelism degree of a wavefront, the extra reads in the MRSW loop would cut half of the barrier overhead in the executor. This explains the RAP performance difference between Fig. 16b and 16d in the case of a nonuniform access pattern and between Fig. 16a and 16c in the case of a uniform access pattern. For loops with uniform access patterns, the benefits from the extra reads in the MRSW loop seem amortized by the over-exposed parallelism. Fig. 16 also shows big performance gaps between the RAP plots due to various array access patterns. This is because a uniform access pattern, in comparison with the nonuniform access pattern, increases the parallelism degree of RAP wavefronts by more than 6 times, as shown in Fig. 17.

To further illustrate the impact of the loop structure and the array reference pattern, we present the breakdown of the execution time of the algorithms in percentages in Fig. 18. First of all, the figure shows that the RAP algorithm spends a high percentage of time in the generation of wavefront-based schedules from reference-level dependence information. The percentage increases with the loop size in the case of a nonuniform access pattern, while the percentage rises and then drops in the case of a uniform access pattern. Let T_{WF} and T_{BAR} denote the average execution time of a wavefront and the barrier cost, respectively. It follows that the execution time of the RAP executor is

$$T_{EX} = n(T_{WF} + T_{BAR}),$$

where n is the wavefront depth of the RAP schedule. In the case of loops with a nonuniform access pattern, $T_{WF} << T_{BAR}$ because of the small average degree of parallelism. From Fig. 17, it is known that the average parallelism degree is linearly proportional to the loop size. That is, T_{WF} increases with the loop size while the wavefront depth n remains nearly unchanged. This leads to a slow change of T_{EX}. Since the overhead of the scheduler is proportional to the loop size, its relative overhead is increased accordingly. In the case of large loops with a uniform access pattern, $T_{WF} >> T_{BAR}$ because of the large parallelism degree of a wavefront. Since T_{WF} increases with the loop size, so does the execution time T_{EX}. This is why the percentage of scheduler overhead drops in this case.

Of the reference-level algorithms, their global analyses incur most of the overhead. The overhead decreases as the loop size increases and, hence, leads to larger speedups for large loops. In applications like computational fluid dynamics and molecular dynamics, it is common that a dynamic loop, as shown in Fig. 2, is to be executed repeatedly and the dependence information exploited in the inspector is able to be reused across many loop executions. In these cases, the PCR and FCR are expected to achieve better performances.

7 Concluding Remarks

In this paper, we have presented a time stamp algorithm for runtime parallelization of DOACROSS loops that have indirect access patterns. The algorithm follows the Inspector/Executor scheme and exploits parallelism at a fine-grained memory reference level. It features a parallel exploitation of parallelism, particularly the parallelism between consecutive reads of the same memory locations. Two variants of the algorithm have been evaluated: One allows partially concurrent reads (PCR) and the other allows fully concurrent reads (FCR).

We have analyzed their complexities and shown that performance of the time stamp algorithms is critically dependent on the loop iteration workload. Since the algorithms cause nonnegligible overhead, parallelization is only beneficial to loops that have fairly large workloads. Experimental results for a Gaussian elimination loop and an extensive set of synthetic loops on a 12-way SMP have verified the above claim for any Inpector/Executor algorithm, including coarse-grained iteration-level parallelization algorithms. For loops with light iteration workload, an alternative speculative runtime parallelization technique is preferred.

Of the time stamp algorithms, we have shown that the PCR makes a better trade-off between maximizing the parallelism and minimizing the analysis overhead. The FCR algorithm is expected to be best suited for loops that are to be executed repeatedly. We have also shown that the time stamp algorithms outperform the iteration-level algorithm in most test cases because the latter is critically dependent upon the barrier cost and sensitive to loop structures and array access patterns.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

