RUN-TIME OPTIMIZATION OF ADAPTIVE IRREGULAR APPLICATIONS

A Dissertation

by

HAO YU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2004

Major Subject: Computer Science

RUN-TIME OPTIMIZATION OF

ADAPTIVE IRREGULAR APPLICATIONS

A Dissertation

by

HAO YU

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Approved as to style and content by:

Lawrence Rauchwerger (Chair of Committee)	
Nancy Amato (Member)	Vivek Sarin (Member)
Marvin L. Adams (Member)	Valerie E. Taylor (Head of Department)

August 2004

Major Subject: Computer Science

ABSTRACT

Run-Time Optimization of Adaptive Irregular Applications. (August 2004)

Hao Yu, B.S., Tsinghua University, PR China;

M.S., Tsinghua University, PR China

Chair of Advisory Committee: Dr. Lawrence Rauchwerger

Compared to traditional compile-time optimization, run-time optimization could offer significant performance improvements when parallelizing and optimizing adaptive irregular applications, because it performs program analysis and adaptive optimizations during program execution. Run-time techniques can succeed where static techniques fail because they exploit the characteristics of input data, programs' dynamic behaviors, and the underneath execution environment.

When optimizing adaptive irregular applications for parallel execution, a common observation is that the effectiveness of the optimizing transformations depends on programs' input data and their dynamic phases. This dissertation presents a set of run-time optimization techniques that match the characteristics of programs' dynamic memory access patterns and the appropriate optimization (parallelization) transformations.

First, we present a general adaptive algorithm selection framework to automatically and adaptively select at run-time the best performing, functionally equivalent algorithm for each of its execution instances. The selection process is based on off-line automatically generated prediction models and characteristics (collected and analyzed dynamically) of the algorithm's input data, In this dissertation, we specialize this framework for automatic selection of reduction algorithms. In this research, we have identified a small set of machine independent high-level characterization parameters

and then we deployed an off-line, systematic experiment process to generate prediction models. These models, in turn, match the parameters to the best optimization transformations for a given machine. The technique has been evaluated thoroughly in terms of applications, platforms, and programs' dynamic behaviors. Specifically, for the reduction algorithm selection, the selected performance is within 2% of optimal performance and on average is 60% better than "Replicated Buffer," the default parallel reduction algorithm specified by OpenMP standard.

To reduce the overhead of speculative run-time parallelization, we have developed an adaptive run-time parallelization technique that dynamically chooses efficient shadow structures to record a program's dynamic memory access patterns for parallelization. This technique complements the original speculative run-time parallelization technique, the LRPD test, in parallelizing loops with sparse memory accesses.

The techniques presented in this dissertation have been implemented in an optimizing research compiler and can be viewed as effective building blocks for comprehensive run-time optimization systems, e.g., feedback-directed optimization systems and dynamic compilation systems.

To Haijing, Eric, mom, dad and Meng.

ACKNOWLEDGMENTS

I feel extremely fortunate to have had Dr. Lawrence Rauchwerger as my advisor. I wish to thank him for his inspiration, technical direction and material support throughout my time at Texas A&M. His unfailing excitement in research and fully-dedicated work style have made him a great leader and role model for me. Without his help this dissertation would not have been possible. He is also a very caring person, willing to offer help at any time. It was he who made my life in College Station much easier and my memories of Texas A&M much richer.

I want to express my gratitude to the members of my advisory committee, Nancy Amato, Vivek Sarin, and Marvin Adams, and my Graduate Council Representative, Arthur Hobbs, for their interest, valuable insights and earnest help.

During my years at Texas A&M, I have been given the opportunity to work with excellent students and researchers, especially the previous and current members of our Parasol compiler group: Francisco Arzu, Julio Carvallo de Ochoa, Francis Dang, Larry Evans, Guobin He, Tao Huang, Keith Jackson, Alin Jula, William McLendon III, Devang Patel, Lidia Onica, Koji Ouchi, Silvius Rus, Steven Saunders, Timmie Smith, Nageswar Tagarathi, Gabriel Tanase, Nathan Thomas, and Dongmin Zhang. I thank them for building a research environment filled with curiosity, excitement, inspiration, and joy. In term of system administration, Timmie, Jack Purdue, Robert Main, and Francis have always been understanding and helpful during my busy research.

In addition, I have always appreciated my experience at the Texas A&M Supercomputing Center, where I have worked with some excellent colleagues: Spiros Vellas, Khalid Sarwar Warraich, Vassilis Kostovassilis, Faisal Chaudhry, Zdenko Tomasic, Keith Jackson, and Michael Thomadakis. They have not only taught me many things, from parallel systems to program optimization, but also helped me with some of the

experiments presented in this dissertation.

Finally, I would like to thank my family for their support and love in my endeavors. I cannot think of words to express my gratitude to them. It is my life-long task to return this debt to them.

TABLE OF CONTENTS

CHAPTER Page

Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

A. Motivation

Improving performance on current parallel processors is a very complex task which, if done "by hand" by programmers, becomes increasingly difficult and error prone. Programmers have obtained increasingly more help from parallelizing (restructuring) compilers; such compilers address the need to detect and exploit parallelism in sequential programs written in conventional languages as well as parallel languages (e.g., HPF). They also optimize data layout and perform other transformations to reduce and hide memory latency, the other crucial optimization in modern large scale parallel systems. The success in the "conventional" use of compilers to automatically optimize code is limited to cases when performance is independent of the input data of the applications.

A large family of applications that traditional optimizing compiler techniques can not effectively utilize are "irregular applications." Irregular applications usually represent systems in the form of sparse and irregular structures, which correspond to sparse metrics or graphs/meshes representing irregular geometries. Such programs have been extensively used in most scientific and engineering computational domains. For instance, N-body simulation and Molecular Dynamic applications [1], which model particle systems at the particle level, are widely used in disciplines such as Physics, Chemistry [2], Biology [3], etc. Another application domain is computation fluid dynamics (CFD), which is widely used in most scientific or engineering disciplines, e.g. Mechanical Engineering, Astrophysics [4], Nuclear Engineering [5],

---

The journal model is *IEEE Transactions on Parallel and Distributed Systems.*

Geophysics [6], etc., to simulate or model a fluid system at the "flux" granule-level.

In most irregular applications, the computation depends on indirect data structures. For instance, sparse metrics or graphs are usually stored in compact formats to reduce the storage requirement and computation time. The compact representation must not only store the attributes of the nodes and/or edges, but also store various indexing information to access the attributes. For such applications, the information needed for optimization purposes is not available at compile-time because the contents of such indirect data structures are often read from input data or computed during program execution. Therefore, the traditional optimization approach (via optimizing compilers) can hardly deliver a satisfactory performance.

The challenge raised by irregular applications has been addressed via run-time optimization, which performs program analysis and adaptive optimization during program execution. Run-time techniques can succeed where compilation fails because they have access to the input data. For example, input dependent or dynamic data distribution, memory accesses guarded by run-time dependent conditions, and complex subscript expressions can all be analyzed unambiguously at run-time.

A very important experience in parallel applications, in comparison to sequential applications, is that their performance is much more sensitive to input data, system architecture, and dynamic execution environment. Many important (frequently used and time consuming) algorithms used in such programs are indeed application-, input-, and environment-dependent. For example, we have previously shown that, parallel reduction algorithms are quite sensitive to their input memory reference patterns and system architectures [7]. Similar conclusions have been reached for parallel sorting algorithms [8], for FFT algorithms [9], and for regular computations such as matrix multiplication [10]. In general, adaptive run-time optimization is the key to achieving improved performance when running irregular applications on today's parallel

systems, and performance can be greatly enhanced if we can adaptively tailor the optimizations to the particular program execution instance,

B. Thesis

The goal of this dissertation is to explore adaptive run-time optimization techniques to efficiently parallelize irregular applications. Specifically, this dissertation presents novel compiler and run-time techniques that allow irregular applications to adaptively select algorithms or data structures that are most suited to a particular execution instance.

When codes are irregular (memory references are irregular) and/or dynamic (memory reference patterns change during the same program execution instance), it is very likely that important performance-affecting program characteristics will be input- and environment-dependent. Our preliminary research has demonstrated that one of the most powerful optimization methods compilers can employ is to substitute entire algorithms instead of trying to perform low level optimizations on sequences of code [7]. In Chapter II, we first present a general framework to automatically and adaptively select at run-time the best performing, functionally equivalent algorithm for each of its execution instances. The selection process is based on off-line automatically generated prediction models and characteristics of the algorithm's input data which is collected and analyzed dynamically. While the framework has also been applied to adaptively select sorting algorithms [8], in this dissertation, we concentrate on the automatic selection of reduction algorithms.

In Chapter III, we discuss reduction parallelization and present a small library of parallel reduction algorithms. With experiments, we demonstrate that the best performance can be obtained only if we dynamically select the most appropriate one for a

particular program-input combination. Then in Chapter IV, we present a systematic and automatic process for generating prediction models that match the parallel reduction algorithms to execution instances of reduction loops. After establishing a small set of high-level parameters that can characterize irregular memory reference patterns, we measure the relative performance of the candidate algorithms for a number of synthetic reduction execution instances in a factorial experiment. This is achieved by running a synthetic reduction loop which generates reduction references with the memory reference patterns selected by the factorial experiment. The end result of this off-line process is a mapping between various points in the memory reference pattern space and the best available reduction algorithm. At run-time, the memory reference characteristics of the actual reduction loop are extracted and matched through a regression to the corresponding best algorithm (using the previously extracted map).

With the experimental results presented in Chapter V, we show that our framework can select the actual best performing algorithms for 85% of the cases studied and the overall selected performance is within 2% of optimal performance. We also show that with our framework, a dynamic program can achieve performance improvements that are not otherwise possible (e.g., applying one algorithm).

The main contribution of this work is an adaptive framework for a systematic process through which input sensitive prediction models can be built off-line and used to dynamically select from a particular list of functionally equivalent algorithms (parallel reductions being just one important example). The same approach can also be used for various other compiler transformations that cannot be easily analytically modeled.

In my dissertation research, we have also developed adaptive run-time techniques to complement existing run-time parallelization techniques. Run-time parallelization techniqu is to detect and explore parallelism at run-time. In general it inserts extra

codes into the original program to facilitate data dependence test. Due to the dynamic natures of irregular programs, it is difficult to always achieve good performance with the same run-time parallelization technique. In Chapter VI, we present an adaptive run-time parallelization technique which can efficiently explore the parallelism of loops with sparse memory accesses. The theme of the technique is adaptively selecting the most efficient shadow data structures to record the memory reference patterns executed by a speculatively parallelized loop.

In Chapter VII, we review related research efforts in the fields of "adaptive optimization," "reduction parallelization," and "automatic parallelization." Finally, in Chapter VIII, we summarize the techniques that have been developed and presented in this dissertation and outline their possible extensions.

CHAPTER II

ADAPTIVE ALGORITHM SELECTION FRAMEWORK

In this chapter we give an overview of our general framework for adaptive and automatic low level algorithm selection, the details of which are presented in Chapters III – V as applied to the optimization of parallel reduction algorithm selection.

In comparison to sequential computing, parallel algorithms for irregular applications are much more sensitive to their data access patterns, system architecture, and environment. Specifically, the relative performance of several equivalent parallel algorithms is application-, input-, and time-dependent. Therefore, for most cases, the performance can be greatly enhanced if we can tailor the choice of algorithm and its parameters to the particular instance in which it is used.

Fig. 1 gives an overview of our adaptive framework. We distinguish two phases: (a) a setup phase and (b) a dynamic selection (optimization) phase.

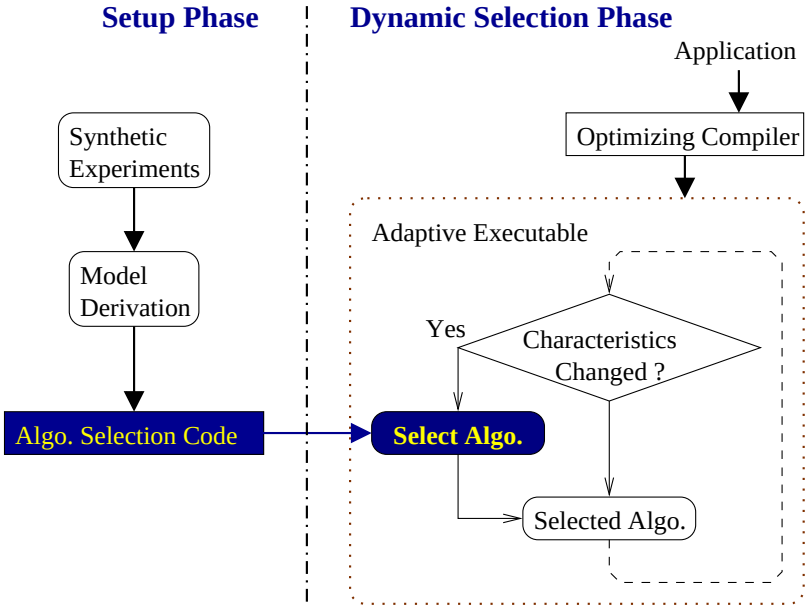


Fig. 1. Overview of the adaptive algorithm selection framework

The *setup phase* occurs once for each computer system and thus, implicitly tailors our process to a particular architecture. We then establish the input domain and the output domain (possible optimizations) of the algorithm selection code.

In the particular case presented in this dissertation (i.e., parallel reductions), the input domain is the universe of all possible and realistic memory reference patterns — because, as previously shown [7], they crucially impact the obtained performance. Architecture type is also important, but is used implicitly. Since it would be impractical to study the entire universe of memory access patterns, we define a small set of parameters that can sufficiently characterize them. The domain of possible optimizations is composed, in our case, of the different parallel reduction algorithms collected in a library.

We then explore our input domain and find a mapping to the output domain. In our case we establish a mapping between different points in the input parameter space (memory reference patterns) and relative performance rankings of the available algorithms. This task is accomplished off-line by running a factorial experiment. We generate a number of parameter sets that have the potential cover our input domain. For each of these data points, we measure the relative performance of our algorithms on the particular architecture, and rank them accordingly.

We should mention here that we have also tried other methods of exploring the data space. In [11], we have used a machine learning algorithm to explore the input space that defines the performance.

The *dynamic selection phase* occurs during actual program execution. Through instrumentation (or otherwise) we extract the set of relevant parameters that characterizes the actual input. Then we use the information obtained during the setup phase to find its corresponding output. Specifically, we use statistical regression to eventually select the appropriate best performing parallel reduction algorithm. In [11], in

turn, we used statically generated decision trees which are dynamically traversed to select the best algorithms.

In this dissertation we specialize our adaptive framework to parallel reduction algorithm selection. We first developed a library of reduction algorithms and identified a set of parameters that can characterize their irregular memory reference patterns. Then we applied a factorial experiment that explores the input space. For each parameter set, we execute a synthetic parameterized loop that generates a memory reference pattern on which we evaluate and rank the different algorithms in our library. A regression method is used to compute prediction models for each parallel reduction algorithm based on the data from the factorial experiment. Finally, at run-time, we compute values for the characterization parameters of the reduction operation in question and use them in our pre-computed models to select the best algorithmic option.

CHAPTER III

IRREGULAR REDUCTION PARALLELIZATION

A. Essentials of Reduction Parallelization

A special and very frequent case of loop dependence patterns occurs in loops which implement reduction operations. In particular, reductions (also known as updates) are at the core of a very large number of algorithms and applications – both scientific and otherwise – and there is a large body of literature dealing with their parallelization.

A reduction variable is a variable whose value is used in one associative and commutative operation of the form $x = x \otimes expr$, where $\otimes$ is the operator and x does not occur in the *expr* or anywhere else in the loop. A simple example is statement `S1` in Fig. 2-(a). Note code or pseudo-code pieces illustrated in this dissertation align to FORTRAN The operator $\otimes$ is exemplified by the + operator, and the access pattern of array `A` is *read*, *modify*, and *write*. The function performed by the loop is to add values computed in each iteration (of the outer loop) to the values stored in array `A`. This type of reduction is sometimes called an *update*.

With the exception of some simple methods using unordered critical sections (locks), reduction parallelization is performed through a simple form of algorithm substitution. For example, a sequential summation is a reduction which can be replaced by a parallel prefix, or recursive doubling, computation [12, 13].

Irregular reductions usually refer to the `S1` statement in Fig. 2-(b). The data (e.g., array `A`) are updated in multiple iterations of the loop using associative and commutative operations and the data elements are then accessed using indirection arrays (e.g., array `X`). In irregular applications (typical examples are simulation programs and programs involving sparse linear algebra), irregular reductions consume

```
      real   A(n)
      do i = 1,m
         do j = 1,n
S1          A(j) = A(j) + expr
         enddo
      enddo
```

(a) Regular reduction loop

```
      real    A(n)
      integer X(m)
      do i = 1, m
S1       A(X(i)) = A(X(i)) + expr
      enddo
```

(b) Irregular reduction loop

```
      do i = 1,m
S1       A(K(i)) = ...
S2       ... = A(L(i))
S3       A(X(i)) = A(X(i)) + expr
      enddo
```

(c) Reduction needing run-time validation

Fig. 2. Examples of reduction loops

large portion of the programs' execution time.

In general, there are two tasks required for reduction parallelization: *recognizing the reduction variable*, and *parallelizing the reduction operation*. Here we briefly describe the available static and run-time techniques for recognizing reduction variables. In the remainder of the chapter, we assume that such techniques have been used appropriately and we have a known reduction operation. Our goal is to select the best parallel reduction algorithm for performing it in parallel.

Static reduction recognition. The problem of recognizing reduction statements has usually been handled at compile-time by syntactically pattern matching the loop statements with templates of generic reductions, and then performing a data dependence analysis of the variable under scrutiny to guarantee that it is not used anywhere

else in the loop except in the reduction statements [14, 15, 16].

Run-time reduction validation. In cases where data dependence analysis cannot be performed at compile-time, reductions have to be validated at run-time. For example, in Fig. 2-(c), although statement `S3` in the loop matches a reduction statement, it is still necessary to check at run-time that the elements of array `A` referenced in `S1` and `S2` do not overlap with those accessed in statement `S3`. Thus, augmented code needs to check at run-time that there is no intersection between the references in `S3` and those in `S1` and/or `S2`. Of course, all other potential dependences caused by the references in `S1` and `S2` will have to be checked, because they cannot be analyzed at compile-time.

The technique we use to verify such reductions at run-time is the LRPD test, which is described in detail in [17], and with more implementation guidelines given in [18]. The main idea of the LRPD test with respect to reduction validation is to speculatively execute a loop in parallel (including the reduction operation) and subsequently test if the reduction operation was indeed a valid reduction, i.e., it tests if any data dependences occurred and if the references appearing in the reduction statements were accessed anywhere else in the loop. If the test passes, the reduction variables, which were privatized and accumulated on each processor during the speculative execution, are merged into the shared array after loop termination. If the test fails, the loop is re–executed in a safe manner, e.g., sequentially.

B. Reduction Algorithm Library

Reductions are associative recurrences, and they can be parallelized in several ways. Our library currently contains two types of methods: *direct update methods*, which update shared reduction variables during the parallel execution of the loop, and *private*

accumulation and global update methods, which accumulate in private storage during the parallel execution of the loop and update shared variables with each processor's contribution afterwards. Direct update methods include the classical *recursive doubling* [12, 13], *unordered critical sections* [19, 14], and *local write* [20]. Our library only includes *local write* because the others are not competitive for parallelizing reductions involving array variables. Private accumulation methods in our library include *replicated buffer* [12, 13, 21, 22], and two novel algorithms we have proposed: *replicated buffer with links* and *selective privatization* [7].

First, we describe a graphical representation for illustrating parallel reduction algorithms. Fig. 3 gives a sequential irregular reduction loop and its graphical representation. The loop has 9 iterations and accesses 5 data elements. Each iteration of the loop has 2 distinct reduction statements. In the graph, the squares represent data elements and the circles represent iterations of the reduction loop. Edges between the squares and the circles indicate data elements accessed by reduction statements in different iterations. In addition, we assume that 3 processors are used and static iteration scheduling is used whenever it is applicable.

1. Replicated Buffer (RepBuf)

To allow the original loop to execute as a `DOALL` (executing the iterations of the loop concurrently on multiple processors), RepBuf privatizes the reduction variables and accumulates partial results in private storage. After loop execution, the partial results are accumulated across processors and the corresponding shared array is updated. The parallel version of the reduction loop in Fig. 3 and the corresponding graphical representation are given in Fig. 4 and Fig. 5, respectively.

As shown in Fig. 4, the parallel loop applying RepBuf includes three fully parallel loops. The first loop initializes the replicated private array to zeros. The second

```
integer X(2,9)
real    A(5)
P=3, static iteration scheduling
do i = 1,9
   ......
   A(X(1,i))=A(X(1,i))+expr1
   A(X(2,i))=A(X(2,i))+expr2
enddo
```

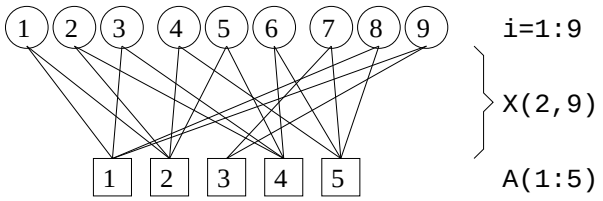


Fig. 3. An irregular reduction loop and its graphical representation

loop executes the original loop iterations in parallel. Each processor (thread) executes a part of the original loop iterations and the corresponding reduction operations operate on the processor's private array. Finally, the third loop updates the shared variable with all the processors' contributions.

The advantages of this technique are as below. First, it works well if the access pattern is dense, i.e., if most elements of the replicated arrays are indeed written during execution. Second, the method is simple to implement and no additional work is generated during loop execution. The final cross-processor update (reduction) is easy to implement (it is fully parallel) and its work increases linearly with the number of processors (more processors implies more private arrays). Third, individual reduction operation accessing the replicated reduction variable is very fast (i.e., via array access).

The disadvantage of RepBuf is that if the reference pattern is sparse or the average number of iterations referencing a reduction data element (which we call *degree of contention*) is low, then many elements may not be touched at all or only

```
1   DOALL p = 1, P
      pA(1:n,p) = 0

2   DOALL i = 1, M
      p = get_pid()
      ......
      pA( X(1,i),p ) = pA( X(1,i),p ) + expr1
      pA( X(2,i),p ) = pA( X(2,i),p ) + expr2

3   DOALL i = 1, N
      A(i) = A(i) + pA(i, 1:P)
```

Fig. 4. Replicated buffer algorithm (pseudo-code)

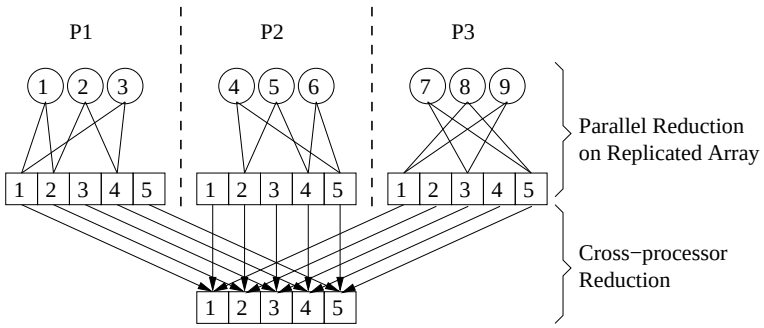


Fig. 5. Replicated buffer algorithm (graphical representation)

by a small subset of the processors. Thus, the maximally allocated private space is not efficiently used and can create problems associated with referencing a large data structure during the parallel loop execution (e.g., poor spatial cache locality, page misses, TLB misses, etc.). Moreover, the final cross–processor reduction performs a lot of unnecessary work, i.e., instead of being proportional to the actual number of distinct memory references on each processor, the total work of the phase is proportional to the dimension of the private array; this makes the scheme not scalable with the number of processors.

2. Replicated Buffer With Links (RepLink)

To avoid traversing the not-used (though allocated) private elements at the cross-processor reduction phase of RepBuf algorithm (the third `DOALL` loop in Fig. 4), we have developed RepLink. This algorithm is very similar to RepBuf and is in essence a sparse storage scheme that can be processed easily in parallel. Here we also allocate, on all processors, private arrays conformable to the original shared reduction array. However, we also provide additional links to the private data. The links are to connect all the used "private copies" of original shared reduction elements. After the parallel loop finishes, the cross-processor merging operation can be performed only along the chain of links to visit just the used private elements, thus reducing the memory footprint and the number of remote misses to those absolutely necessary. This is useful when the *degree of contention* is low and only a few processors need to participate in the merging operation. The scheme is illustrated in Fig. 6.

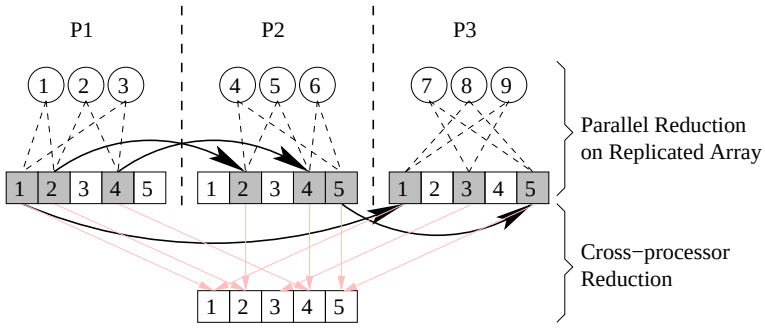


Fig. 6. Replicated buffer with links algorithm (graphical representation)

In this algorithm, the links are established either during an inspector phase or during the merging phase of RepBuf, when the reduction data access pattern (determined by the contents of index arrays) changes. That is, when we detect a pattern change, we update the linked list structure. The disadvantage of this algorithm is that, for the loop execution instance where reduction data access pattern

changes, it executes a setup phase which has extra work and requires more memory to be allocated.

3. Selective Privatization (SelPriv)

To reduce the memory pressure associated with allocating large, sparsely-used replicated arrays, we wish to replicate only array elements that are referenced by multiple processors. The main idea of SelPriv is to first determine which elements are referenced on multiple processors and then allocate for them (and only for them) private units. By excluding unused private elements from the fully replicated array, SelPriv maintains a dense private space where all allocated private elements are used (except for elements allocated to eliminating unnecessary cache coherence traffics). This releases memory pressure and increases spatial locality in terms of cache usage and paging activities. The technique is illustrated in Fig. 7.

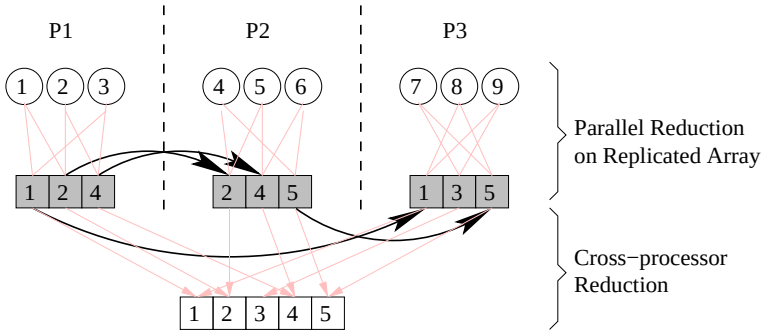


Fig. 7. Selective privatization algorithm (graphical representation)

In SelPriv, the private space does not align with the original data array. The straightforward implementation would be to use an indirection array to direct the references of the original data elements to the selectively privatized elements. To avoid introducing this additional level of indirection in the reduction operations, we make a copy of the index array (most applications that we have studied reference their reduction arrays through subscript arrays) and modify the appropriate elements of

the copied index array to point to the selectively privatized data elements. In this manner, the execution of the parallel loop is not slowed down and the memory usage is reduced. In the final merging phase, only the privatized portions of the array will be merged, further improving performance. It should be noted, though, that each reference during the merging phase is more expensive because the private storage is not conformable and the links among the private elements of the same shared element are traversed.

An additional optimization is that we assume the shared data is uniformly distributed across processors and therefore we only privatize the "remote" data. This way one processor can always write directly to the original shared array without any contention. The benefits may seem small but if there are only 2 sharers the final merge time is significantly reduced, i.e., instead of merging cross-processor, the shared element is simply updated with one private contribution.

A potential disadvantage of SelPriv is the existence of the copy of the index array. The size of this index array is proportional to the number of referencs, and is not necessarily proportional to the distinct number of referencs. So the setup phase needs to traverse the array and modify all elements where the corresponding data elements need to be privatized. This traversal can be quite expensive if it cannot be amortized across many invocations of the loop. An alternative is to use the original subscript array and modify it where elements need to be privatized; each processor then points to its own (private) space. Because the subscript array may be also used outside the scope of the loop, we need to save the original values before modifications are done and restore them after loop end. In our implementation, we used an additional copy of the index array.

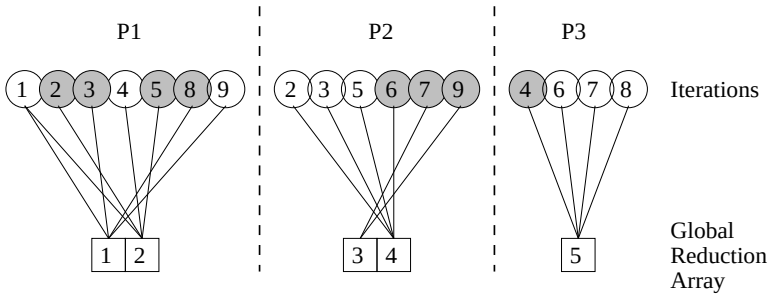


Fig. 8. Local write algorithm (graphical representation) – gray circles represent replicated iterations

4. Local Write (LocalWr)

LocalWr uses a variation of the "owner–computes" method to parallelize irregular codes [20]. The reference pattern is first collected in an inspector loop [23] and is followed by a partitioning of the iteration space based on the "owner-computes" rule. Memory locations referenced across processors have their iterations replicated so that the reduction operations access data local to each processor. The technique is illustrated in Fig. 8.

This algorithm works well when contention across iterations, or a measure called *connectivity* by its authors (the ratio of number of iterations and number of distinct elements referenced), is low and thus work (iteration) replication can be kept to a minimum. Because this method eliminates any contention through iteration replication, no critical sections are necessary. Another significant advantage of LocalWr is the fact that it maximizes locality (via "owner-computes"), and thus its performance is maximized in distributed memory environments where latency and message passing are quite expensive.

The disadvantage of this scheme is that the inspector phase can be quite expensive because it traverses and collects all references to the reduction array occurring in the loop. The resulting data structure may be quite large and its analysis expensive.

Another potential drawback of the method is that for reference patterns with high *connectivity*, work duplication may be large.

5. Qualitative Comparison

In Table I, we present a qualitative comparison of the algorithms discussed above.

TABLE I
QUALITATIVE COMPARISON OF PARALLEL REDUCTION ALGORITHMS

Issues	RepBuf	RepLink	SelPriv	LocalWr
Good when contention	High	Low	Low	Low
Locality	Poor	Poor	Good	Good
Need schedule-reuse	No	Yes	Yes	Yes
Extra Work	No	No	No	Yes
Extra Space	`NxP`	`NxP`	`NxP+M`	`MxP`

`M` is the number of iterations; `N` is the size of the reduction array; `P` is the number of processors.

In the table, the contention of a reduction is the average number of iterations (# processors when running in parallel) referencing the same element. When contention is low, many unused replicated elements in RepBuf are accumulated across processors, while other algorithms only pass useful data. SelPriv works on a compacted data space and therefore potentially has good spatial locality. In LocalWr, each processor works on a specific portion of the data array and therefore potentially has good temporal and spatial locality. With respect to overhead, RepLink, SelPriv and LocalWr all have auxiliary data structures that depend on the access pattern and must be updated when it changes. Their overhead costs can be reduced with *schedule reuse* [24].

C. Preliminary Experimental Results

In this section we show performance data (speedups) of reduction loops from several codes that have been parallelized using known and newly introduced reduction techniques and have been executed with several different input sets.

1. Experimental Setup

The experimental setup for our speedup measurement consisted of a 16 processor HP-V class system with 4Gb memory and 4Mb cache per processor, running the HPUX11 operating system. It is a directory based cache coherent shared memory machine with uniform memory accesses (UMA). Due to the limited size of our input sets and constraints on our single user time allocation, we used 8 processors.

TABLE II
APPLICATIONS AND LOOPS TO EVALUATE PARALLEL REDUCTION ALGORITHMS

Program	**Description**	**Lang.**	**Lines**	**Source**	**#inp**
IRREG	CFD kernel using FE methods	F77	223	[20]	4
NBF	M.D. kernel (from GROMOS)	F77	116	[20], [25]	4
MOLDYN	synthetic M.D. program	C	688	[20], [26]	4
CHARMM	M.D. kernel (from CHARMM)	F77	277	[27]	3
SPARK98	unstructured FE simulation	C	1513	SPEC'2000	2
SPICE	circuit simulation	F77	18912	SPEC'92	4
FMA3D	3D FE method for solids	F90	60122	SPEC'2000	1

Program	**Loop**	**Coverage**
IRREG	apply effects of nodes (do 100) – traverse grid edges	$\sim 90\%$
NBF	non-bounded force calc. (do 50) – traverse neighbor list	$\sim 90\%$
MOLDYN	non-bounded force calc. – traverse interaction list	$\sim 70\%$
CHARMM	non-bounded force calc. – traverse neighbor list	$\sim 80\%$
SPARK98	smvp loop – Symmetric Matrix-Vector Product	$\sim 70\%$
SPICE	bjt loop – traverse BJT devices, update circuit nodes	11–45%
FMA3D	Scatter_Element_Nodal_Forces_Platq loop	$\sim 30\%$

We have augmented the Polaris optimizing compiler [28] to generate different

versions of parallel reduction loops implementing 4 parallel reduction algorithms, which are *replicated buffer*, *replicated buffer with links*, *selective privatization*, and *local write*. We have chosen, from a variety of scientific domains, 7 known programs and kernels, which are specified in Table II and briefly described below. For most of the programs, we have chosen or specified multiple inputs to explore the input-dependent nature of irregular programs. While specifying the inputs, we have tried, where possible, to use data sets that exercise the entire memory hierarchy of our parallel machine in order to get the performance data of "real-life" applications.

IRREG is an iterative PDE solver used in CFD applications. It uses an unstructured mesh to model physical structures. The code uses nodes and edges of a graph to represent its mesh. The reduction loop applies the force associated with each edge to its two end points. After evaluating the forces at each node the program performs an irregular reduction to update them with the new values. The different input sets have almost the same amount of work (# iterations).

NBF is a kernel, computing non-bounded forces among molecules, reduced from the GROMOS molecular dynamics benchmark [29]. It is typical of two dimensional N-body simulations in that the code maintains a continuously updated list of the neighbors with which it interacts. At every time step, forces are evaluated at each node and applied through a reduction operation across the whole data structure of the program.

MOLDYN is a synthetic program abstracted from the molecular dynamics code CHARMM [3]. While it is similar to NBF in the sense that the molecules interact only with nodes falling in a cutoff distance, the accesses in MOLDYN are different from NBF. MOLDYN maintains the neighborhood information in a single interaction list that is updated with a user-specified frequency.

CHARMM is also a kernel reduced from CHARMM [3], which has been used

by the CHAOS group at the University of Maryland [27]. Similar to NBF, this kernel works on the neighbor list but it works in a three dimensional domain (and thus it updates forces in 3 orthogonal directions stored in 3 arrays). While IRREG, NBF, and MOLDYN generate their meshes in the program with some random processes and few controlling parameters read in from the input, CHARMM reads the mesh from an input file.

SPARK98 is a collection of 10 sparse kernels developed by David O'Hallaron at CMU [30]. The sparse matrices are induced from a pair of three-dimensional unstructured finite element simulations of earthquake ground motion in the San Fernando Valley. Each kernel is a program/mesh pair. There are 5 C programs (`smv, lmv, rmv, mmv, hmv`) and 2 relatively small input data sets (`sf10` and `sf5`). We have chosen the `rmv` kernel, which computes irregular reductions (same as `equake` in the SPEC CPU'2000 benchmark suite). The meshes determine both the size and non-zero structure of symmetric sparse matrices used. The irregular reduction loop does matrix vector multiplication. Since the program only stores the upper triangle of the matrices, although the computation (updating the resulting vectors) associated with the upper triangle is regular and loop-independent, the computation associated with the lower triangle consists of irregular updates. This code (written in `C`) has been transformed by hand because our compiler infrastructure can handle only Fortran code.

SPICE 2G6 is a well-known circuit simulation code written in an older Fortran style. Its main feature is that it does its own memory management inside a statically allocated large array named `VALUE`. Therefore all references to arrays are through indirection, which makes almost any compiler analysis impossible. We have transformed the code for efficient parallel execution using the run-time techniques described in [31, 32]. The main reduction loop in subroutine `BJT` evaluates the device

model and updates the mesh nodes of the circuit (reduction). The program iterates to a fixed point solving a linear system and then re-evaluates the device model for the newly found values. The BJT device model evaluation loop takes between 11% and 45% of the total sequential execution time depending on the complexity of the devices and circuits being simulated. There are 28 distinct reduction statements for each iteration.

FMA3D is a 3D inelastic, transient dynamic response simulation code based on the finite element method. We choose one reduction loop in the `scatter element nodal force platq` subroutine. The loop we choose updates 8 nodes of each "8-node hexahedral continuum element" and therefore there are 8 distinct index (subscript) expressions and three different reductions sharing the same index.

2. Results and Discussion

Fig. 9 shows a quantitative evaluation of the parallel reduction algorithms. All speedup graphs account to the reduction loops, including all the time associated with setting up the auxiliary data structures used by the various reduction schemes. Each bar group within the graphs corresponds to a specific program-input case. The inputs are labeled with data size (the number of elements of reduction data arrays) and a parameter (*connectivity*, which is defined as the ratio of number of iterations of the reduction loop and the data size). In the graph titled "SPARK98 & FMA3D," abbreviations of the two applications, **S** for SPARK98 and **F** for FMA3D, are added to the labels of the bar groups. The legends of the graphs correspond to 4 parallel reduction algorithms: *replicated buffer* (RepBuf), *replicated buffer with links* (RepLink), *selective privatization* (SelPriv), and *local write* (LocalWr).

For most of the program–input cases, LocalWr does not perform as well as the other algorithms. This is because LocalWr is designed for systems with distributed

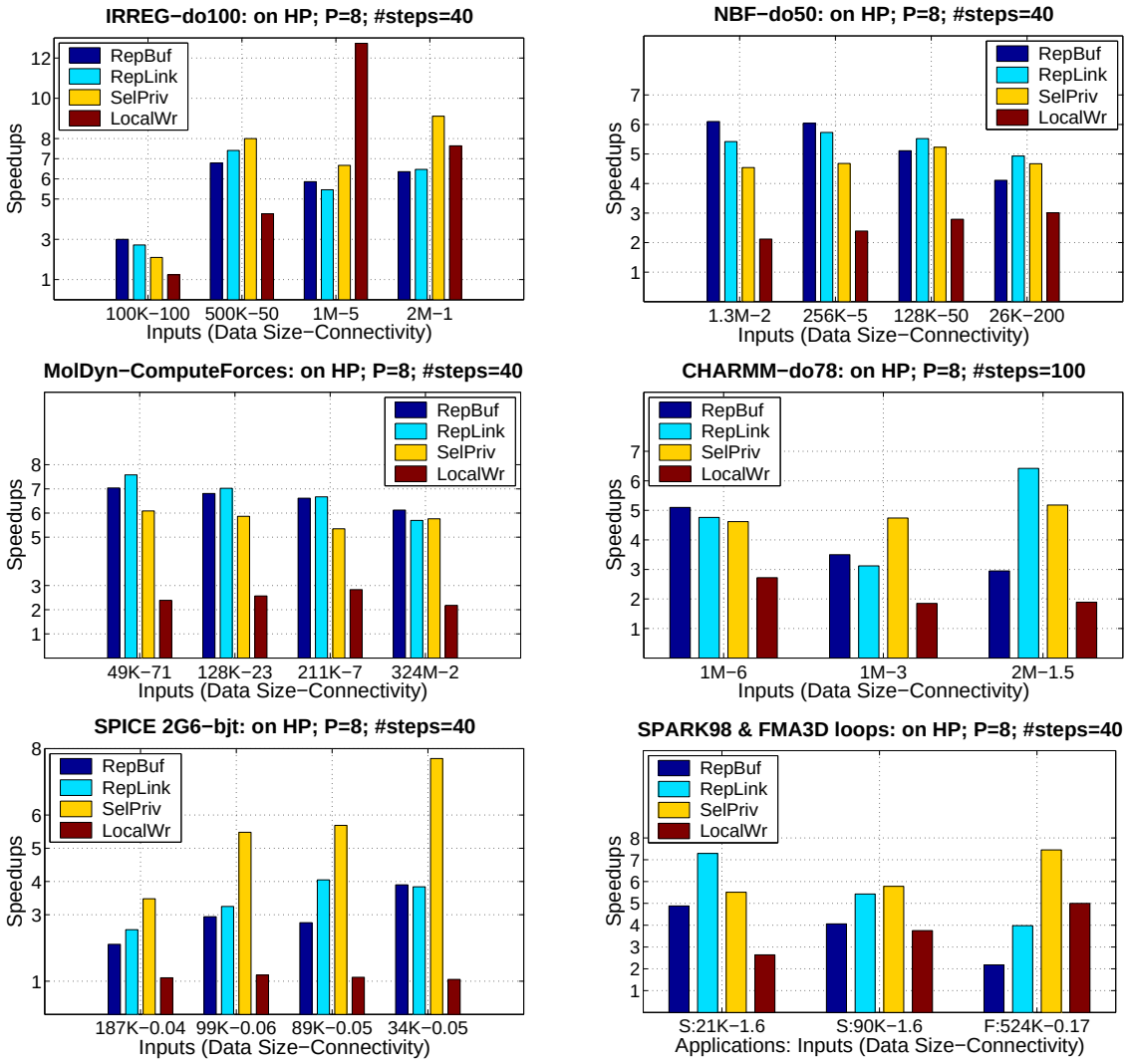


Fig. 9. Performance of parallel reduction algorithms

memory. One exception is an input of IRREG, which works well with LocalWr because it manages to increase locality and break up the working set among processors. Thus, in spite of some code replication (about 80% more work due to iteration replications), the code gets a superlinear speedup for an input with a certain data size.

Overall, we emphasize two observations from the experimental results. First, our newly proposed parallel reduction algorithms, i.e., RepLink and SelPriv, provide competitive performance for most of the program–input cases and perform the best

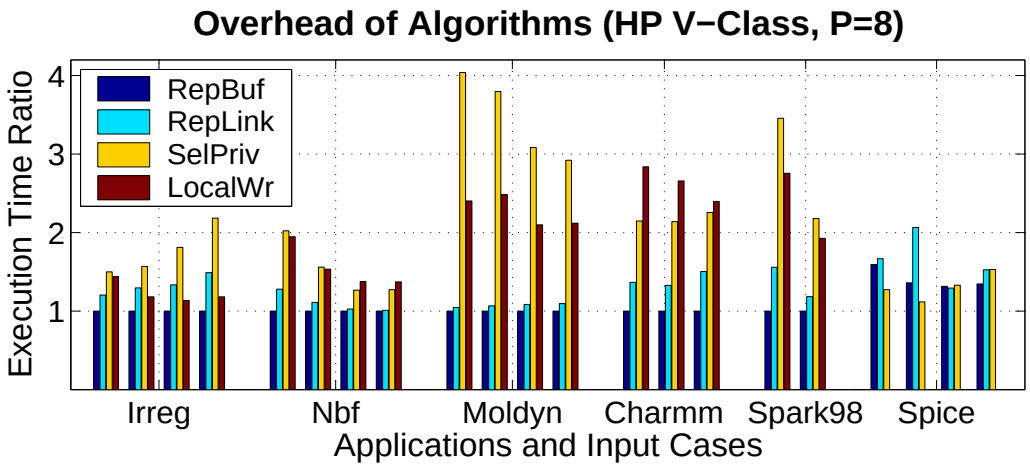


Fig. 10. Overhead of parallel reduction algorithms

for many cases. Secondly, we do not find any parallel reduction algorithm that can perform the best for all the cases. This motivated us to derive performance models to select the best performing parallel algorithms for given program–input cases (the corresponding techniques are described in the next chapter).