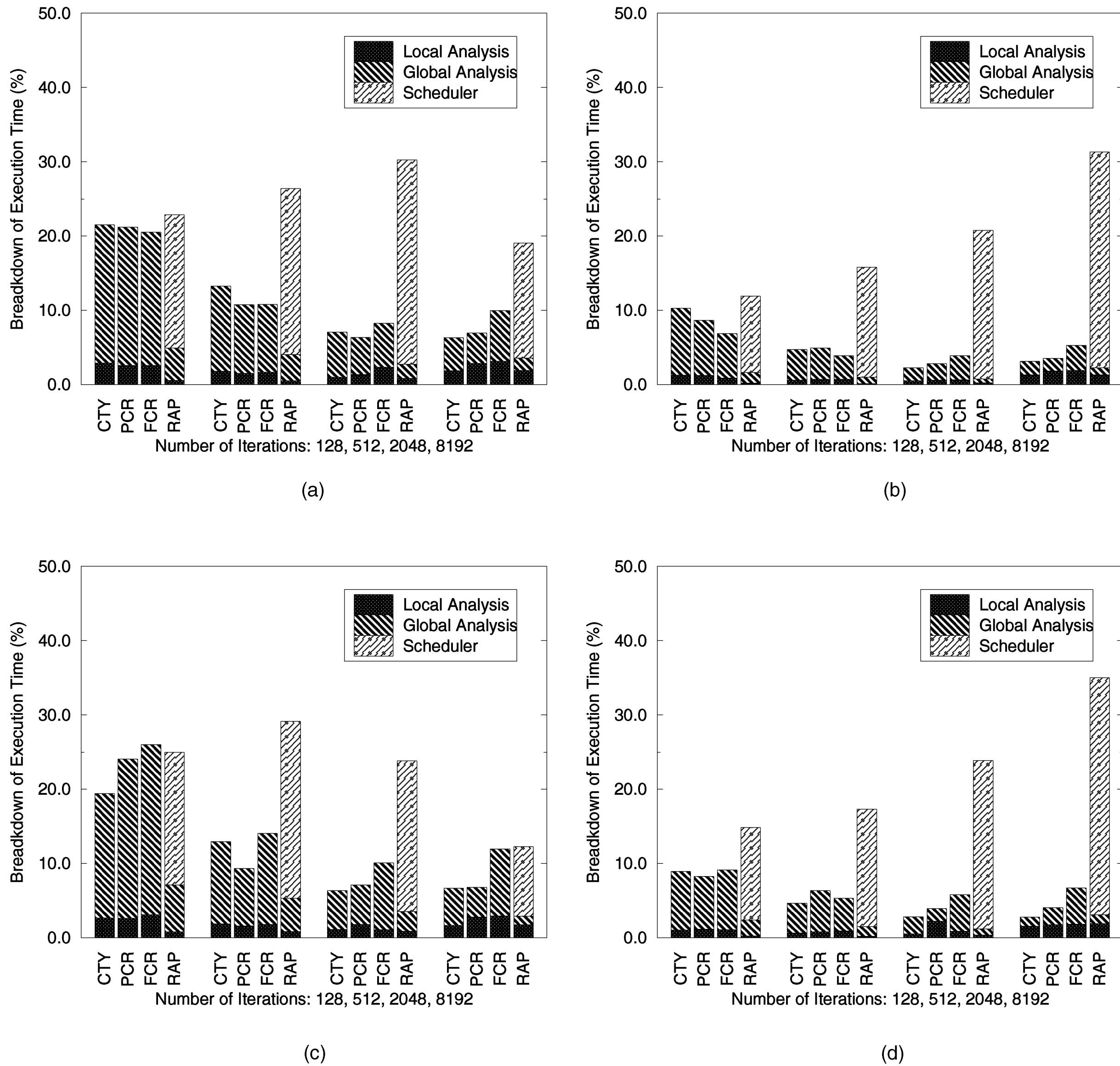

Fig. 18. Breakdown of exectution time by percentage on different stages. (a) SRSW and Uniform. (b) SRSW and Nonuniform. (c) MRSW and Uniform. (d) MRSW and Nonuniform.

We note that the time stamp algorithms are targeted at DOACROSS loops with any type of cross-iteration dependences. They have been compared with previous algorithms of the same generality. We have examined many benchmarks in public domains and found most of the loops are in special forms (e.g., without output dependences). Since loops of special forms can be parallelized efficiently by other techniques, as we reviewed in Section 2, the time stamp algorithms have not been experimented with over such loops. While the algorithms for arbitrary loop models have limited applications in today's benchmarks, we argue that the commonplace of the special loops is partly because all benchmarks are efficiently coded by well-trained programmers. However, it may not be the case in real world applications.

We also note that runtime parallelization incurs non-negligible overhead. The overhead cannot be amortized in execution without enough iteration workload. A major source of the runtime overhead, with the time stamp algorithms, is barrier synchronization and synchronized memory references between threads. Both barrier and synchronized access were implemented in a blocking scheme. We expect lightweight busy-waiting implementations would reduce the runtime overhead and, consequently, relax the requirement for runtime parallelization to some extent. Support for thread level parallelism is gaining momentum [5], [10], [34]. With architectural support for multithreading and fast synchronization primitives, reference-level runtime parallelization techniques, including the time stamp algorithms, are expected to be more efficient for dynamic loops.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

ACKNOWLEDGMENTS

This work was sponsored in part by US National Science Foundation MIP-9309489, EIA-9729828, EIA-9977815, CCR-9988266, US Army Contract DAEA-32-93-D-004, and Ford Motor Company Grant #0000952185. We thank the anonymous reviewers for their very detailed constructive comments and suggestions. Thanks also go to Loren Schwiebert, Pen-Chung Yew, and David Padua for their discussions and valuable comments on this work.