Fig. 10 shows the overhead associated with the various parallel reduction algorithms. We set the input parameters to let the reduction pattern change every instance (for kernels) or to consider only the instances where patterns change (for real applications). The obtained speedups are normalized to that of the RepBuf algorithm for each loop-input case, since it is the simplest one and many production compilers implement the algorithm (specified as default algorithm by OpenMP standard [33]). The results in Fig. 10 show that SelPriv has a relatively large overhead due to the restructuring of the private space. LocalWr has a large overhead due to the inspector loop and work duplication. For SPICE, since all the data accesses are in the same array `Value`, we have inserted an inspector to identify the range of the array where the irregular reductions operate. Because this work has to also be done for RepBuf, the relative overhead of other algorithms (compared to that of

RepBuf) is low comparing to the overhead of other loop-input cases.

D. Summary

In this chapter we discussed in general how to identify reduction operation, a frequent associative (most often also commutitive) operation that can be parallelized in many ways. We also presented several parallel reduction algorithms, including both previously developed algorithms and novel algorithms, which we proposed for efficiently parallelizing irregular and/or sparse reductions. We presented preliminary experimental results that show that the performance of our novel techniques provide competitive performance compared to other known techniques for many applications' dynamic execution instances (application and input combinations).

The first conclusion of this chapter is that our novel algorithms have added a valuable complement to the family of parallel reduction algorithms. Secondly, the algorithms discussed in this chapter consist of a library of parallel reduction algorithms which are specialized for different classes of access behaviors. In addition, as long as we select the right algorithm for any given dynamic execution instance, the overall performance of irregular reductions can be improved significantly.

In the following chapter, we describe how we specialize our proposed *adaptive algorithm selection* framework to select the best parallel reduction algorithms for given execution instances of irregular reduction loops.

CHAPTER IV

ADAPTIVE REDUCTION SELECTION

A. Introduction

As demonstrated in the previous chapter, not all parallel reduction algorithms and/or implementations are equally suited as substitutes for the original sequential algorithm. Each dynamic data access pattern of reductions, though irregular, has its own characteristics and will best be parallelized with an appropriately tailored algorithm or a customized implementation of an existing algorithm.

In this chapter, we discuss how we apply our *adaptive algorithm selection* framework (outlined in Chapter II), to adapt reduction parallelizations to the actual reference pattern executed by a reduction loop, i.e., to the particular input data and dynamic phase of a program. More precisely, we dynamically characterize irregular reductions' reference patterns and choose the most appropriate method for parallelizing it. We use the library of parallel reduction algorithms composed by the algorithms discussed previously.

The matching of algorithm to reference pattern is performed using a synthetic experiment approach. First we characterize the data access patterns and reduction loops with a set of parameters (identified manually) whose values are computed statically and measured dynamically. Then we automatically generate prediction models (mapping from parameters to the best parallel algorithms) from synthetic experimental results running a parameterized synthetic reduction loop (parallelized with various parallel reduction algorithms) with a set of parameter values generated for a factorial experiment. So far, we have applied both multi-regression and decision tree learning to generate the models from the synthetic experimental results. The

generated prediction models can be applied for different irregular reduction execution instances with low overhead. All processes for establishing such prediction models and their use in a real applications are automated.

The rest of this chapter is organized as follows. In Section B, we introduce a small number of high-level parameters we have identified, which can be used to characterize reduction access patterns and discriminate the different parallel reduction algorithms. In Section C, we describe a systematic process through which input sensitive predictive models can be built off-line and used dynamically to select from a particular list of functionally equivalent algorithms.

B. High-Level Parameters

In this section, we describe the parameters we have chosen to characterize reduction operations. Ideally, they should require little overhead to measure and they should enable us to select the best parallel reduction algorithm from our library for each reduction instance in the program. We first define the identified parameters, and then present a summary of the decoupled effects of the parameters on the performance of the parallel reduction algorithms to illustrate the effectiveness of the chosen parameters.

1. The Parameters

Below, we enumerate the parameters in no specific order.

N is the *number of data elements* involved in the reduction (often the size of the reduction array). It strongly influences the loop's working set size, which may impact performance depending on the machine's cache size, etc. In some applications, several reduction arrays have exactly the same access pattern; here **N** includes the

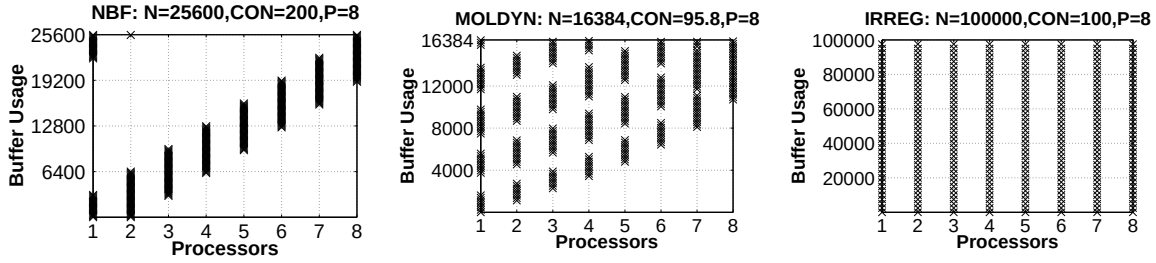


Fig. 11. Memory access patterns of replicated buffer algorithm

data elements for all arrays.

CON, the *Connectivity* of a loop, is the ratio between the number of iterations of the loop and **N**. This parameter is equivalent to the parameter defined by Han and Tseng in [20]; there, the underlying data structures (corresponding to the irregular reductions) represent graphs $G = (V, E)$ and *Connectivity* is defined as $|V|/|E|$. Generally, the higher the connectivity, the higher the ratio of computation to communication, i.e., if the connectivity is high, a small number of elements will be referenced by many iterations.

MOB, the *Mobility* per iteration of a loop, is the number of distinct subscripts of reductions in an iteration. For the *local write* algorithm, the effect of high iteration Mobility (actually lack of mobility) is a high degree of iteration replication. MOB is a parameter that can be measured easily at compile-time.

OTH, represents the *Other (non-reduction) work* in an iteration. If **OTH** is high, a penalty will be paid for replicating iterations. To measure this parameter, we instrument a parallel loop transformed for the *replicated buffer* algorithm using light-weight timers ($\sim$ 100 clock cycles).

SP, *Sparsity*, is the ratio of the total number of referenced private elements and total allocated space on all processors using the *replicated buffer* algorithm ($P \times N$). Intuitively, **SP** indicates whether *replicated buffer* is efficient.

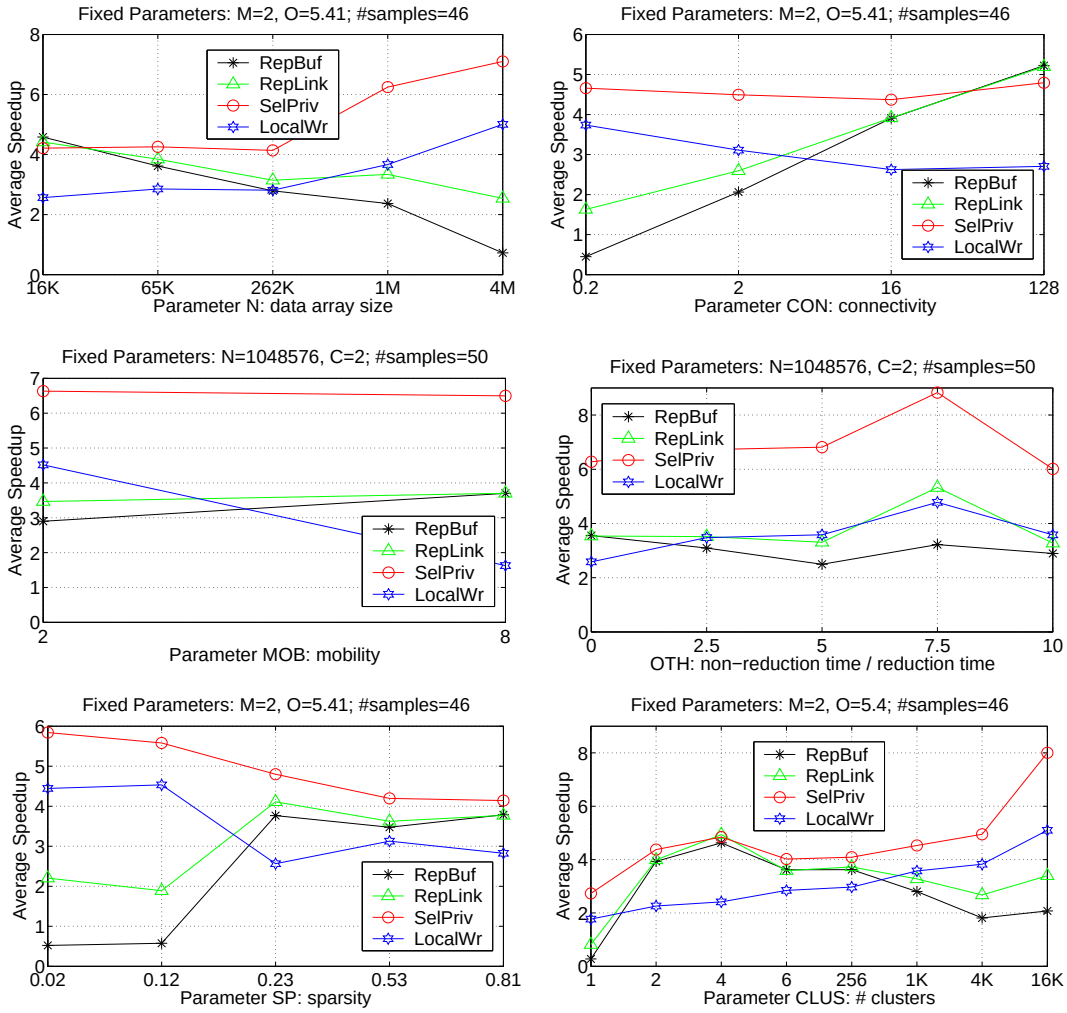


Fig. 12. Decoupled effect of parameters (on a HP V-Class, with #processors=8)

CLUS, the *Number of Clusters*, reflects spatial locality and measures whether the used private elements in the *replicated buffer* are scattered or clustered on each processor. Fig. 11 shows three memory access patterns (space usage pattern) that are executed by applying *replicated buffer* algorithm. The patterns can be classified as *clustered*, *partially-clustered*, and *scattered*. Currently, **SP** and **CLUS** are measured by instrumenting parallel reduction loops using the *replicated buffer* algorithm, and the overhead is proportional to the number of used private elements. **CLUS** measures the average number of clusters of the used private elements on each processor.

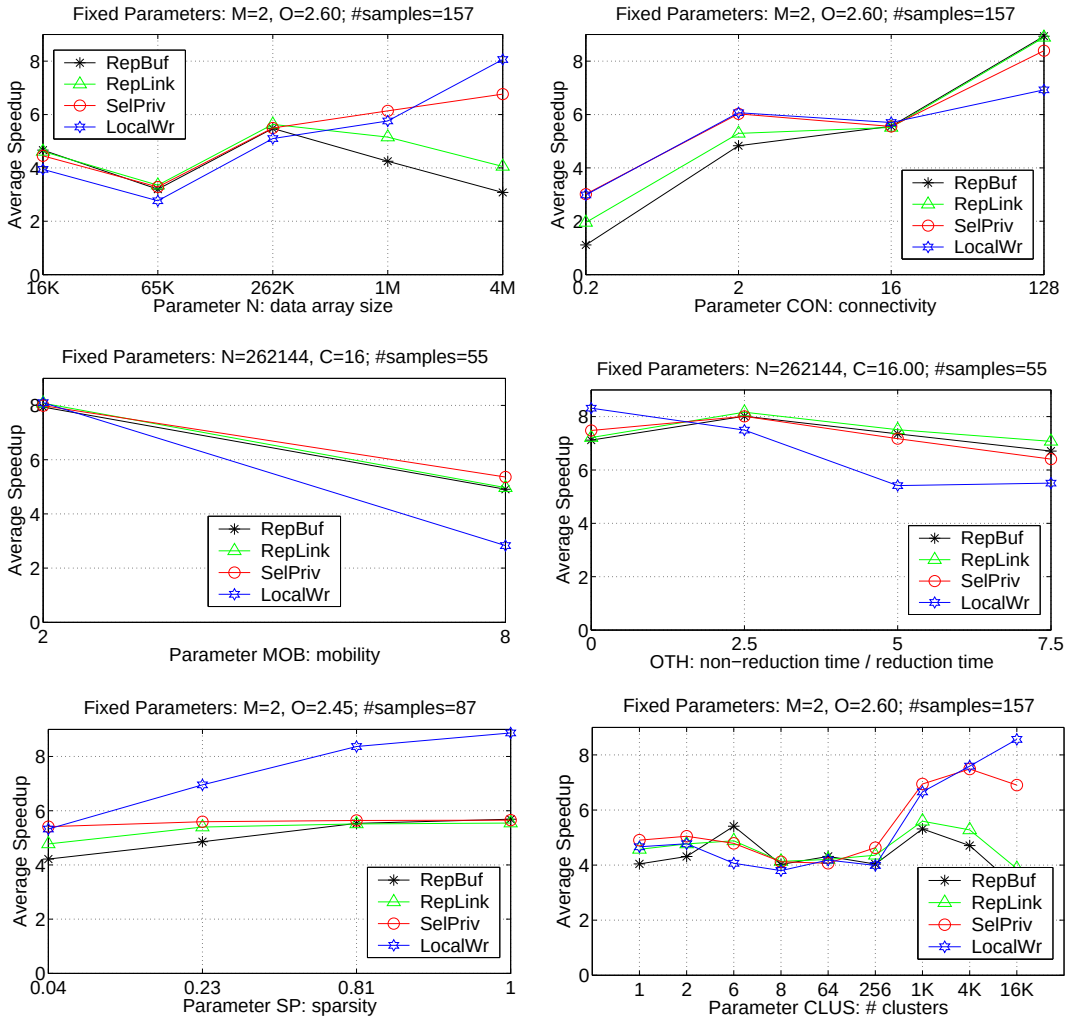


Fig. 13. Decoupled effect of parameters (on an IBM Regatta p690, with #processors=16)

2. Decoupled Effects of the Parameters

We investigated the decoupled effects of the parameters on the performance of the parallel reduction algorithms. Although the decoupling is not realistic, it is useful for discovering qualitative trends.

In Fig. 12 and Fig. 13, we show the decoupled effect of a single parameter on the speedups of the parallel reduction algorithms in our library. The experiments were done on a HP V-Class machine (UMA) and an IBM p690 machine (NUMA).

The detailed specifications of the two systems are given in Table V of Chapter V. The graphs are based on instantiating parallel versions of a parameterized synthetic reduction loop with factorial combinations on selected parameter values (see details in Section C.1). The vertical axis corresponds to the average speedups on a set of samples. The title indicates the fixed parameters and the number of samples on which the graph is based. In the titles, `M` is **MOB**; `C` is **CON**; and `O` is **OTH**. Table III summarizes the trends illustrated in Fig. 12 and Fig. 13. The trends for RepLink are similar to RepBuf and are not listed in the table.

TABLE III
SUMMARY OF THE DECOUPLED EFFECTS

Parameters	RepBuf	SelPriv	LocalWr
N	↗	↑	↑
CON	↑	↗	–
MOB	↑	↘	↓
OTH	↑	↑	↗
SP	↗	↓	↘
CLUS	↗	↑	–

↑: positive effect; ↓: negative effect; ↗: little positive effect; ↘: little negative effect; –: no effect.

The effect of **N** is straightforward, compared to the sequential reduction loop, SelPriv and LocalWr have much smaller data sets on each processor and therefore their speedups increase with **N**. **CON** is inversely correlated with inter-processor communication; hence, larger **CON** values indicates better scaling for the data replication-based algorithms (RepBuf and SelPriv), which have two reduction loops, one accumulating in private space and one accumulating cross-processor shared data.

Large **MOB** values (indicating a large number of references to index arrays) may imply poor performance for SelPriv, because accesses to the reduction array must

access both the original and the modified index arrays; for LocalWr, large **MOB** values often result in high iteration replication. Large values of **OTH** indicate good performance for RepBuf and SelPriv because the first private accumulation loop has a larger iteration body; since LocalWr replicates non-reduction work, it will not benefit as much.

Low **SP** is good for SelPriv and also for LocalWr, since it correlates with low contention and hence low iteration replication. For large **CLUS**, because SelPriv compacts the sparsely used data space, this algorithm achieves better speedups than LocalWr and RepBuf, which work on original (non-compacted) data or in private spaces conformable to the original data.

C. Adaptive Reduction Selection

In this section we elaborate on the *setup* and *dynamic selection* phases of our *adaptive algorithm selection* framework (see Chapter II). The *setup* phase is executed only once, during machine installation, and generates a map between points in the universe of all inputs (memory reference patterns characterized with the previously defined parameters) and their corresponding best suited reduction algorithms. The *dynamic selection* phase is executed every time a targeted reduction is encountered. It uses the map built during the setup phase, a parameter collection mechanism to characterize the memory references, and an interpolation function (the actual algorithm selector) to find the best suited algorithm in the library.

We then explain how these methods have to change in order to optimize dynamic programs, i.e., codes change their characteristics during execution.

1. Setup Phase

We now outline the design of the initial map between a set of synthetically generated parameter values and the corresponding performance ranking of the various parallelization algorithms available in our library. The overall *setup* phase, is illustrated in Fig. 14.

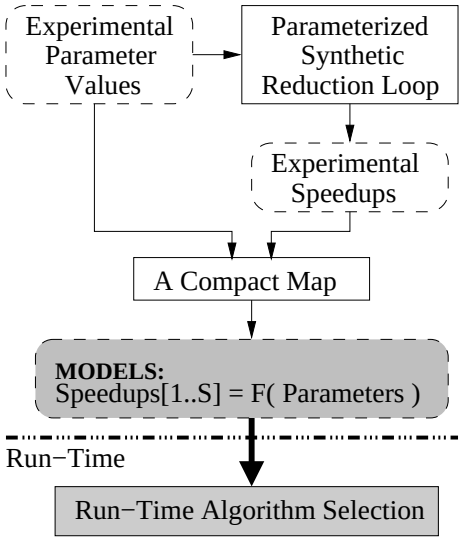


Fig. 14. Setup phase for adaptive selection of reduction algorithms

The domain of values that can be taken by the input parameters is explored by setting up a factorial experiment [34]. Specifically, we choose several values (typical of realistic reduction loops) for each parameter and generate a set of experiments from all combinations of the chosen values. The chosen parameter values for our reduction experiment are shown in Table IV.

To measure the performance of different reduction patterns, we have created a synthetic reduction loop. The structure of the loop is shown in Fig 15, with C-like pseudo-code. The non–reduction work and reductions have been grouped in two loop nests. Because sometimes the native compiler cannot unroll the inner loop nests in the same manner for different versions of the loop, we have performed this transformation

TABLE IV
PARAMETER VALUES FOR THE FACTORIAL EXPERIMENT

Parameters	Values				
N (array size)	16384	65536	262144	1048576	4194304
Connectivity	0.2	2	16	128	
Mobility	2	8			
Other Work	1	4			
Sparsity	0.02	0.2	0.45	0.75	0.99
# clusters	1	4	20		

with an additional pre–processing step. The dynamic pattern depends strictly on the `index` array, which is generated automatically via a randomized process satisfying the requirement specified by the parameters **SP** and **CLUS**. **OTH** is a dynamically measured parameter that represents the ratio between the time spent performing reductions and the rest of the loop.

```
FOR j = 1 to N*CON
  FOR i = 1 to OTH      /* non-reductions */
    memory read & scalar computation;
  FOR i = 1 to MOB          /* reductions */
    data[ index[j][i] ] += expr;
```

Fig. 15. The parameterized synthetic reduction loop

The performance ranking of the parallel reduction algorithms in our library is accomplished by simply executing them all for each parameter combination (i.e., synthetically generated access pattern) and measuring their actual speedups, as shown in Fig. 14.

The end result is a compact map from parameter values to performance (speedups) of candidate parallel algorithms. From this map we can now create the prediction code (the models) that can then be used by an application at run-time. We applied two methods, *general linear regression* and *decision tree learning*. In this disserta-

tion, we will describe only the modeling process using *general linear regression*. For description and experimental results using *decision tree learning* method, please refer to our previous publication [11].

As illustrated in Fig. 14, the generated models are polynomial functions from the parameter values to the speedups of the parallel algorithms. We follow a standard "term selection" process that automatically selects polynomial terms from a specified pool [35]. We have specified a maximum model as $F = (\lg N + \lg C + MOB + OTH + \lg S + \lg L + 1)^3$, and a minimal model as $F = \lg N + \lg C + MOB + OTH + \lg S + \lg L + 1$. For brevity, C is **CON**, S is **SP**, and L is **CLUS**. Then, from the minimal model, relevant terms are randomly selected from the 84 terms of the maximum model.

The samples are first separated into training data and testing data. When adding a term into the model, the training data are used to least square fit the coefficients and the fitted model is evaluated using the testing data. If the test error is higher than that of the model before (including the newly added term), the term will be dropped. Though this sequential term selection process will not give us the optimal model, it is fast and the automatically generated models have produced good results when used to predict relative performance of the parallel algorithms on real reduction loops. Here, the order of the model, 3, is chosen mainly due to practical reasons, such as generating less experiment samples with less synthetic experimentation time and avoiding term explosion. For parameter **MOB**, we have chosen 2 values for the experiment and we have excluded the terms containing a non-linear **MOB** factor from the maximum model. For both **OTH** and **CLUS**, though we specified few values, the ones used in the map are measured and computed from the generated `index` array.

The final polynomial models contain about 30 terms. The corresponding C library routines are generated automatically to evaluate the polynomial $F()$ for each algorithm at run-time.

2. Dynamic Selection Phase

During the dynamic selection phase, to avoid executing the parameter collection and algorithm selection steps for every time a reduction loop is invoked, we use a form of memoization, *decision reuse*, which detects if the inputs to our selector function have changed. When a new instance of a reduction is encountered and the input parameters have not changed from the previous execution instance, we directly reuse the previously selected algorithm, thus saving run-time selection overhead. This is accomplished with standard compiler technology, i.e., the compiler instruments two version loops for each parallel algorithm (illustrated in Fig. 16).

During this phase, the pre-generated model evaluation routines are called to estimate speedups of all the algorithms, rank them, and select the best one. The evaluation of the polynomial models is fast as each of the final models contains only about 30 terms.

We utilized a research compiler – Polaris [28] – to identify irregular reduction loops and generate code that performs run-time adaptive reduction selection. The compiler is capable of extracting the condensed access descriptors of a loop (similar to an inspector loop) and, where it is not possible, uses our run-time parallelization pass [17, 36, 31] to collect data during actual loop execution. The compiler inserts calls to the run-time library that computes the various parameters we are interested in.

The compiler also instruments multiple versions of parallel reduction loops that implements the candidate parallelization algorithms. For each algorithm, two versions of the loop are instrumented. The code construct for one parallel algorithm is illustrated in Fig. 16. In the diagram, **SCH_adapt** is the *adaptive* version, which carries out reduction operations, traces the access pattern of reductions, and updates

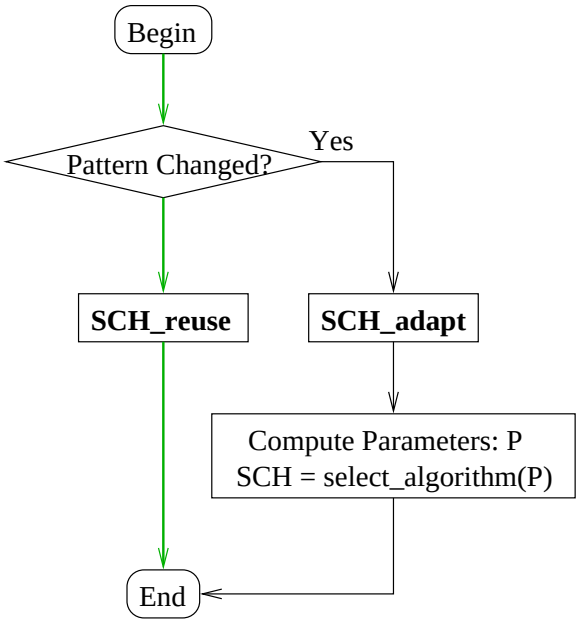


Fig. 16. Adaptive reduction parallelization at run-time

auxiliary data structures whenever necessary. **SCH_reuse** is the *reusing* version, which is better optimized (only carrying out useful computations) and is called when the pattern is not changing.

We used a technique we named *Global Schedule Reuse Control* to reuse the information across execution instances of the loop. Instead of proving that addresses do not change from one instance of the loop to the following one(s), we check, at run-time, for any potential change of addresses in the global context, e.g., we instrument checks at all places that update the index array. The checking sites are inserted at the possible outer–most level of the loop nests by utilizing dominance relations.

3. Selection Reuse for Dynamic Programs

In the previous section we described a systematic process to generate prediction models that can recommend the best parallel reduction algorithm by collecting a set of static and dynamic parameters. We mentioned that if the memory characterization

parameters do not change we can reuse our decision and thus reduce run-time overhead. This may be of value if the compiler can automatically prove statically that the reference pattern does not change. If, however this is not possible, which is often the case for irregular dynamic codes, we have to perform a selection for each reduction loop instance. In this section we show how to reduce this overhead by evaluating a trade-off between the run-time overhead of selection and the benefit of finding a better algorithm.

To better describe the run-time behavior of a dynamic program, we introduce the notion of *dynamic phase*. For a loop containing irregular reductions, a *phase* is composed of all the contiguous execution instances for which the pattern does not change. For instance, assume that the access pattern of the irregular reductions changes at instance t and the next change of pattern is at instance $t^{'}$; the loop instances in $[t, t^{'} - 1]$ form a *dynamic phase*. We can therefore group the execution instances of irregular reduction loops into *dynamic phases*.

We define the *reusability* of a phase as the number of dynamic instances of the reduction loop in that phase. It intuitively gives the number of times a loop can be considered invariant and thus does not need a new adaptive algorithm selection. That is, the overhead of selecting a better algorithm at run-time can be amortized over *reusability* instantiations.

Dynamic irregular programs can present a number of phases each with their own *reusability*. However, it may be that even in the case of a phase change, no new recommendation (for a better algorithm) can be made. This is because simulations change their characteristics slowly.

In the following we will describe how we include *reusability* in the previously developed models to predict phase-wise performance, and thus obtain a better overall performance. This model will be employed at run-time to decide when it is worth

changing the algorithm.

First, we formulate the phase-wise speedup of a parallel algorithm as

$$\frac{R.T_{seq}}{(R-1).T_{par} + (1+O).T_{par}}$$

Then the best algorithm would be the one that has the smallest value of

$$\frac{R+O}{Speedup}$$

In the above formulas, R is *reusability*, O is the ratio of the set-up phase overhead (in time) and the parallel execution time of one instance of the parallel loop applying the considered algorithm, and *Speedup* is the speedup (relative to sequential execution) of the considered algorithm excluding the set-up phase overhead. T_{par} and T_{seq} are the execution times of the parallel and sequential loop, respectively, which are solely used to derive the latter formula.

Using the same off-line experiment process described in previous sections, we generate models to compute the O (setup overhead ratio). Together with the predicted speedup (excluding the setup overhead), we are able to evaluate $\frac{R+O}{Speedup}$ at run-time and select the best scheme for a dynamic execution phase.

Here, *reusability*, the number of times a scheme can be reused, is evaluated using the known compiler technique "schedule reuse" [24], which can find the last definition point of the memory addresses (similar to loop invariant hoisting [37]).

The higher the redefinition point in the code's loop-nest hierarchy, the higher the *reusability*. If the redefinition of the addresses used in the reduction loop is done outside a loop nest containing the reduction loop, then *reusability* is equal to the number of times the reduction loop is invoked, i.e., the product of the iteration counts of the loop nest. If the inner loop bounds cannot be evaluated statically, then it can be measured at run-time.

If the compiler cannot perform the loop invariant hoisting (for various reasons), we can flag the point where assignments to the addresses are made (i.e., detect changes). This information can be used dynamically to flag the need for a new algorithm selection. Furthermore, *reusability* can be collected dynamically and a statistical prediction scheme can be employed; initially, we assume that *reusability* is high because memory reference patterns change slowly in many physical simulations.

In our experiments we apply the schedule reuse technique as well as the dynamic flagging of address redefinition events. For the codes that are statically hard to analyze, we profilie and use statistical prediction (running averages) with very accurate results.

To further reduce the modeling overhead, instead of applying the default (usually slow) scheme, RepBuf, we simply instrument parameter collection codes on the fast parallel algorithms, e.g., SelPriv and LocalWr. This way, we do not have to switch back to the RepBuf scheme to collect the parameters when the pattern changes, which reduces the execution time for the loop instances.

D. Summary

In this chapter, we described how we specialized our proposed *adaptive algorithm selection* framework for parallelizing irregular reductions.

We identified few high-level, architecture-independent parameters to characterize programs' static structure and dynamic data access patterns, and to discriminate candidate transformations. We developed a systematic model generation process, including an off-line synthetic experiment process, to generate performance prediction models that are used to dynamically select the most appropriate optimization transformations among several functionally equivalent candidates.

With parallel reduction algorithms being just one important example, our *adaptive algorithm selection* framework can also be used for various other compiler transformations that cannot be easily analytically modeled.

CHAPTER V

EXPERIMENTAL RESULTS OF ADAPTIVE REDUCTION SELECTION

In this chapter, we present experimental results on evaluating the effectiveness of applying our adaptive algorithm selection framework on adaptive reduction selection. We first show results using a set of static irregular applications to demonstrate that our framework can select the best performing algorithm and significantly improve performance. We then show that our technique can adaptively select the best algorithm for each phase of a dynamic program to achieve performance otherwise not possible.

Since our goal is to dynamically select the best algorithm, we mainly show the relative performance of the candidate algorithms, to better clarify the approach.

A. Evaluation of Algorithm Selection Framework

In this section we evaluate our automatically generated performance models. We show performance data (speedups) for reduction loops from several codes parallelized using the parallel reduction algorithms in our library and executed with several different inputs on multiple platforms. We compare the actual performance data of the algorithm selected by our automatically generated prediction models with the other algorithms.

1. Experimental Setup

We studied two parallel systems: an UMA HP V-Class with 16 processors [38] and a NUMA IBM Regatta p690 system with 32 processors [39]. The machine configurations are briefly described in Table V. In the IBM Regatta p690 system, each POWER4 chip contains 2 processors and a multi-chip module (MCM), which con-

TABLE V

SPECIFICATIONS OF EXPERIMENTAL PARALLEL SYSTEMS

	HP V2200	IBM Regatta p690
CPU Type	PA-8200	POWER 4
CPU Clock	200 MHz	1300 MHz
Data Cache	2 MB	32 KB / 1.48 MB / 32 MB
Physical Memory	4 GB	64 GB
# CPUs	16	32 / 4 MCMs
Topology	16 × 16 crossbar	buses / token ring
OS	HP-UX 11.0	AIX 5
Compiler	HP f90, c89	xlf_r, xlc_r

tains 4 POWER4 chips connected via 4 buses. Each chip sends requests, commands, and data on its own bus but snoops all buses. However, when interconnecting multiple MCMs, the intermodule buses act as a ring. The result is that communication across MCMs is significantly slower than that within a MCM.

Due to the limited size of our input sets and constraints on our available single user time, we used an 8-processor subsystem of the HP machine and a 16-processor subsystem of the IBM machine, which was sufficient for us to exercise the architectural characteristics of these two systems.

For the results obtained on the HP system, we used the same 7 programs (described in Table II) used in Section C.1 of Chapter III and ran all the 22 application/input combinations. Due to limitations on our single user time allocation, we did not obtain results for FMA3D on the IBM system, and so we only show results on the IBM system for 21 application/input cases.

In Table VI, the parameters of the application/input cases are given. As we mentioned in Section 1 of Chapter IV, among the parameters, the **MOB** is statically available. The **N** and the **CON** are treated as input dependent. The **OTH**, **SP** and (**CLUS**) have to be computed or measured at run-time and they have different values for the same loop and input case on the two machines.

TABLE VI

ACTUAL PARAMETER VALUES

APP	Static Parameters			Dynamic Parameters					
				HP, P=8			IBM, P=16		
	N	CON	MOB	OTH	SP	CLUS	OTH	SP	CLUS
IRREG	100000	100	2	0.98	1	1	0.98	1	1
	500000	50		0.88	0.92	37679	0.88	0.92	37679
	1000000	5		1.21	0.71	204295	0.83	0.47	248662
	2000000	1		1.20	0.22	344600	1.15	0.12	207438
NBF	1280000	2	2	5.34	0.26	2.12	0.59	0.13	2.06
	256000	5		5.60	0.25	2.12	0.97	0.12	2.06
	128000	50		5.87	0.25	2.12	0.54	0.12	2.06
	25600	200		5.26	0.25	2.12	0.44	0.12	2.06
MOLDYN	49152	71.3	2	4.02	0.50	9.99	2.22	0.36	34
	127776	23.3		4.12	0.39	19.93	2.44	0.25	61
	210912	7		4.57	0.29	46	2.93	0.24	148
	324000	2		5.01	0.29	601	2.13	0.24	73
CHARMM	995328	5.97	2	0.22	0.05	30	0.71	0.04	618
	995328	2.99		0.41	0.03	16.60	0.94	0.03	601
	1990656	1.49		0.38	0.03	16.60	0.38	0.02	630
SPARK98	21282	1.6	2	0.72	0.11	113	0.72	0.11	113
	90507	1.63		0.81	0.10	337	0.81	0.10	337
SPICE	186944	0.040	28	2.99	0.06	960	2.99	0.06	960
	98691	0.058		2.64	0.06	571	2.64	0.06	571
	89026	0.048		3.25	0.06	111	3.25	0.06	111
	33726	0.047		3.20	0.06	71	3.20	0.06	71
FMA3D	524286	0.167	8	0.89	0.13	8972			

2. Results of Reduction Algorithm Selection

Fig. 17 presents the results obtained on the HP V-Class system and the IBM Regatta p690 system. Each group of bars shows the relative performance (normalized to the best speedup obtained for that group) of the four parallel reduction algorithms for one program/input case. In most cases, the algorithm rankings resulting from the automatically generated regression model were consistent with the actual rankings. Overall, our regression model correctly identified the best algorithm for 18 out of 22 cases on the HP system and 19 out of 21 cases on the IBM system. As the arrows

in the graphs show, in all the mis-predicted cases the regression model identified the algorithm that performs close to the best one. Moreover, in all such cases, there was little performance difference between the best algorithm and the one that our models recommended.

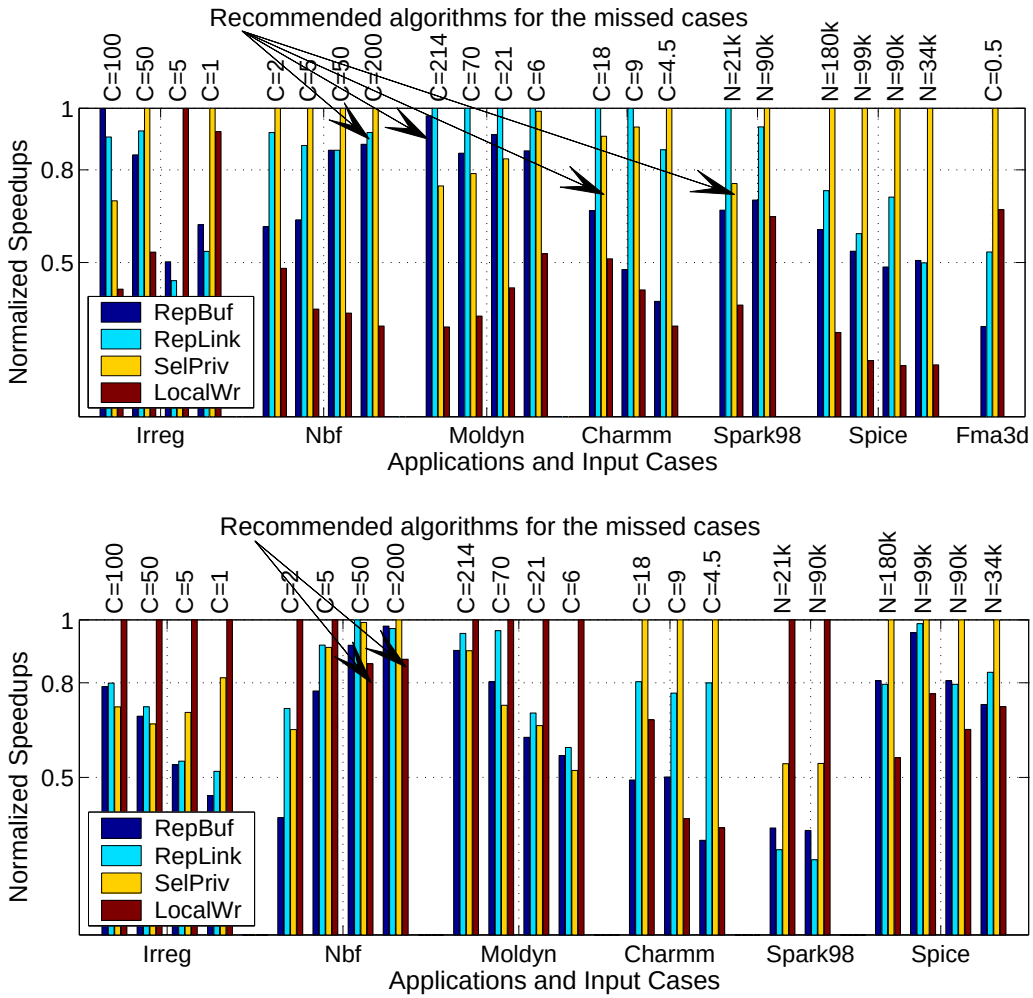


Fig. 17. Relative performance of parallel reduction algorithms (obtained on an 8 processors HP V-Class subsystem and a 16 processors IBM Regatta p690 subsystem)

To give a quantitative measure of the performance improvement obtained using the algorithms recommended by our models, we compute the *relative speedup* which we define as the ratio between the speedup of the algorithm chosen by an *alternative*

selection method and the speedup of the algorithm recommended by *our model*. We compared the effectiveness of our prediction models against the following *alternative selection methods*:

- **The Best** is a "perfect predictive model," or an "oracle," that always selects the best algorithm for a given loop–input case.
- **RepBuf** always applies *replicated buffer*, which is the simplest algorithm and it is specified as default by OpenMP standard [33].
- **Random** randomly selects a parallel algorithm. The speedup obtained is the average speedup of all the candidate parallel algorithms for that case.
- **Default** always applies the default parallel algorithm for a given platform. Based on our experimental results, we chose SelPriv and LocalWr as the default algorithms for the HP and IBM systems, respectively.

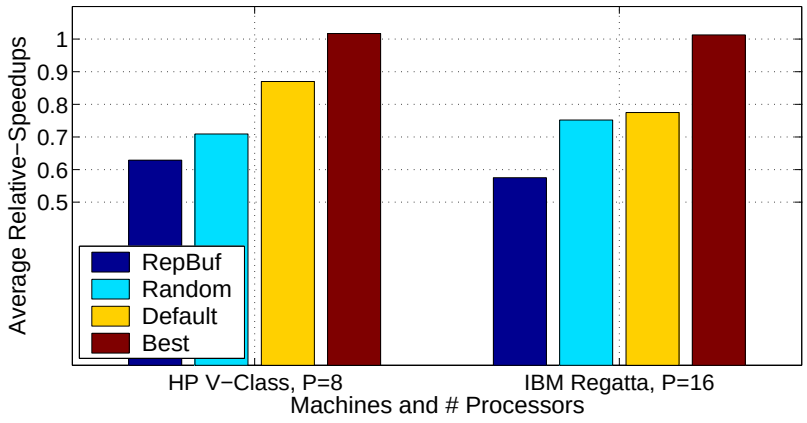


Fig. 18. Average relative-speedups

Fig. 18 gives the average *relative speedups* — normalized (to the speedups obtained when applying the algorithms recommended by our models) speedup across

all the loop-input cases. In the graph, the smaller the *relative speedup* value, the better the relative effectiveness of our prediction models. Here, the comparisons show that our automatically generated prediction models work almost as well as the "perfect predictive models," obtaining more than 98% of the best possible performance. Comparing to other "non-perfect" selection methods, our prediction models can improve the performance of irregular reductions significantly. Specifically, the average performance obtained using our prediction model is 60% better than using *replicated buffer*, which is the default parallel reduction implementation specified by OpenMP standard [33].

As mentioned in Fig. 16, our instrumentation introduces extra computation in the RepBuf algorithm to measure the parameters, **OTH**, **SP** and **CLUS**. We measured the run-time overhead (normalized to the RepBuf execution time) of collecting these parameters. While the overhead of measuring **OTH** has been reduced to a negligible level using a light-weight timer ($\approx$ 100 clock cycles) and only measuring 0.2% of all iterations, the overhead of computing **SP** and **CLUS** is proportional to the size of the reduction data array because we instrument the cross-processor reduction phase. Across all cases, the average overheads on the HP and IBM systems are 11.95% and 11.65%, which can be largely amortized since this overhead is only incurred on loop instances where the reduction pattern changes.

Finally, we note that knowledge of the dynamically collected parameters (**N**, **CON**, **OTH**, **SP** and **CLUS**) is very important for our models. From Fig. 17, the best schemes for different inputs change and our technique predicts the best schemes correctly by collecting and utilizing the dynamically collected parameters (especially for the bars corresponding to the results for IRREG and CHARMM on the HP system).

3. Validation for Regular Reductions

To test the robustness and generality of our prediction models, we applied them to two regular reduction loops: loop `loops_do400` in SU2COR from the SPEC'92 suite, and loop `actfor_do500` in BDNA from the PERFECT suite. For regular reduction loops, the size of the reduction data (usually a vector) is relatively small and all elements of the vector are accessed in every loop iteration. Either selectively replicating the vector data or partitioning the vector will not help performance (reducing cross–processor communications). The experimental results indicate that *replicated buffer* is always the best algorithm and that our prediction models give the correct recommendations. Hence, our adaptive reduction selection technique is generally applicable to all reductions.

B. Experimental Results on Dynamic Programs

In this section, we present experimental results showing that our adaptive algorithm selection technique can select the best parallelization scheme dynamically and thereby improve the overall performance of adaptive irregular programs. In this section, DynaSel represents applying algorithm selection dynamically for every computation-phase (described in Section 3).