REFERENCES

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," *Proc. Conf. 10th ACM Symp. Principles of Programming Languages,* pp. 177-189, Jan. 1983.
[2] D.F. Bacon, S.L. Graham, and O.J. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys,* vol. 26, no. 4, pp. 345-420, Dec. 1994.
[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua, "Automatic Program Parallelization," *Proc. IEEE,* vol. 81, no. 2, 1993.
[4] W. Blume and W. Eigenmann, "Nonlinear and Symbolic Data Dependence Testing," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 12, pp. 1180-1194, Dec. 1998.
[5] L. Carter, J. Feo, and A. Snavely, "Performance and Programming Experience on the Tera MTA," *Proc. SIAM Conf. Parallel Processing for Scientific Computing,* 1999.
[6] D.-K. Chen and P.-C. Yew, "An Empirical Study on Doacross Loops," *Proc. Supercomputing,* pp. 620-632, 1991.
[7] D.-K. Chen and P.-C. Yew, "On Effective Execution of Nonuniform DOACROSS Loops," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 5 pp. 463-476, May 1996.
[8] D.-K. Chen and P.-C. Yew, "Redundant Synchronization Elimination for DOACROSS Loops," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 5, pp. 459-470, May 1999.
[9] D.-K. Chen, P.-C. Yew, and J. Torrellas, "An Efficient Algorithm for the Run-Time Parallelization of Doacross Loops," *Proc. Supercomputing,* pp. 518-527, Nov. 1994.
[10] "Overview of Cray MTA Multithreaded Architecture System," Cray Research Inc., http://tera. com/products/systems/craymta, 2000.
[11] D. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1998.
[12] R. Cytron, "DOACROSS: Beyond Vectorization for Multiiprocessors," *Proc. Int'l Conf. Parallel Processing,* pp. 836-844, 1986.
[13] R. Eigenmann, J. Hoeflinger, and D. Padua, "On the Automatic Parallelization of the Perfect Benchmarks," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 1, pp. 5-23, Jan. 1998.
[14] M. Gupta and R. Nim, "Techniques for Speculative Runtime Parallelization of Loops," *Proc. Supercomputing (SC98),* 1998.
[15] J. Hoeflinger, "Run-Time Dependence Testing by Integer Sequence Analysis," technical report, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana-Champaign, Jan. 1992.
[16] A. Hurson, K. Kavi, and J. Lim, "Cyclic Staggered Scheme: A Loop Allocation Policy for DOACROSS Loops," *IEEE Trans. Computers,* vol. 47, no. 2, pp. 251-255, Feb. 1998.
[17] A. Hurson, J. Lim, K. Kavi, and B. Lee, "Parallelization of DOALL and DOACROSS Loops: A Survey," *Advances in Computers,* vol. 45, 1997.
[18] Y.-S. Hwang, R. Das, J. Saltz, B. Brooks, and M. Hodoscek, "Parallelizing Molecular Dynamics Programs for Distributed Memory Machines: An Application of the CHAOS Runtime Support Library," *IEEE Computational Science and Eng.,* vol. 2, no. 2, pp. 18-29, 1995.
[19] J. Ju and V. Chaudhary, "Unique Sets Oriented Partitioning of Nested Loops with Nonuniform Dependences," *Computer J.,* vol. 40, no. 6, pp. 322-339, 1997.
[20] V. Krishnan and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Trans. Computers,* vol. 48, no. 9, pp. 866-880, Sept. 1999.
[21] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing.* Benjamin/Cummings, 1994.
[22] S.T. Leung and J. Zahorjan, "Improving the Performance of Runtime Parallelization," *Proc. Fourth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 83-91, May 1993.
[23] S.T. Leung and J. Zahorjan, "Extending the Applicability and Improving the Peformance of Runtime Parallelization," technical report, Dept. of Computer Science, Univ. of Washington, 1995.
[24] J. Lim, A. Hurson, K. Kavi, and B. Lee, "A Loop Allocation Policy for DOACROSS Loops," *Proc. Symp. Parallel and Distributed Processing,* pp. 240-249, 1996.
[25] S. Midkiff and D. Padua, "Compiler Algorithms for Synchronization," *IEEE Trans. Computers,* vol. 36, no. 12, Dec. 1987.
[26] "Harwell-Boeing Collection of Sparse Matrices," Nat'l Inst. of Standards and Technology, http://math.nist.gov/matrixMarket/data/Harwell-Boeing, 2000.
[27] J. Oplinger, D. Heine, S.-W. Liao, B. Nayfeh, M. Lam, and K. Olukotun, "Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor," Technical Report CSL-TR-97-715, Computer Science Lab., Stanford Univ., Feb. 1997.
[28] D. Patterson and J. Hennessy, *Computer Architecture: A Quantiative Approach,* second ed., Morgan Kaufmann, 1996.
[29] L. Rauchwerger, "Run-Time Parallelization: Its Time Has Come," *Parallel Computing,* vol. 24, nos. 3-4, pp. 527-556, 1998.
[30] L. Rauchwerger, N. Amato, and D. Padua, "Run-Time Methods for Parallelizing Partially Parallel Loops," *Int'l J. Parallel Programming,* vol. 23, no. 6, pp. 537-576, 1995.
[31] L. Rauchwerger and D. Padua, "The LRPD Test: Speculative Runtime Parallelization of Loops with Privatization and Reduction Parallelization," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 2, pp. 160-180, Feb. 1999.
[32] J. Saltz, R. Mirchandaney, and K. Crowley, "Run-Time Parallelization and Scheduling of Loops," *IEEE Trans. Computers,* vol. 40, no. 5, May 1991.
[33] Z. Shen, Z. Li, and P.C. Yew, "An Empirical Study on Array Subscripts and Data Dependencies," *Proc. Int'l Conf. Parallel Processing,* vol. II, pp. 145-152, 1989.
[34] J.-Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.-C. Yew, "The Superthreaded Processor Architecture," *IEEE Trans. Computers,* vol. 48, no. 9, pp. 881-902, Sept. 1999.
[35] T.H. Tzen and L.M. Ni, "Dependence Uniformization: A Loop Parallelization Technique," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 5 pp. 547-558, May 1993.
[36] C. Xu, "Effects of Parallelism Degree on Runtime Parallelization of Loops," *Proc. 31st Hawaii Int'l Conf. Systems Science,* pp. 86-95, 1998.
[37] C. Xu and F. Lau, *Load Balancing in Parallel Computers: Theory and Practice.* Kluwer Academic, 1997.
[38] Y. Yan, X. Zhang, and Z. Zhang, "A Memory-Layout Oriented Runtime Technique for Locality Optimization," *Proc. Int'l Conf. Parallel Processing,* 1998.
[39] H. Yu and L. Rauchwerger, "Techniques for Reducing the Overhead of Runtime Parallelization," *Proc. 9th Int'l Conf. Compiler Construction,* Mar. 2000.
[40] C. Zhu and P.C. Yew, "A Scheme to Enforce Data Dependence on Large Multi-Processor Systems," *IEEE Trans. Software Eng.,* vol. 13, no. 6, pp. 726-739, June 1987.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.