1. 2D Adaptive Mesh Refinement

AmrRed2D is a synthetic program written by us. It is inspired by modern *Computation Fluid Dynamics* and *Structural Simulation* applications which use *Adaptive Mesh Refinement (AMR)* [40, 41]. Fig. 19 gives the high-level description of the program. AmrRed2D implements an irregular reduction on the nodes of an unstructured 2D triangular mesh and it does not conduct any "useful" computation, e.g., it does not

solve any equations to simulate physics. Our purpose here is to simulate the effect of the adaptive mesh refinement on irregular reductions.

```
0     Initialize 2D triangular mesh
      FOR (each time step: T) DO
        IF (T mod F = 0) THEN
1         Refine and coarsen parts of the mesh
2       FOR (each node: A) do
          FOR (each neighbor node: B) DO
2.1         Compute interactions of A & B.
2.2         Update data associated to A & B.
```

Fig. 19. High level description of AmrRed2D

We number the nodes in lexicographic order based on their spatial (x, y) coordinates. After every step of refinement and coarsening of the mesh, the nodes are re-numbered. Each mesh adaptation (refinement and coarsening) indicates the start of a new dynamic computation phase. In terms of data distribution, we distribute the nodes in a blocked manner (each processor hosts a block of nodes with contiguous node IDs). This way, the characteristics of the reductions change across adaptation phases; in particular, the inter-processor communication pattern of the reductions might change. For instance, suppose one node and its neighbor are distributed on two contiguous processors. After one or more steps of the refinement, the node and its neighbor may end up being distributed on processors further apart.

For AmrRed2D, we experimented with two inputs with different initial mesh sizes (50x50 and 300x300 nodes, respectively) and different refinement rates (specifying a fraction of mesh to refine) on the 8-processor subsystem of our HP V-class machine (see Table V). The first input, with an initial mesh having 50x50 nodes, executes the reduction loop 600 times and each phase contains 20 instances. The second input, with an initial mesh having 300x300 nodes, executes the reduction loop 300 times and

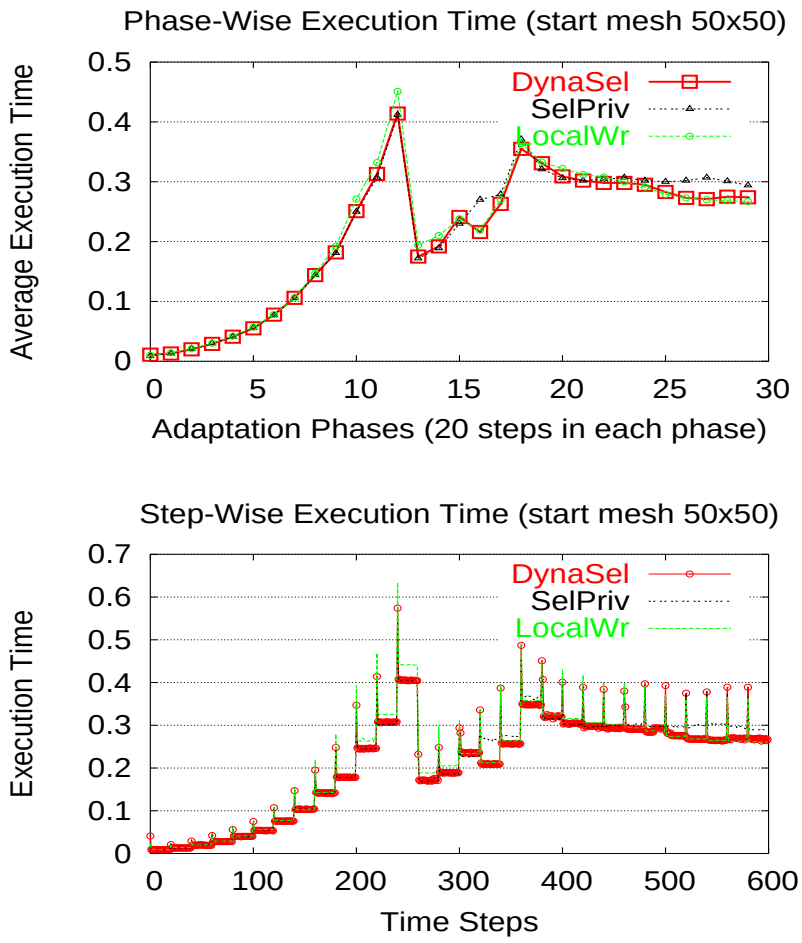


Fig. 20. Phase-wise and step-wise effects of dynamically selecting algorithms AmrRed2D with initial mesh sizes 50x50

each phase contains 15 instances. Together with the inputs, the program continuously refines the lower left part of the mesh within a decreased domain area and coarsens the rest of the mesh. When the number of the nodes of the entire mesh hits an upper limit, the program coarsens the entire mesh several times and starts refining again.

Fig. 20 and 21 show the step-wise and phase-wise execution times when applying different algorithms (with DynaSel as an "algorithm"). The graphs titled "step-wise execution time" report the execution time in seconds for each execution

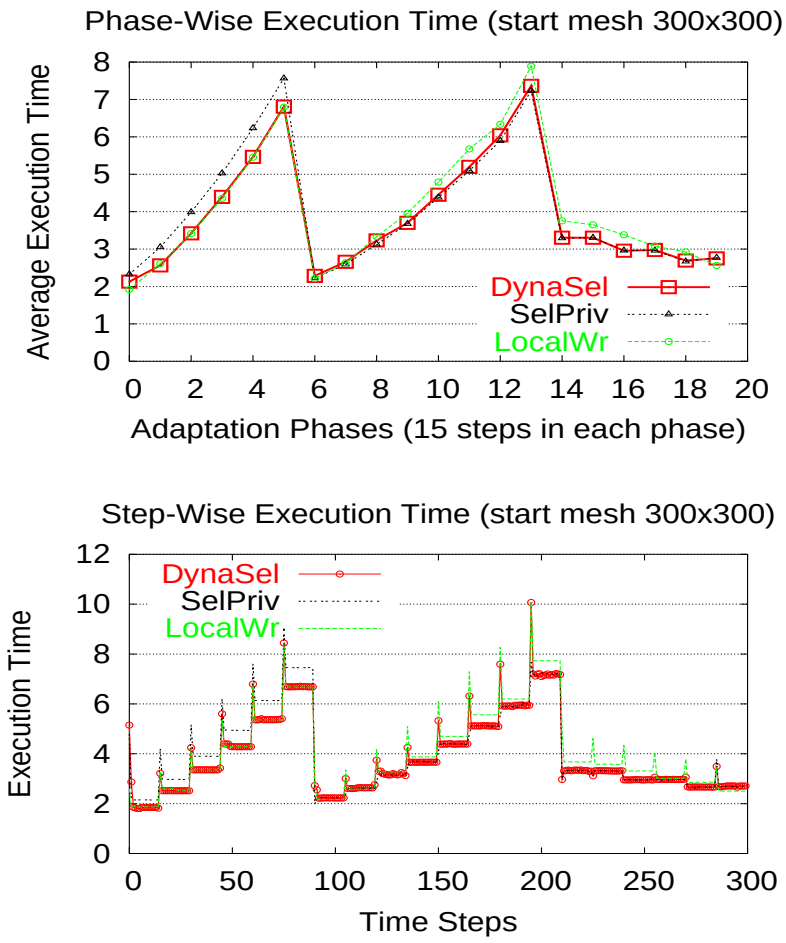


Fig. 21. Phase-wise and step-wise effects of dynamically selecting algorithms AmrRed2D with initial mesh sizes 300x300

instance (corresponding to time step) of the parallel reduction loop. The graphs titled "phase-wise execution time" report the average execution time in seconds for each execution phase of the parallel reduction loop. Since RepBuf performed very poorly for AmrRed2D, we choose not to include its execution time in the graph in order to to show the effect of DynaSel more clearly. The main trend of the change of the step-wise/phase-wise execution time is due to the change of the mesh size (measured as the number of nodes of the adaptive mesh). There are a couple of observations

from these graphs. First, the best parallel algorithms change dynamically during execution. Second, for most dynamic phases, DynaSel makes the right decision and applies the appropriate transformation algorithms. That is, DynaSel capitalizes on the opportunity for run-time optimization.

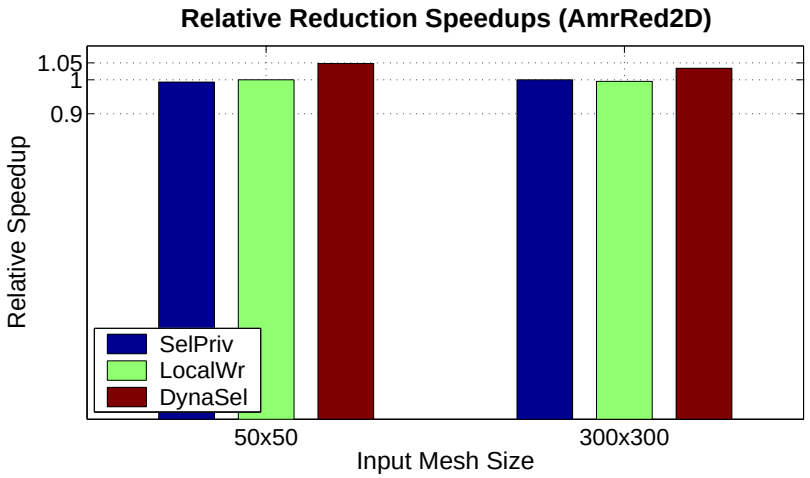


Fig. 22. Relative speedups of adaptively selecting algorithms on AmrRed2D

Fig. 22 gives the relative speedups of the reduction loop across all steps for different parallel reduction algorithms. The speedups are normalized to the best results obtained by applying one algorithm to clarify the improvement. The conclusions here are that using our adaptive technique to select and apply the best parallel reduction algorithm for every dynamic phase has little overhead and that it out-performs any other single algorithm and improves overall performance by $\sim 5\%$.

2. Molecular Dynamics

Classical molecular dynamics (MD) is a widely used computation tool for simulating the properties of liquids, solids, and molecules. The atoms or molecules in the system are treated as point masses and Newton's equations are integrated to compute their motion.

MOLDYN is a synthetic benchmark, conducting non-bonded force calculations in a molecular dynamics simulation. It has been widely used for the purpose of evaluating systematic optimization transformations [20, 26, 42].

```
Initialize the coordinates of particles.
FOR (N time steps) DO
1   Move particles based on their forces and velocities.
2   IF (N mod K = 0) THEN
      Build a neighbor list for each particle (with a
              specified radius).
3   FOR (each node: A) DO
      FOR (each neighbor: B) DO
        Compute the interaction of A and B and update
              the forces of A and B.
4   Update the velocities of the particles.
```

Fig. 23. A high level description of MOLDYN

TABLE VII
SPECIFICATIONS OF DYNAMIC INPUTS OF MOLDYN

ID	#molecules	cut-off radius	avg. CON	#steps	#phases
1	23328	4.0	157	60	23
2	186624	2.5	37.6	60	23
3	100800	2.0	21.8	60	21
4	186624	1.5	6.6	60	21
5	186624	1.2	5.4	60	21

A high-level description of MOLDYN is given in Fig. 23. In this original implementation, Step 2 is an $\Theta(N^2)$ operation that iterates through all the pairs of particles to build a neighbor list for each particle, and Step 3 is in quasi-linear time and performs the real computations. However, for most molecular dynamics applications, the non-bonded forces between particles are limited to a range, usually called the *cut-off radius*, and there is no need to iterate through every pair of particles to evaluate their interactions. We replaced the previous implementation of Step 2 with

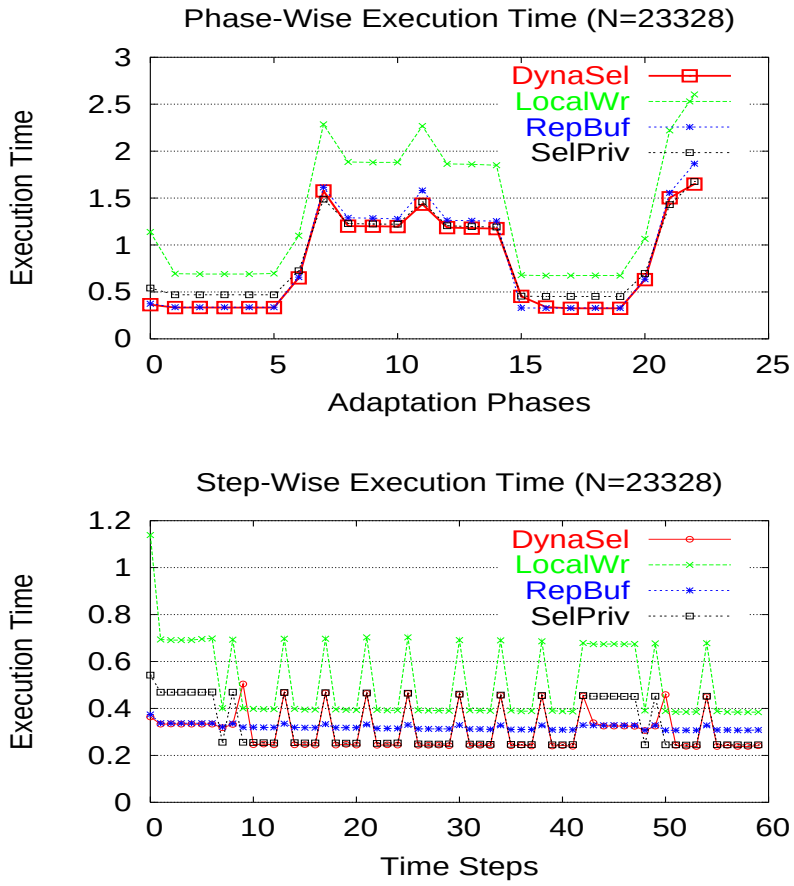


Fig. 24. Phase-wise and step-wise effects of dynamically selecting algorithms MOLDYN (inputs #1)

an algorithm using a link-cell data structure to generate the neighbor lists [1]. Here, space is tiled with 3D cubes with sides slightly greater than the cut-off radius, and the atoms are placed in the cube containing their centers. This way, the neighbor list for a particle can be found by checking only the particles residing in neighboring boxes. With this modification, the execution time of Step 2 is significantly reduced so that it is comparable to the time required for Step 3 for a simulated system with a realistic number of particles. Since step 3 is executed more times than Step 2, the execution time of the modified MOLDYN is now truly dominated by the force computation,

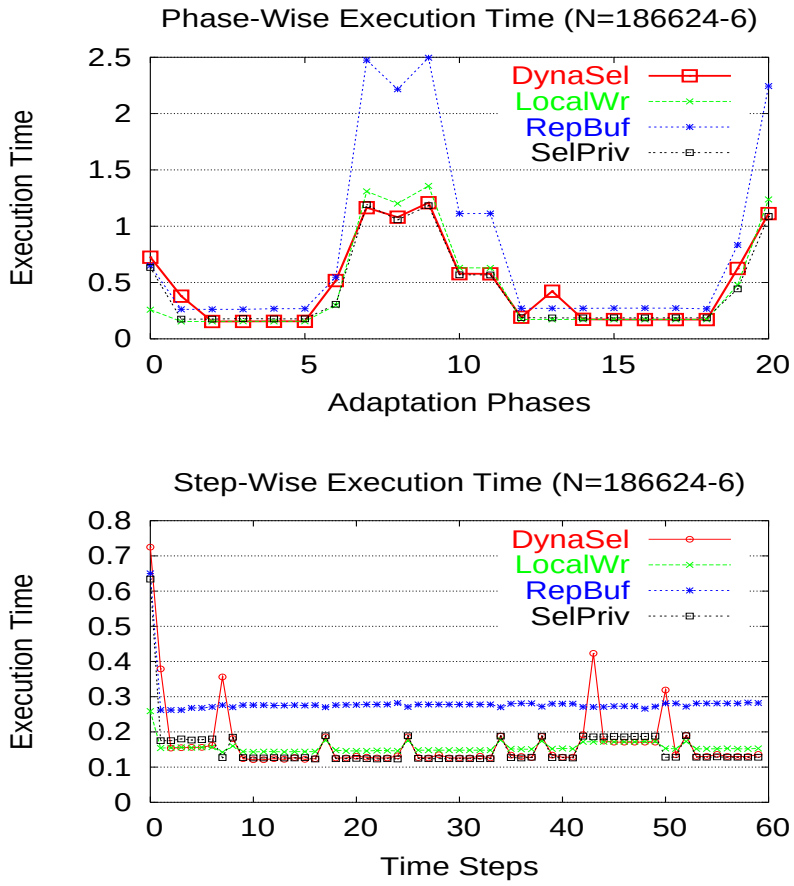


Fig. 25. Phase-wise and step-wise effects of dynamically selecting algorithms MOLDYN (input #4)

which is the irregular reduction we would like to optimize.

For MOLDYN, since we have not observed much performance change across dynamic phases for parallel algorithms, we artificially set the *reusability* (the number of steps) for each dynamic phase so that the phase-wise best algorithms can change due to the different setup overheads of the algorithms. This way, we can examine both the effectiveness of the prediction models and the efficiency of selecting and switching algorithms. We experiment with 5 different inputs on the 8-processor subsystem of our HP V-class machine (see Table V). The input specifications are given in Table VII.

Note that different phases may have different numbers of time steps. In this program, since the **CON** parameter changes from phase to phase, the `connectivity` column is the average value across phases.

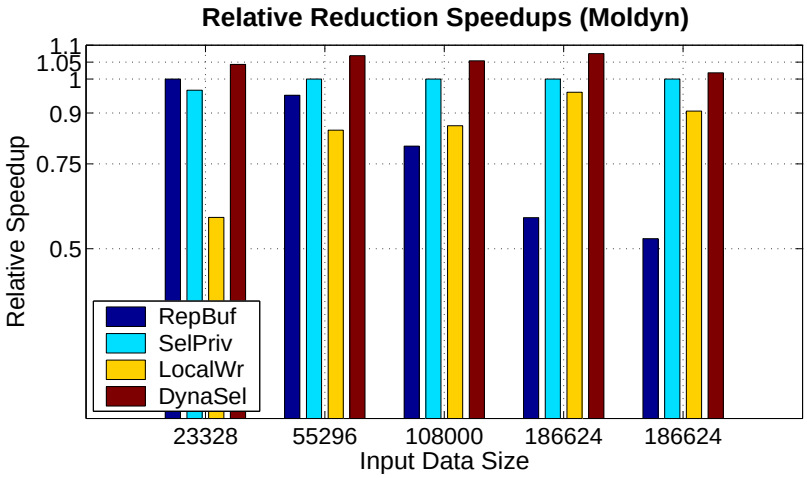


Fig. 26. Relative speedups of adaptively selecting algorithms on MOLDYN

Due to space limitations, we show the step-wise and phase-wise execution time of DynaSel and the best two individual algorithms (RepBuf and SelPriv) in Fig. 24 and 25 for two inputs (#1 and #4). We note that since the number of steps of the phases may change, therefore in the phase-wise plots, we plotted the accumulated time of each phase instead of the average time. Again, the results show that DynaSel out-performs applying any one algorithm for most cases.

3. PP2D in FEATFLOW

We have applied our adaptive technique to a real program, PP2D in FEATFLOW package. FEATFLOW is a general purpose subroutine system (in F77) for the numerical solution of the incompressible Navier-Stokes equations in 2D and 3D. The software package is comprehensive. It contains preprocessing, solver and post-processing parts. The solution part contains the solver package purely time dependent (projec-

tion schemes) and solving stationary and non-stationary problems in a fully coupled way. For detailed information of FEATFLOW, please refer to [43].

Particularly, PP2D solves nonlinear coupled equations using multi-grid solvers and has about 17,000 lines of code (excluding codes of the underneath libraries).

In program PP2D, for the given input-data, we have found a relatively heavy loop in subroutine GUPWD containing irregular reductions (about 11% of the whole program). GUPWD updates a sparse matrix with old velocity values associated to grid nodes. The memory access pattern of the irregular reduction is fully determined by the indirection data structures describing the sparse matrix.

The program uses multi-grid method to solve linear systems for each time step. The multi-level grids are predefined (specified in input data) and the sparse matrix structures associated to different grids are defined in an initialization step outside the time-step loop. For the input we have, 4 grids are used (the program specifies maximal number of grid levels as 9). For the reduction loop, the four different sparse-matrix data structures of the grids are used in an interleaved manner in GUPWD.

Therefore, our adaptive algorithm selection technique can be applied for each grid level via "schedule reuse." Specifically, the augmented code selects the best reduction algorithm for each grid right before the first invocation of the reduction loop for the grid and the decision is reused for the later invocations for the same grid. As a result, we can treat the dyanmic invocations of the loop for each grid as an input-dependent case and only 4 algorithm selections are needed.

Due to the complexity of the program (50k lines of codes for the compilation of PP2D) and the limitation of our research compiler infrastructure, Polaris, we have instrumented part of the code by hand.

For PP2D, we have experimented with one input (`comp`) that is distributed together with the source code for benchmarking purpose. For this input, the program

TABLE VIII
SPECIFICATIONS OF GRID LEVELS OF THE INPUT FOR PP2D

Level	#unknowns	#nodes	#elements	#instances
2	12930	1890	920	86
3	52620	7460	3680	86
4	206280	29640	14720	86
5	824720	118160	58880	166

works on 4 grids, with minimal and maximal grid level specified as 2 and 5, respectively. The detailed specifications (in the interests of adaptive reduction selection) of the 4 grids are given in Table VIII.

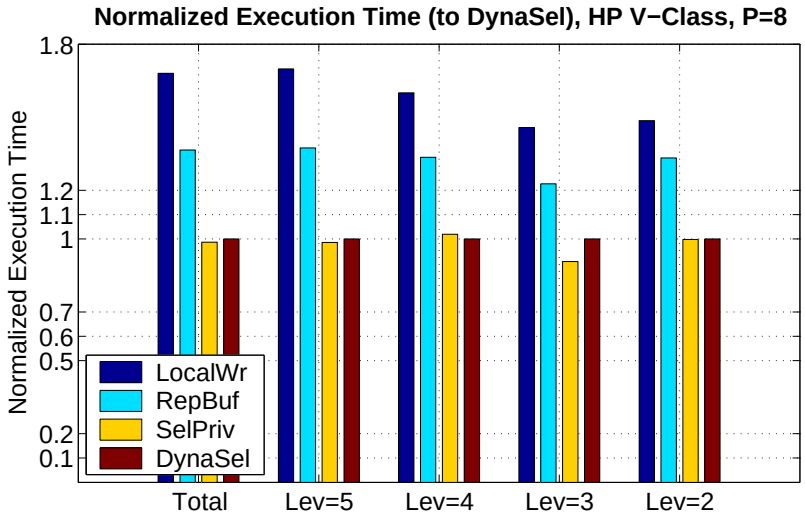


Fig. 27. Relative performance of adaptive algorithm selection for PP2D, on a HP V-Class

Fig. 27 gives the relative execution time of the GUPWD loop obtained on our HP V-Class system (see Table V). The legends REPBUF, LOCALWR and SELPRIV are results obtained by apply the specified algorithms for all 4 levels. The legends DYNASEL corresponds to apply our adaptive algorithm selection framework to select and apply algorithms for different grid levels. All the execution time are normalized to the execution time of the GUPWD loop applying DYNASEL. In the graph, we

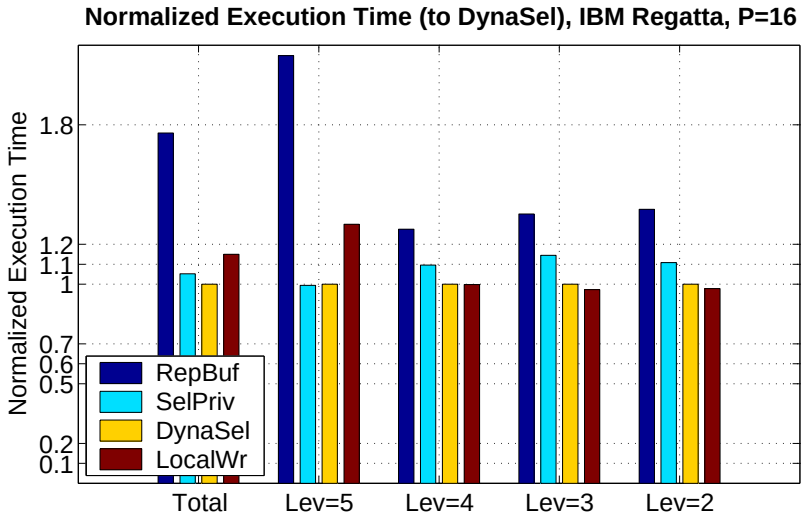


Fig. 28. Relative performance of adaptive algorithm selection for PP2D, on an IBM Regatta p690

also show the relative time for the 4 grid levels. These times are normalized to the corresponding execution time of the GUPWD loop applying DynaSel. On this HP system, although our DynaSel has selected the best algorithm (SelPriv for all the grid levels), it did not improve the overall performance of the loop. Nevertheless, this experiment shows that our DynaSel introduced negligible overhead (because the dynamic algorithm selection code is only called once for each grid). Note that for level 3, the graph shows that SelPriv is about 5% faster than DynaSel. This is because the actual execution time of one loop instance for the coarse grid levels (e.g., level 2 and level 3) is fairly short (0.0025sec for level 2 and 0.01sec for level 3), which are close to the parallel loop forking overhead of the system [44]. We expect that larger inputs would have more stabler results.

Fig. 28 gives the relative execution time of the same loop obtained on the 16-processor subsystem of the IBM Regatta p690 (see Table V). The relative time for all the grid levels indicate that the performance of DynaSel for each grid level align to the actual best algorithm for the grid level (due to correctly selection between Sel-

PRIV and LOCALWR) . This way, DYNASEL obtains the overall best performance, which is shown via the group of bars marked as "Total" in the graph.

At this point, we want to note that for all dynamic applications we have experimented, AmrRed2D, MOLDYN and PP2D, though the performance improvement using our adaptive algorithm selection framework is limited (upto 8%), the results demonstrate the success of our technique on capitalizing the opportunity for run-time optimization. The performance improvement is otherwise not possible.

C. Summary

In this chapter, we have illustrated the effectiveness of our adaptive reduction selection technique (an important application of our adaptive algorithm selection framework on parallelizing irregular reductions).

Our experiments on the IBM Regatta and HP V-Class systems show that our framework: (a) selects the best performing algorithms in 85% of the cases studied; (b) for the best possible algorithms were not selected all the time, the overall performance was still within 2% of the optimal scheme's performance; (c) achieves better performance by adaptively selecting the best algorithm for each phase of a dynamic program; and (d) adapts to the underlying machine architecture.

In short, through our experiments, we have demonstrated that the presented adaptive algorithm selection framework can model programs' irregular and dynamic behavior extremely well and can customize solutions every time we need to. The end result of using this framework is that much better sustainble performance can be achieved.

CHAPTER VI

ADAPTIVE RUN-TIME PARALLELIZATION FOR LOOPS WITH SPARSE MEMORY ACCESSES

A. Introduction

Current parallelizing compilers cannot identify a significant fraction of parallelizable loops because they have complex, or statically insufficiently defined access patterns. As parallelizable loops arise frequently in practice, Rauchwerger and Padua have introduced a general speculative parallelization framework for their identification. In [17], they have shown that employing run-time techniques that trace "relevant" memory references and decide whether a loop is parallel or not is a viable method to improve the results of classic, static automatic parallelization.

While previous work on speculative parallelization [36] has shown that this method is inherently scalable, its practical success depends on the fraction of ideal speedup that can be obtained on modest to moderately large parallel machines. Maximum parallelism can be obtained only through a minimization of the run-time overhead of the method, which in turn depends on its level of integration within a classic restructuring compiler and on its adaptation to characteristics of the parallelized application.

In this chapter we present a set of compiler and run-time techniques designed specifically for optimizing the run-time parallelization of loops with irregular and sparse memory access patterns. We show how we minimize the run-time overhead associated with the speculative parallelization of sparse applications by using static control- and data-flow information to reduce the number of memory references that have to be collected at run-time. We then present heuristics to speculate on the

reference type and data structures used by the program and thus reduce the memory requirements needed for tracing the sparse access patterns. Experimental results conclude the chapter. All static and dynamic techniques introduced here represent a sparse implementation of the LRPD test [17].

B. Foundational Work - the LRPD Test for Dense Problems

To detect and exploit loop level parallelism in various cases encountered in irregular applications, a number of techniques [45, 46, 47, 48, 17] have been developed. Several representative techniques are: (i) a speculative method to detect fully parallel loops (the LRPD test [17]); (ii) an inspector/executor technique [45] to compute wavefronts (sequences of mutually independent sets of iterations that can be executed in parallel); and (iii) a technique for parallelizing `WHILE` loops (`DO` loops with an unknown number of iterations and/or containing linked list traversals) [48]. In this chapter we refer mostly to the LRPD test and how it is used to detect fully parallel loops. To make this dissertation self-contained we briefly describe a simplified version of the speculative LRPD test. A detailed description can be found in [46, 17].

1. the LRPD Test

The LRPD test speculatively executes a loop in parallel and subsequently tests whether any data dependence could have occurred. If the test fails, the loop is re-executed in a safe manner, e.g., sequentially. To qualify more parallel loops, *array privatization* and *reduction parallelization* can be speculatively applied and their validity tested after loop termination.[1] For brevity the details related to reduction

[1] *Privatization* creates, for each processor cooperating on the execution of the loop, private copies of the program variables. A shared variable is privatizable if it is always written in an iteration before it is read, e.g., many temporary variables [49, 50, 51].

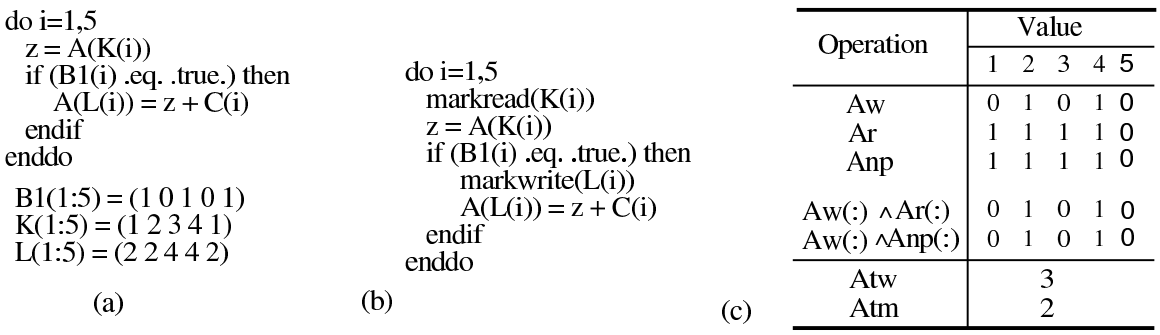
recognition and parallelization are not given here; it is tested in a similar manner as independence and privatization. The LRPD test is fully parallel and requires time $O(a/p + \log p)$, where p is the number of processors, and a is the total number of accesses made to the array under data dependence test in the loop.

Consider a `DO` loop for which the compiler cannot statically determine the access pattern of a shared array `A` (Fig. 29(a)). The LRPD test allocates shadow array $\mathtt{A_w}$ for marking the write accesses, $\mathtt{A_r}$ for the read accesses, and an array $\mathtt{A_{np}}$, for flagging non-privatizable elements. The loop is augmented with code (Fig. 29(b)) that marks, during speculative execution, the shadow arrays every time A is referenced (based on specific rules). The result of the marking can be seen in Fig. 29(c). For the first time, an element of `A` is written in an iteration and the corresponding element in the write shadow array $\mathtt{A_w}$ is marked. If, during any iteration, an element of `A` is read, but never written, then the corresponding element in the read shadow array $\mathtt{A_r}$ is marked. Another shadow array $\mathtt{A_{np}}$ is used to flag the elements of `A` that cannot be privatized. An element of $\mathtt{A_{np}}$ is marked if the corresponding element of `A` is both read and written, and is read first in any iteration.

After the speculative parallel execution, a post-execution analysis, illustrated in Fig. 29(c), determines whether there were any cross-iteration dependencies between statements referencing `A` as follows. If $\mathtt{any}(\mathtt{A_w}(:) \wedge \mathtt{A_r}(:))$[2] is true, then there is at least one flow- or anti-dependence that was not removed by privatizing `A` (some element is read and written in different iterations). If $\mathtt{any}(\mathtt{A_{np}}(:))$ is true, then `A` is not privatizable (some element is read before being written in an iteration). If Atw, the total number of writes marked during the parallel execution, is not equal to Atm, the total number of marked elements (computed after the parallel execution), then there

[2] `any` returns the "OR" of its vector operand's elements , i.e., $\mathtt{any}(\mathtt{v}(1:\mathtt{n})) = (\mathtt{v}(1) \vee \mathtt{v}(2) \vee \ldots \vee \mathtt{v}(\mathtt{n}))$.

is output dependence (existing elements written concurrently in different iterations). However, if A is privatizable (i.e., if any(A_{np}(:)) is false), then these dependencies are removed by privatizing A.



Operation	Value				
	1	2	3	4	5
Aw	0	1	0	1	0
Ar	1	1	1	1	0
Anp	1	1	1	1	0
Aw(:) ∧Ar(:)	0	1	0	1	0
Aw(:) ∧Anp(:)	0	1	0	1	0
Atw	3				
Atm	2				

Fig. 29. An example of LRPD test on a DO loop. (a) sequential loop; (b) transformed for speculative execution, the markwrite and markread operations update the appropriate shadow arrays; (c) shadow arrays after loop execution. In this example, the test fails

2. Overhead of the LRPD Test for Dense Access Patterns

The overhead spent performing the LRPD test, tracing the accesses to A and verifying the validity of the parallel execution, scales well with the number of processors and data set size of the parallelized loop. For dense access patterns the best choice for the shadow structures are *shadow arrays* conformable to the arrays under test because they provide fast random access to its elements and can be readily analyzed in parallel during the post-execution analysis phase. The efficiency of the algorithm is high because (almost) all allocated shadow elements are used. We can break down the extra time spent on speculatively parallelizing a loop with the LRPD test into the following components:

1. The *initialization of shadow structures* - takes time proportional to the size of the shadow structures (arrays).

2. *Checkpointing* the state of the program before entering speculation takes time proportional to the number of distinct shared data elements that may be modified by the loop. The work involved is approximately equal to saving all modified shared arrays and is thus highly program dependent.

3. The overhead associated with the speculative execution of the loop is equal to the time spent marking (recording) the references to the arrays under test, i.e., proportional with the dynamic count of the references to the arrays.

4. The final *analysis of the marked shadow structures* is, in the worst case, proportional to the number of distinct memory references marked on each processor and to the (logarithm of the) number of processors. For dense access patterns, this phase is equivalent to a parallel merge of p shadow arrays (p is the number of processors).

5. In case the speculation fails, the *safe re-execution of the loop* may cost as much as the restoration of the saved (via checkpointing) variables and a sequential re-execution of the original loop.

Each of these steps (except the sequential re-execution) is fully parallel and scales with the number of processors. An important measure of performance of run-time parallelization is its *relative efficiency*. We define this efficiency as the ratio between the speedup obtained through the automatic run-time parallelization techniques and the speedup obtained through hand-parallelization. In case hand-parallelization is not possible due to the dynamic nature of the code, then an ideal speedup is used. Another measure of performance is *potential slowdown*, i.e., the ratio between sequential, unparallelized execution time and the time it takes to speculate, fail, and re-execute. Our goal is to simultaneously maximize these two measures (equal to 1) and thus

obtain an optimized application with good performance.

While we do not consider increasing efficiency and reducing potential slowdown as being orthogonal, in this chapter we focus on presenting avenues to improve *relative efficiency*, i.e., how to increase speedups obtained for successful speculation.

3. Some Specific Problems in Parallelization of Sparse Codes

The run-time overhead associated with loops exhibiting a sparse memory access pattern has the same break-down as the one described in the previous section. The scalability and relative efficiency of the technique is, for practical purposes, jeopardized if we use the same implementation as the one used for dense problems.

The essential difficulty in sparse codes is that the size of the arrays under test may be orders of magnitude larger than the number of distinct elements referenced by the parallelized loop. Therefore, the use of shadow arrays can become prohibitively expensive. In addition to allocating much more memory than necessary (and cause all the known problems), the work of the initialization, analysis and checkpointing phases would not scale with data size and/or number of processors. We would have to traverse more elements than have been actually referenced by the loop and thus drastically reduce the relative efficiency of our general technique.

For these reasons we conclude that sparse codes need compacted shadow structures. However, such data structures (e.g., hash tables, linked lists, etc) do not have, in general, the desirable random, fast access time of arrays. This in turn increases the overhead represented by the actual marking (tracing) of references under test during the execution of the speculatively parallelized loop.

Another important optimization specific to sparse codes is the parallelization of reductions. This is a quite common operation in scientific codes and also has to be specialized for cases of sparse codes. Since we have presented such techniques

in previous chapters of this dissertation, they will not constitute the focus of this chapter.

Sparse codes rely almost exclusively on indirect, often multi–level addressing. Furthermore, such loops may traverse linked lists (implemented with arrays) and use equivalenced offset arrays to build C-like structures. These characteristics, as we will show later, result in a statically un-analyzable situation in which even the most standard transformations like loop distribution and constant propagation, cannot be performed (all statements end up in one strongly connected component). It is therefore clear that different, more aggressive techniques are needed. We will further show that a possible solution to these problems is the use of compiler heuristics to speculate on the type of the data structures used by the original code, which can be verified at run-time.

A representative and complex example can be found in subroutine BJT of SPICE 2G6 [52], a well known and widely used circuit simulation code. The unstructured loop (implemented with `GOTO` statements) traverses a linked list and evaluates the model of a transistor. It then updates the global circuit matrix (via sparse and irregular reductions). All shared memory references are to arrays that are equivalenced to the same name (`value`) and use several levels of indirection. Because almost all references may be aliased, no classic compiler analysis can be directly applied.

4. Overhead Minimization

Our simple performance model of the LRPD test gives us the general directions for performance improvement. To reduce slowdown we need to improve the probability of successful parallelization and reduce the time it takes to fail a speculation. The techniques handling this problem are important but will not be detailed here. Instead, we will now present several methods to reduce the run-time overhead associated with

run-time parallelization. First we present, in Section C), a generally applicable technique that uses compile-time (static) information to reduce the number of references that need to be traced (marked) during speculative execution. Then in Section D we will present a method for sparse codes that speculates about the data structures and reference patterns of the original loop and customizes the shape and size of the shadow structures.

C. Redundant Marking Elimination

1. Same Subscript and Access Type Based Aggregation

While in previous implementations of the LRPD test we traced every reference to the arrays under test we find that such an approach incorporates significant redundancy. We only need to detect attributes of the reference pattern that will insure correct parallelization of loops. For this purpose, we classify memory references, similar to [53], as: (1) read only (RO), (2) write-rirst (WF), (3) read-first-write (RW), and (4) not referenced (NO). Here, NO or RO references can never introduce data dependence; WF references can always be privatized; and RW accesses must occur in only one iteration (or processor), otherwise they will potentially cause flow-dependences and invalidate the speculative parallelization. The overall goal of the algorithm is to mark only the necessary and sufficient sites to unambiguously establish the type of reference, WF, RO, RW, or NO by using the dominance (on the control graph) relationship.

Based on the control flow graph of the loop, we can aggregate the markings of read and/or write references (to the same address) into one of the categories listed above and replace them with a single marking instruction. The intuitive and elementary rule for combining Reads and Writes to the same address is shown in Fig. 30.

The algorithm relies on a depth first traversal of the control dependence graph

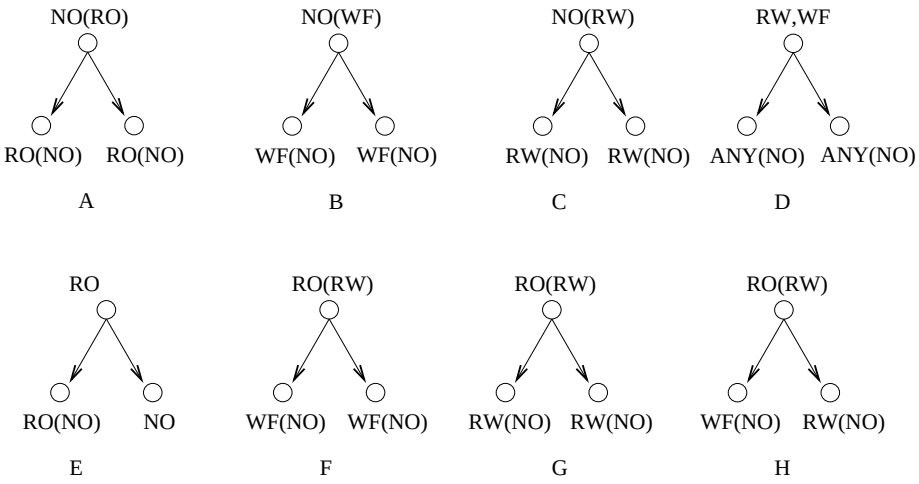


Fig. 30. Simple aggregation situations. Notes: the currently visited node is the root of an elementary CDG. `XX` represents the reference type before an aggregation and `(YY)` represents the type after an aggregation. For example, in (D), if the root is RW or WF, then it remains that way and the previous marks of the children, if any, are removed.

(CDG) [15] and the recursive combination of the elementary constructs shown in Fig. 30. First all Read and Write references are initialized to RO and WF respectively. Then, at every step of the CDG traversal we attempt to aggregate the siblings with the parent of the subgraph, remove the original marks and update the one at the root of the subgraph (or add a new one when applicable). When the marks of siblings cannot be directly replaced with the mark of the parent (because they are not of the same type), then references (marks) and their predicates are combined via logical union and passed to the next level. The final output of the algorithm is a loop with fewer marks than the number of memory references under test. Simplification of boolean expressions will further reduce the marking sites. The effectiveness of this method is program dependent and thus does not always lead to significant improvement (less marks).

It is important to remark that if predicates of references are loop invariant,

then the access pattern can be completely analyzed before the loop execution in an inspector phase. This inspector is equivalent to a LRPD test (or simpler run-time check) of a generalized address descriptor. Such address descriptors have been implemented in a more restricted form (for structured control-flow graphs) [54].

2. Grouping of Related References

We say that two memory addresses are *related* if they can be expressed as a function of the same base-pointer. For example, when subscripts are of the form *ptr + affine-function*, then all addresses starting at the point *ptr* are related. Here, both the *ptr* and the *affine-function* are specified based on their relation to the indices of the loop nest under test. The *ptr* is the sub-expression of a subscript that can not be represented as a linear function of the loop indices, where the *affine-function* is otherwise. For example, in SPICE, we find many indices to be of the form *ptr + const*, where *const* takes values from 1 to 50. In fact they are constructed through offsets of `EQUIVALENCE` declarations for the purpose of building C-like structures. The *ptr* takes a different value at every iteration. Intuitively, two related references of the same type can be aggregated for the purpose of marking if they are executed under the same control flow conditions, or more aggressively, if the predicates guarding two references have "logical imply" relation between them.