Cheng-Zhong Xu received the BS and MS degrees from Nanjing University, China, in 1986 and 1989, respectively, and the PhD degree from the University of Hong Kong, Hong Kong, China, in 1993, all in computer science. He is currently an associate professor of electrical and computer engineer at Wayne State University. Prior to joining Wayne State, he was a guest professor in the Department of Computer Science at Paderborn University, Germany, from January 1994 to February 1995. Dr. Xu has published more than 40 peer-reviewed papers in various journals and conference proceedings in the areas of resource management in parallel systems, parallelizing compilers, high performance cluster computing, mobile agent technology, and semantics-based web information retrieval. He is a coauthor of the book *Load Balancing in Parallel Computers: Theory and Practice* (Kluwer Academic, 1997) and the creator of Naplet, the mobile agent system. He has served on technical program committees of numerous professional conferences, including the IEEE International Conference on Distributed Computing Systems (ICDCS'2001), IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'1999-2001), and the First International Workshop on Internet Computing and E-Commerce (ICEC 2001). He is a recipient of the year 2000 "Faculty Research Award" at Wayne State University. He is a member of the IEEE Computer Society.

Vipin Chaudhary received the BTech (Hons.) degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1986, where he was awarded the President of India Gold Medal for academic excellence. He received the MS degree in computer science and the PhD degree in electrical and computer engineering from the University of Texas at Austin, in 1989 and 1992, respectively. He is currently an associate professor of electrical and computer engineering and computer science and he is the associate director of the Institute for Scientific Computing at Wayne State University. He is also the senior director for Advanced Development at Cradle Technologies, Inc. From January 2000 to January 2001, he was chief architect at Corio, Inc. From January 1992 to May 1992, he was a postdoctoral fellow with the Computer and Vision Research Center, the University of Texas at Austin. His current research interests are programming environments for high performance parallel and distributed systems, scientific computing, high performance networking and I/O, mobile computing, and bioinformatics. He has published more than 50 peer reviewed papers in the broad areas of parallel and distributed computing, image processing, security, and scientific computing. He has served on program committees for numerous international conferences in the above areas. He received the Research Initiation Award from National Science Foundation (NSF) in 1993 and his research has been funded by NSF, Army Research Labs., Cray Research, Inc., IBM, and Ford. He is on technical advisory boards for several companies. He is also a member of the IEEE Computer Society.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 23, 2008 at 12:20 from IEEE Xplore. Restrictions apply.