Formally, we define a *marking group* as a set of subscript expressions of references to an array under a run-time test that satisfies the following conditions: (1) the addresses are derived from the same base-pointer; (2) for every path from the entry of the considered block to its exit, all *related* array references are of the same type, i.e., they have the same attribute from the list WF, RO, RW, and NO. The *grouping algorithm* (outlined in Fig. 31 and further explained in the following section) tries to find a minimum number of disjoint sets of references of maximum cardinality

```
extract_grp (N, Bcond)

Input:      CdgNode       N
            Predicate     Bcond
Output:     Grouping      localGrp

S1    localGrp = compute_local_grp(N, Bcond)

      IF (N leads branch nodes) then
      FOR (each branch node B leaded from N) DO
        Grouping  branchGrp
        Predicate new_Bcond =
                  Bcond AND (Predicate of branch B)
        FOR (each cdg node N1 rooted in B) DO
          subGrp = extract_grp(N1, new_Bcond)
S2        branchGrp = grp_union(branchGrp, subGrp)

S3      localGrp = grp_intersect(localGrp, branchGrp)

      return localGrp
```

Fig. 31. Recursive grouping algorithm

(subscript expressions) to the array under test. Once these groups are found, they can be marked as a single abstract reference. The net result is: (a) a reduced number of marking instructions (because we mark several individual references at once) and (b) a reduced size (dimension) of the shadow structure that needs to be allocated because we map several distinct references into a single marking point. Fig. 32 gives an example showing the instrumented markings after applying the *grouping algorithm*. In the example, assume predicate `A` is loop variant so that the compiler could not hoist the marking codes out of the loop.

3. Outline of the Grouping Algorithm

This section gives more details of the *grouping algorithm* that we have mentioned in the previous section.

```
Before marking

S0      DO i = 1,N
S1        IF (A) THEN
S2          A(B(i)+1) = ...
S3          A(B(i)+2) = ...
S4          A(B(i)+3) = ...
          ELSE
S5          .. = A(B(i)+2)
S6          .. = A(B(i)+3)
S7          .. = A(B(i)+4)
          ENDIF
S8        IF (A) THEN
S9          A(B(i)+5) = ...
S10         A(B(i)+6) = ...
          ENDIF
        ENDDO

Groups:
    grp1: {B(i)+i | i=1,5,6}
    grp2: {B(i)+i | i=2,3}
    grp3: {B(i)+i | i=4}
```

```
After grouping and marking

S0      DO i = 1,N
S1        IF (A) THEN
            MARK_WRITE( grp1 )
S2          A(B(i)+1) = ...

            MARK_WRITE( grp2 )
S3          A(B(i)+2) = ...
S4          A(B(i)+3) = ...
          ELSE
            MARK_READ( grp2 )
S5          .. = A(B(i)+2)
S6          .. = A(B(i)+3)
            MARK_READ( grp3 )
S7          .. = A(B(i)+4)
          ENDIF

S8        IF (A) THEN
S9          A(B(i)+5) = ...
S10         A(B(i)+6) = ...
          ENDIF
        ENDDO
```

Fig. 32. Illustration of marking sites after grouping related references

a. CDG and colorCDG Construction

We represent control dependence relationships in a control dependence graph [15], with the same vertices as the CFG and an edge $(X - cd \rightarrow Y)$ whenever Y is control dependent on X. Fig. 33(a) shows the CDG for the loop example in Fig. 32. In Fig. 33(a), each edge is marked as a predicate expression. For multiple nodes that are control dependent on one node with the same predicate expression (e.g., Node S2, S3, S4 are control dependent on node S1 with predicate expression A) we put a branch node between S1 and S2, S3, S4 with label A. We name the resulting graph a colorCDG: the white node is the original CDG node and the black node is a branch node. The corresponding colorCDG for the example in Fig. 32 is shown in Fig. 33(b),

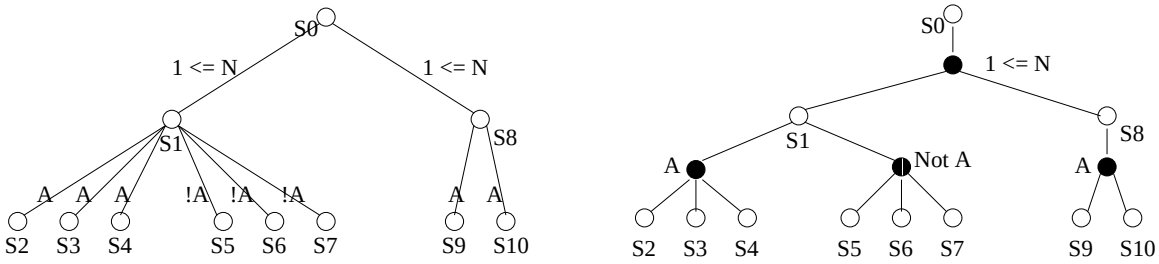


Fig. 33. CDG(a) and colorCDG(b) of the loop example

where node S1 represents a `IF` statement which leads to two branch nodes. Each of these two nodes leads to multiple CDG nodes which are control dependent on the edges `S1---A` and `S1---Not A`.

b. Recursive Grouping

For each CDG node, the `extract_grp` function of our *recursive grouping algorithm* (in Fig. 31) returns the group sets of the current sub-colorCDG. Siblings are visited in control flow order. In our example, the grouping heuristic is applied in three places: S1, S2, S3. Since references in one CDG node have the same predicate, the `compute_local_grp` function needs only to put subscripts with the same base-pointer and access type into one group. `grp_union` does the same work as that of `compute_local_grp` except that the operators are groups of subscripts. When two groups with common elements (same subscript expressions) cannot be combined, their intersection (a set operation) is computed, This intersection operation can generate three new groups:

$$out\_group1 = group1 - group2$$

$$out\_group2 = group1 \cap group2$$

$$out\_group3 = group2 - group1$$

The access type and predicate properties of out_group1 and out_group3 remain those of group1 and group2. The access type and predicate properties of out_group2 are the union of those of group1 and group2.

c. Marking the Groups

In this step, compiler simply mark the groups where the first element of a group is referenced.

D. Shadow Structures for Sparse Codes

Many sparse codes traverse linked structures when processing their data structures. The referenced pointers can, in principle, take any value (in address space) and give the overall "impression" of being very sparse and random. For example, in SPICE 2G6, the device evaluation loops (in subroutine `LOAD` and its descendants, e.g., `BJT`) traverse linked lists and process C-like structures pointed to by each node in the list. Because the program does its own memory management out of a large statically allocated array, all pointers index into the same space (the code uses different array names but are overlaid). This makes the task of efficiently shadowing and representing memory references extremely difficult.

However a static analysis reveals a single-statement strongly connected component, a recurrence between address and data, that is initialized before the loop and whose values are used as indices in the loop body. It is of the form `loc = NODPLC(loc)`. Furthermore, we find more such recurrences in the loop body, with the difference that they are initialized within the loop.

After this type of static analysis, we speculate with a high degree of confidence that the code traverses a linked list and that the addresses it references are in some

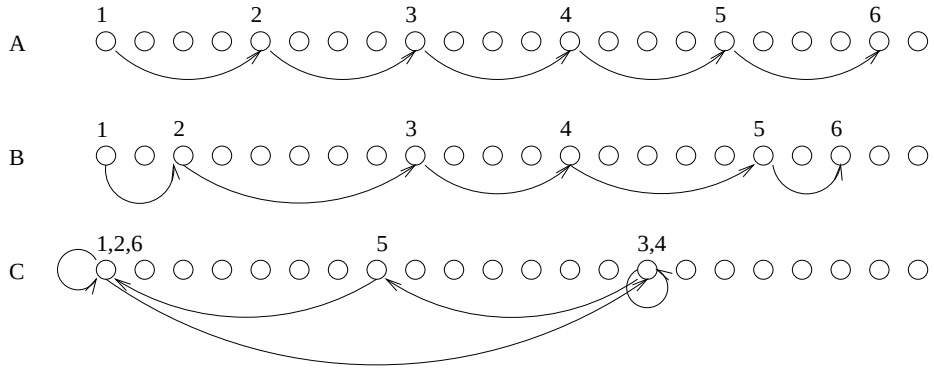


Fig. 34. Various irregular memory access patterns in a loop. (A) regular accesses indeed; (B) monotonic accesses with variable strides; (C) random accesses

"advantageous" order which is amenable to optimization.

We therefore identify the base-pointers used by the loop (the various names of overlaid names) and classify the accesses associated to them as:

1. monotonic accesses with constant stride;

2. monotonic accesses with variable stride;

3. random accesses.

Fig. 34 and 35 show examples of such accesses. For each of these possible reference patterns we have adopted a specialized representation.

- Monotonic constant strides can be represented by a closed-form descriptor and thus be recorded in a triplet [offset,stride,count].
- Monotonic addresses with variable stride can be recorded in an array with the additional fields [min,max] of their value.
- Random addresses can be stored in hash tables (if we expect a large number of them) or simple lists which are sorted later. Range information is also be maintained and recorded.

The run-time marking routines are adaptive, i.e., they verify the class of the access

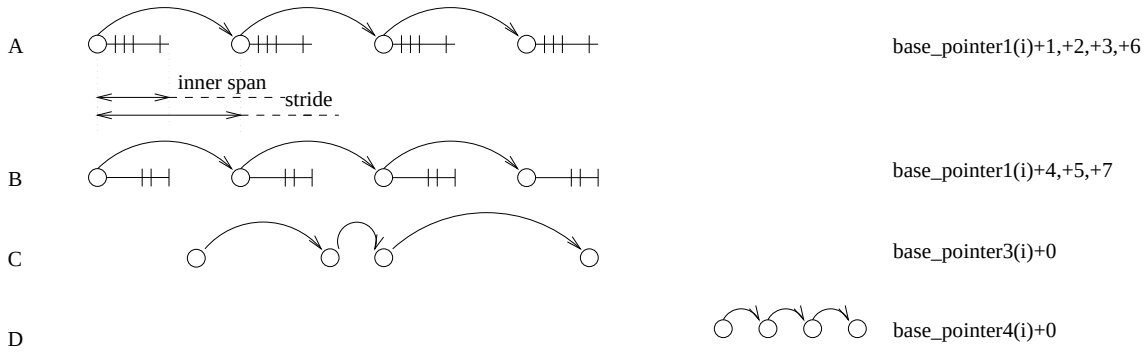


Fig. 35. Intersections of different run-time memory access regions. (A) and (B) have the same base-pointers, inner span, and stride. Actual array reference indices are different because they are in two groups. To verify no overlap between A and B, only check whether 'stride > inner span'; (A) and (C) have different base-pointers, and C doesn't have constant stride. To verify for no overlap between A and C, merge A and C and check for collisions; (C) and (D) To verify for no overlap between C and D, compare ranges.

pattern and use the simplest possible form of representation. Ideally, all references can be stored as a triplet, dramatically reducing space requirements. In the worst case, the shadow structures are proportional to the number of marked references. The type of reference, i.e., WF, RO, RW, and NO are recorded in a bit vector which is as long as the number of recorded references.

After loop execution, the analysis of the recorded references uses algorithms that range from the simplest to the most time consuming.

The test of the data dependence conditions is done by detecting whether pointers (and their associated groups, as defined in Section C) collide through the following hierarchical procedure:

- check for overlap of address ranges traversed by the base-pointers (linked lists) using min/max information;
- if there is overlap, check for collision of the shadow structures associated to a

pair of base-pointers.

We have implemented, in our run-time library, the comparisons for all the possible situations. The comparison between two triplets is done analytically. The comparisons among sorted lists and hash tables are through merging, which have linear complexity. The comparisons among triplets and sorted lists/hash tables also have linear complexity. If at any time a collision is detected, then the type of reference is read from the bit vector for that particular address and any possible data dependence will be detected.

This scheme uses shadow data structures that are, in general, more expensive (no random access) to access and analyze than the shadow arrays used in dense problems. However, if the speculation about the code's reference pattern is correct, then storage requirements are minimized and only inexpensive operations are performed. Of course, should the speculation fail, then the only advantage of this technique is its compact storage. As we show in the following section, we have devised reasonably accurate compile-time heuristics for a successful speculation.

E. Experimental Results

1. Run-time Overhead Reduction

We implemented the previously presented method of reducing marking points in a program through the grouping algorithm in the POLARIS compiler infrastructure [28].

The grouping algorithm has been implemented as part of our run-time parallelization pass, the last optimization/transformation step before the code generation pass in Polaris. We have run it on several important loops from the Perfect Benchmarks (SPICE 2G6, Ocean), SPEC (TFFT2), and a N-body code from NCSA (P3M).

TABLE IX
EFFECT OF THE AGGREGATION OF MARKINGS

Program	SPICE 2G6	P3M	OCEAN	TFFT2
Loop	BJT loop	PP_do100	FTRVMT do9109	CFFTZ do#1
#references under test	259	24	18	18
#markings w/o grouping	150	24	6	18
#markings w/ grouping	13	9	3	8
Reduction of static markings	91.3%	62.5%	50%	55%
Reduction of dynamic markings	84%	41%	50%	69%
Loop speedup ratio	1.46	1.54	1.21	1.69

Table IX summarizes the effects of our compile-time techniques that reduce the run-time tracing overhead. In the table, we give the number of references to the arrays under run-time test in the original code, the number of references that were marked in a previous implementation of the LRPD test (that already had some optimizations based on simple dominator relation between references), and the resulting static and dynamic counts of marking sites after applying the grouping technique. The "loop speedup ratio" is the ratio of the speedups of the speculative parallel loop obtained with and without applying our proposed techniques. The results show that the reductions of both static and dynamic counts of marking sites are significant in all cases and does indeed contribute to improved performance.

2. A Case Study: SPICE 2G6

We have chosen the loop in subroutine BJT of the program SPICE 2G6 as the target of our detailed experiment. This loop has an almost identical access pattern as most of the device evaluation steps and represents between 11% and 45% of the total execution time of the code. The SPICE 2G6 is a program with sparse memory accesses that offers us the opportunity to evaluate our grouping methods (which are also applicable to dense codes), the choice of shadow structures, and sparse reduction validation and

optimized parallelization (discussed in previous chapters of this dissertation).

The unstructured loop was first brought to a structured `DO` loop form (a separate pass we developed in Polaris). Then, through a different technique, we distributed the dominating recurrence outside the loop. This is in fact the loop containing the linked list traversal that controls the traversal of all data structures of the loop and has the form `LOC = NODPLC(LOC)`. This first loop is executed sequentially and all pointers are collected in a temporary array of pointers that is used by the remainder of the BJT loop (and has random access).

We then have used the run-time pass of the compiler to instrument the minimal number of reference groups for run-time marking. The loop invariant part of marked addresses was hoisted outside the loop and set up as an inspector loop. It represents the flow-insensitive traversal of all base-pointers (13 of them) that the loop can reference. These are the base-pointers of all marking groups. The predicates guarding their actual execution are loop variant and have to be left for marking inside the loop itself. The traversal and analysis of the inspector loop give us a conservative result whether there is any cross-processor collision (overlap) among the references. The shadow data structures used by our run-time library for reference tracing are *triplets* for 7 pointers, list of values for 3 other pointers and hash tables for the reduction operand addresses. Had our "guess" been incorrect, then our adaptive run-time library would have automatically "demoted" the triplets (for linked lists with constant, monotonic stride) to lists and then hash tables. The run-time library also collects range information on the fly (min/max values of specific base-pointers).

We generate 4 versions of the loop representing a combination of four situations:

1. conservative test (inspector) is sufficient to qualify the loop as parallel;

2. speculative execution is needed in order to mark the dynamic existence of the

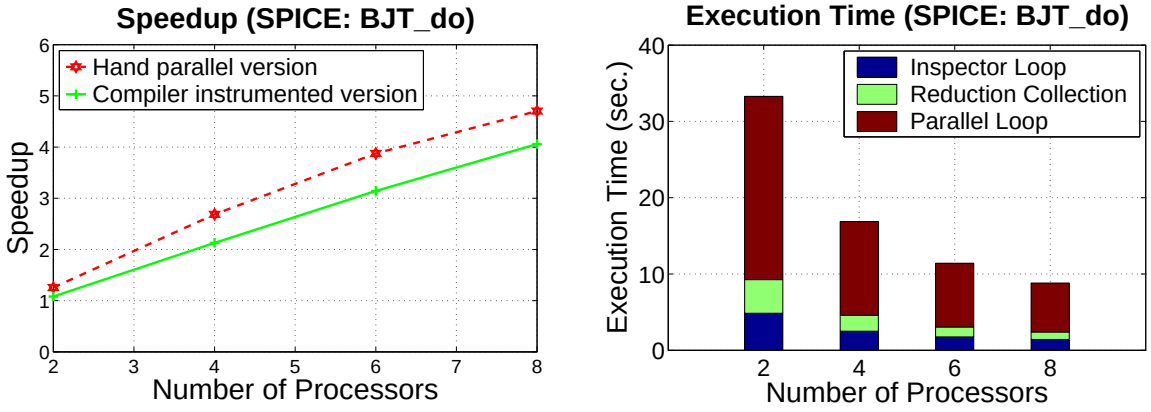


Fig. 36. Performance of SPICE BJT loop with input 1 (The input is extended from the input file of SPEC89-92. The sequential coverage of loop BJT is about 11%)

groups (based on the actual control flow) and qualify/disqualify the loop as parallel after execution;

3. the reduction parallelization needs to be verified (not described in this chapter);

4. the parallelization is known to be valid because it has been proven in a previous instantiation and no modification of addresses has been found in the outer loop.

Finally, we have instrumented (with the help of the same grouping algorithms) the remainder of the outer-loop containing calls to BJT to flag any modification of shared integer array variables (potential address modifications).

Depending on the dynamic situation, simple code generated by the compiler decides which version to run.

In our experiments with two different input sets, we had to run the conservative inspector and validate the reduction parallelization only 2 times when address modifications outside the BJT loop were flagged. (For the reduction validation, it was sufficient to show that the range of the reduction operand addresses did not overlap with the rest of the references.) The experimental setup for our speedup measure-

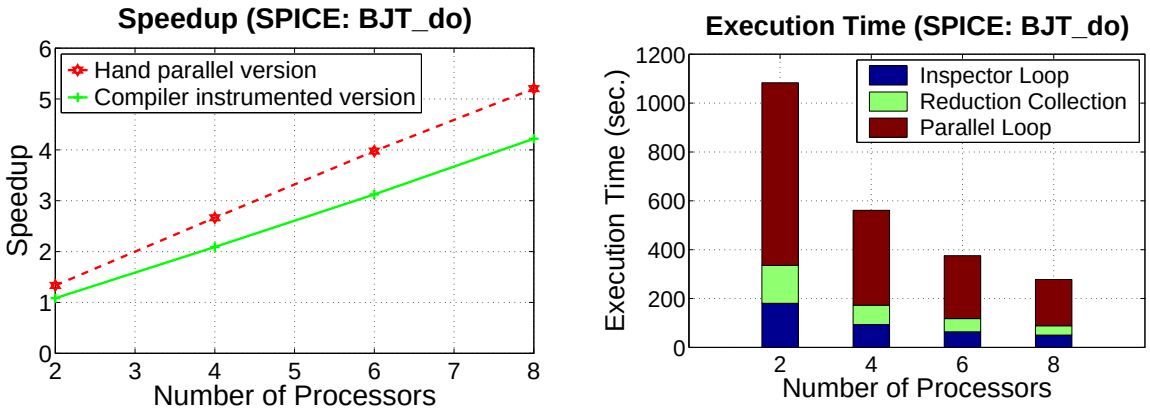


Fig. 37. Performance of SPICE BJT loop with input 2 (The input is implementing a 256-bits adder (extended from a 8-bits adder). The sequential coverage of loop BJT is about 31%)

ment consisted of a 16 processor HP-V class system with 4Gb memory, running the HPUX11 operating system.

In Fig. 36 and 37 we show the speedups obtained for two input sets. The graphs report speedup with no run-time checking but with our reduction parallelization and actual overall obtained speedup. The results show that obtained speedups scale up to 8 processors. We do not report numbers for a larger number of processors because our input set was fairly small. (Forking overhead is 5% of the serial time - very significant). Additional insights are presented, i.e., overall execution time breakdown.

F. Summary

This chapter presents compile-time and run-time techniques to increase the potential speedup and efficiency of parallel loops that parallelized via speculative run-time parallelization techniques. Great emphasis was put on efficiently applying run-time parallelization for sparse codes. The run-time library adaptively selects the most

profitable shadow data structures to minimize the run-time marking overhead and memory usage. The techniques presented complement the existing speculative run-time parallelization technique (the LRPD test) for efficiently parallelizing loops with sparse memory accesses.

The detailed case study, SPICE, is one of the most difficult codes, and our techniques proved themselves to be quite useful. We believe that other sparse codes will behave similarly or better. SPICE is an interesting case study because it requires all the methods presented in this chapter and – more importantly – it is most similar to the problems arising in C codes: memory management, extensive use of pointers, linked structure traversals, etc. By parallelizing SPICE we have shown that the techniques presented here are generally applicable. More specifically, we have gained valuable experience applicable to C programs.

CHAPTER VII

RELATED WORK

In this chapter, we review work related to our research. The techniques reviewed under "adaptive optimization," "reduction parallelization," and "automatic parallelization" are related to our adaptive algorithm selection framework, novel irregular parallel reduction algorithms, and adaptive run-time parallelization techniques, respectively.

A. Adaptive Optimization

There does not appear to be a great body of work in the area of adaptive selection of low level algorithms. Brewer [55] is probably the most extensive previous work aimed at making a framework for this decision process. In this approach, performance models are consulted to determine the expected running time of possible implementation, with the minimum running algorithm then chosen for use. These models are linear systems provided, along with some code annotations, by the end user. A benchmarking phase determines the coefficients of these models, and the entire process has been shown to be effective for both selecting among parallel sorting algorithms and determining data distribution for a parallel equation solver.

Li, Garzaran, and Padua [56] present an approach for choosing between several sequential sorting algorithms based on data size, data entropy, and an installation benchmarking phase that correctly selects the best algorithm for the given situation. However, no attempt is made to generalize the technique into a general approach and no discussion of the more difficult parallel case is given. There has been many previous works aim at tuning specific algorithm parameters. Examples are Spiral [57] and FFTW [9] for FFT signal processing, and ATLAS [10] for matrix multiplica-

tion. These approaches are very narrow (though quite effective) in scope and do not constitute a general framework for generic algorithm selection.

In practice, profiling is widely used to obtain information about the dynamic characteristics of a program [55, 58]. To enable profile-directed (feedback-directed) optimizations, many commercial compilers have options such as: `-fb` and `-fb_create` options of SGI compilers [59], `-qpdf1` and `-qpdf2` options of IBM AIX XL compilers [60], `+P` option of HP PA9000 `cc` compiler [61], etc. Profiling obtains coarse-grain dynamic information related to common program behaviors such as hot-spots and fine-grain dynamic information such as hardware counter values [62]. Most techniques involving profile-directed optimization are based on the assumption that the actual execution environment is similar to that of the profiling run(s); this limits the applicability of the approach.

Another somewhat relevant approach is that of *dynamic feedback* [63], which selects code variants based on on-line profiling, to try to overcome limitations on the applicability of off-line profiling. *Dynamic feedback* selects code variants based on the measured execution times of a few execution instances for all the candidate code variants during a sampling phase. The different code variants are generated at compile-time. For programs with periodic behaviors, our technique can select the best candidate optimization only after one execution instance of the code chunk (to be optimized); *dynamic feedback* usually requires several execution instances of the code chunk.

To put run-time optimization techniques in use systematically, optimizing compilers have been extended to generate multiple versions [64] or highly parameterized codes [65]. The limitations of these approaches are that multiple versioning could lead to code explosion, and parameterized code could slow down program execution. To overcome these limitations, dynamic compilation systems have been proposed. Dy-

namic compilation systems, with representative examples as DyC [66], Tempo [67], ADAPT [68], etc., generate optimized and specialized code variants during program execution and are therefore able to make use of run-time information, e.g., on-line code or value profiling information, to specialize code sections. Recently, a more specific approach of dynamic compilation systems – trace-based optimization systems (e.g., DYNAMO [69] and JAVA just-in-time compiler [70]), have been utilized in real-life applications. These systems apply efficient optimizations for programs' *hot–spots*, detected via on-the-fly profiling, and the performance gain is based on the frequent reuse of highly optimized code chunks.

Compared to optimizing compilers, dynamic compilation systems are believed to be a more general paradigm for optimizing dynamic programs. Some of them have been applied in improving the performance of sequential programs or internet-based applications. However, little effort has been undertaken for scientific applications, especially for irregular programs.

B. Reduction Parallelization

Reduction parallelization is a very effective optimization and many related techniques have been proposed in the literature. In Chapter III, we briefly discussed some fundamental work on reduction recognition [14, 17, 18] and parallelization [19, 14, 12, 13, 21, 22]. Here we discuss some recently developed irregular reduction parallelization techniques.

The "owner computes" method has been mostly employed in the parallelization of irregular codes. The most straightforward reduction parallelization technique applying the "owner computes" rule is the *data affiliated loop* proposed by Lin and Padua [22]. In this technique, each processor traverses all the iterations in the re-

duction loop and checks whether the reduction array entry (or entries) referenced in the current iteration has been assigned to it. The owned assignments are executed while the rest are simply skipped. This basic technique introduces two major overheads. First, every reduction statement in the loop is guarded by a predicate to check whether the reduction is operating on local data. Second, it executes every iteration even if the iteration does not contain any reduction operation on the local data.

Two techniques have been proposed to improve the *data affiliated loop* method. *local write* [20] generates a schedule to have each processor only go through the iterations that have more than one reduction on local data. While the intention is to reduce the replication ratio, the technique does not remove the predicate guarding each reduction statements. Slightly different from *local write*, *Data write affinity* [71] avoids both iteration replication and the predicates guarding the reductions introduced by the *data affiliated loop*. For loops containing multiple reduction statements, they introduce $P - 1$ synchronizations for group computations into non-conflicting phases to ensure mutual exclusion for reductions. Because of the synchronizations, the "owner computes" rule is violated and therefore its performance is not as good as that of *local write* [72]. Potential disadvantages of methods utilizing the "owner computes" rule are that they may heavily replicate unnecessary computations and load balance is not sustained (unlike "data replication" based methods).

Recently, Zoppetti and Agrawal [73] developed a technique to parallelize adaptive irregular reduction loops on a multi-threaded architecture. The technique follows a fixed iteration distribution and pipelines the reduction data across processors. The update of the data happens only when the data is moved in the processor where the corresponding reduction operation resides. The computation is divided into $k \times P$ stages, where `P` is the number of processors and k is a small constant. The paper shows results for k as 2 and 4. To avoid the synchronization overhead for machines

supporting the fine-grained multi-threaded program execution model, the technique overlaps computation and communication. These experimental results were obtained in a simulation environment. The potential drawback of this technique is associated to the computation "stages," which may introduce unnecessary communication (pipeline data sections to irrelevant computation threads) or load imbalance.

Adaptive Data Repository (ADR) infrastructure [74] was developed to perform range queries with user-defined aggregation operations on multi-dimensional datasets, which are generalized reductions. In the *ADR* infrastructure, three data aggregation strategies are used: fully replicated accumulation, sparsely replicated accumulation, distributed accumulation, which are analogous to RepBuf, SelPriv and LocalWr discussed in this paper. These experiments have shown that none of the three strategies worked best for various query patterns and that prediction models were desired.

C. Automatic Parallelization

It has long been realized that not only is programming parallel and distributed programs a tedious, error-prone task, but also that the performance of programs running on such systems is disappointing without good system-level support. The only avenue for bringing parallel processing to every desktop is to make parallel programming as easy as programming current uniprocessor systems. The traditional path to achieve this goal is through good programming languages, and mainly, through automatic compiler parallelization and optimization.

In scientific programs, the execution of loops dominates the overall performance of whole programs and therefore much effort has been concentrated on parallelizing loops. A loop can be executed in fully parallel form, without synchronization, if and only if the desired outcome of the loop does not depend in any way upon the

execution ordering of the data accesses from different iterations (or groups of iterations on different processors). In order to determine whether or not the execution order of the data accesses affects the semantics of the loop, the data dependence relations between the references to the same arrays in the loop body must be analyzed. The available techniques for array data dependence analysis include: Banerjee's test [75], Omega test [76], Range test [77], Interprocedural memory classification analysis [53] , etc.

While the flow-dependences express a fundamental relationship in the data flow of the program, anti- and output-dependences can be removed by *array privatization*. *Array privatization* identifies arrays that are used as temporary work spaces within a loop iteration and allocates private (temporary) copies of the arrays for each thread. Compile-time *array privatization* techniques [49, 51, 78] combine control-flow and data-flow information to query whether there are any read references to array elements which are not preceded by write references to the same elements.

Another important technique to enable parallelization is *reduction parallelization*. *Reduction parallelization* and its related work has already been discussed in detail (previously in Chapter III and here).

Putting many of these techniques together, researchers have lately developed integrated techniques [53, 79] and infrastructures [80, 81], which successfully parallelize more loops statically.

To complement static compiler parallelization techniques that explore the parallelism of dynamic and irregular applications, run-time parallelization techniques, postponing program analysis until program execution, have been applied. All run-time optimizations (in general) consist of at least two activities: (a) a test of a set of run-time values as a trace of all relevant memory accesses and (b) the execution of one of the compiler generated options. If the test phase is performed before the execution

of the loop and has no side effects, i.e., it does not modify the state of the original program variables (shared), then the technique is called *inspector/executor* [45] . Its run-time overhead consists only of the time to execute the inspection phase. If the test phase is done at the same time as the execution of the aggressively optimized loop and, in general, the state of the program is modified during this process, then the technique is called *speculative execution* [17, 82, 36]. Its associated overhead consists of the test itself and the saving of the program state (checkpointing). If the optimization test fails, extra overhead is paid during a program ante-loop state restoration phase before the conservative version of the code can be executed. In this scenario the initial optimized loop execution time becomes an additional overhead too.

The above run-time parallelization techniques have relatively large overhead because they analyze all the points referenced. Recently, researchers have adopted comprehensive compiler-time techniques that combine control-flow and data-flow analysis, to generate efficient run-time tests that decide whether the loops are parallel or not by evaluating a small set of run-time values [82, 83]. Although these techniques reduce run-time overhead dramatically, i.e., from proportional to the number of dynamic references to that proportional to the size of the data (or even constant), they are far from silver bullets. Therefore, comprehensive run-time test techniques (i.e., the LRPD test [17] and the adaptive techniques described in Chapter VI) are still needed to treat worst case scenarios.

CHAPTER VIII

CONCLUSIONS

Motivated by the fact that adaptive run-time optimization is the key for achieving sustainable performance for irregular applications running on today's high-performance systems, this dissertation has concentrated on run-time parallelization and optimization of adaptive/dynamic irregular applications. This has been identified as a particularly difficult task for programmers and limited success has been reached. In this chapter, we first summarize research presented in this dissertation and then briefly present some future directions that could follow from this dissertation.

A. Dissertation Research

The main contributation of this dissertation is a set of compiler and run-time techniques that can adaptively select and deploy algorithms or data structures that are most suited to a particular program execution instance.

1. Adaptive Algorithm Selection Framework

The Adaptive Algorithm Selection framework presented in this dissertation provides a systematic process for generating prediction models that can select, at run-time, the best performing, functionally equivalent algorithm for each of its execution instances. The *setup phase* occurs once for each computer system to implicitly tailor the process to a particular architecture. With pre-defined high-level characterization parameters and a synthetic experiment process, a mapping between different points in the parameter space and a relative performance ranking of the available algorithms is built and interpreted into algorithm selection codes. The *dynamic selection phase* occurs during actual program execution in order to collect the actual parameters and execute

pre-generated selection codes to select and deploy the the most suited algorithms or implementations.

Chapters III – V concentrate on specializing our adaptive framework to parallel reduction algorithm selection. The experimental results presented in Chapter V show that our framework can model program's irregular and dynamic behavior and customize solutions every time this is needed. Specifically, for reduction algorithm selection, the selected performance is within 2% of optimal performance and on average 60% better than "replicated buffer", the default parallel reduction algorithm specified by OpenMP standard [33]. In addition, the results show that the framework is portable and when applied for dynamic applications, it can achieve performance otherwise (e.g., applying only one algorithms) not possible.

2. Adaptive Run-Time Parallelization

The daptive run-time parallelization techniques presented in Chapter VI automatically detect and explore parallelism of loops in irregular programs with sparse memory accesses. With compile-time analysis and augmentation, the developed run-time library (with low run-time overhead) adaptively selects appropriate shadow data structures among: list, hash table, and closed-form representation to record the memory reference patterns executed by a loop. The techniques complement existing speculative run-time parallelization techniques (e.g., the LRPD test) for parallelizing loops with variant memory access characteristics.

B. Future Directions

Trends show that future computing platforms are quite likely to be comprised of parallel and/or distributed systems. Applications running on such platforms exhibit

dynamic behavior with respect to their computation and communication needs. We believe that adaptive optimization is generally an effective strategy to achieve a high level of performance for programs running on high-performance systems. This dissertation has revealed new improvements for achieving good performance on high-performance applications and systems. This section touches on research directions that are important and can be approached through extensions of this work.

1. Extension of Run-Time Parallelization for Other Programming Languages

At this point, the techniques described in this dissertation are implemented in Polaris, which is a research parallelizing compiler for Fortran 77. However, we believe that most of the techniques developed for modern, irregular Fortran 77 codes can be easily applied in Fortran 90, C, or C++. For example, for a C loop traversing a linked-list and processing structures associated to the nodes of the list, we can first traverse the linked list only and assign the addresses of the corresponding structures to a vector and then process the corresponding structures in parallel. While processing the structures in parallel, we can detect whether any contention on shared data or mutation on the linked list occurred.

2. High-Performance Libraries

It is fairly natural to extend the functionality of our adaptive algorithm selection framework for developing high-performance, domain-specific libraries. For example, a call to a library subroutine can be easily identified and replaced by the most appropriate optimized candidate library routine. We can implement an adaptive parallel container in a C++ generic programming platform (e.g., STAPL [8] – STandard Adaptive Parallel Library, a parallel counterpart of STL that is being developed by

our colleagues at Texas A&M University) to adaptively select a specific container among candidates such as vector, list, and tree, based on a program's dynamic memory access pattern.

3. Dynamic Compilation

The adaptive optimization technology described in this dissertation generates multiple versions at compile-time and adaptively selects among the generated versions at run-time. To avoid the potential code explosion introduced by multi-versioning and utilizing more powerful, systematic adaptive optimization, we would like to investigate techniques such as dynamic compilation, which optimize programs and generate new code at run-time.

REFERENCES

[1] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, 1995.

[2] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, and et al., *Gaussian 98, Revision A.11*, Gaussian, Inc., Pittsburgh PA, 2001.

[3] "CHARMM: A program for macromolecular energy, minimization, and dynamics calculations," *Journal of Computational Chemistry*, vol. 4, no. 6, p. 187, 1983.

[4] R. I. Klein, R. T. Fisher, C. F. McKee, and J. K. Truelove, "Gravitational collapse and fragmentation in molecular clouds with adaptive mesh refinement," in *Proceedings of the International Conference on Numerical Astrophysics 1998 (NAP1998)*, T. H. D. K. S.M. Miyama, K. Tomisaka, Ed., Tokyo, Japan, Mar. 1998.

[5] M. Mathis, N. M. Amato, and M. L. Adams, "A general performance model for parallel sweeps on orthogonal grids for particle transport calculations," in *Proceedings of the 14th ACM International Conference on Supercomputing (ICS'00)*, Santa Fe, NM, May 2000, pp. 255–263.

[6] G. A. Glatzmaier and P. H. Roberts, "A three-dimensional self-consistent computer simulation of a geomagnetic field reversal," *Nature*, vol. 377, pp. 203–209, 1995.

[7] H. Yu and L. Rauchwerger, "Adaptive reduction parallelization techniques," in *Proceedings of the 14th ACM International Conference on Supercomputing (ICS'00)*, Santa Fe, NM, May 2000, pp. 66–77.

[8] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger, "STAPL: An adaptive, generic parallel c++ library," in *Proceedings of the 14th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, Cumberland Falls, KY, Aug. 2001, pp. 193–208.

[9] M. Frigo, "SPL: A language and compiler for DSP algorithms," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*, Atlanta, GA, 1999, pp. 169–180.

[10] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–25, 2001.

[11] H. Yu, F. Dang, and L. Rauchwerger, "Parallel reduction: An application of adaptive algorithm selection," in *Proceedings of the 15th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, College Park, MD, July 2002, pp. 171–185.

[12] C. P. Kruskal, "Efficient parallel algorithms for graph problems," in *Proceedings of the 1986 International Conference on Parallel Processing (ICPP'86)*, University Park, PA, Aug. 1986, pp. 869–876.

[13] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. San Francisco, CA: Morgan Kaufmann, 1992.

[14] H. P. Zima, *Supercompilers for Parallel and Vector Computers*. New York, NY: ACM Press, 1991.

[15] M. Wolfe, *High Performance Compilers for Parallel Computing*. Boston, MA: Addison-Wesley, 1995.

[16] K. Kennedy and R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* San Francisco, CA: Morgan Kaufmann, 2001.

[17] L. Rauchwerger and D. A. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95)*, La Jolla, CA, June 1995, pp. 218–232.

[18] D. Patel and L. Rauchwerger, "Principles of speculative run–time parallelization," in *Proceedings of the 11th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'98)*, Chapel Hill, NC, Aug. 1998, pp. 330–351.

[19] R. Eigenmann, J. Hoeflinger, Z. Li, and D. A. Padua, "Experience in the automatic parallelization of four perfect benchmark programs," in *Proceedings of the 4th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'91)*, Santa Clara, CA, Aug. 1991, pp. 65–83.

[20] H. Han and C.-W. Tseng, "Improving compiler and run-time support for adaptive irregular codes," in *Proceedings of the 7th IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, Paris, France, Oct. 1998, pp. 393–400.

[21] W. M. Pottenger, "Theory, techniques, and experiments in solving recurrences in computer programs," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, May 1997.

[22] Y. Lin and D. A. Padua, "On the automatic parallelization of sparse and irregular fortran programs," in *Proceedings of the 4th International Workshop on Languages, Compilers and Run-time Systems for Scalable Computers (LCR'98)*, Pittsburgh, PA, May 1998, pp. 41–56.

[23] J. Wu, J. H. Saltz, S. Hiranandani, and H. Berryman, "Runtime compilation methods for multicomputers," in *Proceedings of the 1991 International Conference on Parallel Processing (ICPP'91), Vol. II - Software*, D. H. Schwetman, Ed. Boca Raton, FL: CRC Press, Inc., 1991, pp. 26–30.

[24] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Transactions on Computers*, vol. 40, no. 5, pp. 603–612, May 1991.

[25] R. von Hanxleden, "Handling irregular problems with Fortran D – a preliminary report," in *Proceedings of the 4th Workshop on Compilers for Parallel Computers (CPC'93)*, Delft, Netherlands, Dec. 1993, pp. 353–364.

[26] C. Ding and K. Kennedy, "Improving cache performance of dynamic applications with computation and data layout transformations," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*, Atlanta, GA, May 1999, pp. 229–241.

[27] Y.-S. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz, "Runtime and language support for compiling adaptive irregular programs on distributed-memory machines," *Software - Practice and Experience*, vol. 25, no. 6, pp. 597–621, 1995.

[28] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. A. Padua, Y. Paek, W. M. Pottenger, L. Rauchwerger, and P. Tu, "Advanced program restructuring for high-performance computers with Polaris," *IEEE Computer*, vol. 29, no. 12, pp. 78–82, Dec. 1996.

[29] W. F. van Gunsteren and H. J. C. Berendsen, "GROMOS: GROningen MOlecular Simulation software," Laboratory of Physical Chemistry, University of

Groningen, Nijenborgh, The Netherlands, Tech. Rep., 1988.

[30] D. R. O'Hallaron, J. R. Shewchuk, and T. R. Gross, "Architectural implications of a family of irregular applications," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-CS-97-189, Nov. 1997.

[31] H. Yu and L. Rauchwerger, "Run-time parallelization overhead reduction techniques," in *Proceedings of the 9th International Conference on Compiler Construction (CC'00)*, Berlin, Germany, Mar. 2000, pp. 232–248.

[32] F. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD test: Speculative parallelization of partially parallel loops," in *CDROM/Abstracts Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, FL, Apr. 2002.

[33] *OpenMP Fortran Application Program Interface, Version 2.0*, OpenMP Architecture Review Board, 2000. [Online]. Available: http://www.openmp.org

[34] R. Jain, *The Art of Computer Systems Performance Analysis.* Hoboken, NJ: John Wiley & Sons, Inc., 1991.

[35] A. Miller, *Subset Selection in Regression (Second Edition).* Boca Raton, FL: Chapman & Hall/CRC, 2002.

[36] D. Patel and L. Rauchwerger, "Implementation issues of loop-level speculative run-time parallelization," in *Proceedings of the 8th International Conference on Compiler Construction (CC'99)*, Amsterdam, The Netherlands, Mar. 1999, pp. 183–197.

[37] T. Autrey and M. Wolfe, "Initial results for glacial variable analysis," in *Proceedings of the 9th Annual Workshop on Languages and Compilers for Parallel*

Computing (LCPC'96), San Jose, CA, Aug. 1996, pp. 120–134.

[38] "PARASOL Systems," PARASOL Lab, Department of Computer Science, Texas A&M University, College Station, TX. [Online]. Available: http://parasol.tamu.edu

[39] "agave — IBM Regatta p690," Texas A&M University Supercomputing Facility, College Station, TX. [Online]. Available: http://sc.tamu.edu

[40] L. Oliker and R. Biswas, "Parallelization of a dynamic unstructured application using three leading paradigms," in *Proceedings Supercomputing '99 (CDROM)*, Portland, OR, 1999, p. 39.

[41] J. G. Castanos and J. E. Savage, "Repartitioning unstructured adaptive meshes," in *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, Cancun, Mexico, May 2000, pp. 823–832.

[42] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings," *International Journal of Parallel Programming*, vol. 29, no. 3, pp. 217–247, 2001.

[43] S. Turek and C. Becker, *FEATFLOW: Finite Element Software for the Incompressible Navier-Strokes Equations, User Manual, Release 1.1*, University of Heidelberg, Institute for Applied Mathmatics, Heidelberg, Germany, Feb. 1998.

[44] R. Iyer, N. M. Amato, L. Rauchwerger, and L. Bhuyan, "Comparing the memory system performance of the HP V-Class and SGI Origin 2000 multiprocessors using microbenchmarks and scientific applications," in *Proceedings of the 13th ACM International Conference on Supercomputing (ICS'99)*, Rhodes, Greece, June 1999, pp. 339–347.

[45] H. Berryman and J. H. Saltz, "A manual for PARTI runtime primitives," ICASE, NASA Langley Research Center, Hampton, VA, Interim Report 90-13, 1990.

[46] L. Rauchwerger, "Run–time parallelization: A framework for parallel computation," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, Sept. 1995.

[47] L. Rauchwerger, N. M. Amato, and D. A. Padua, "A scalable method for run-time loop parallelization," *International Journal of Parallel Programming*, vol. 26, no. 6, pp. 537–576, July 1995.

[48] L. Rauchwerger and D. A. Padua, "Parallelizing WHILE loops for multiprocessor systems," in *Proceedings of the 9th International Parallel Processing Symposium (IPPS'95)*, Santa Barbara, CA, Apr. 1995, pp. 347–356.

[49] P. Tu and D. A. Padua, "Automatic array privatization," in *Proceedings of the 6th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'93)*, Portland, OR, Aug. 1993, pp. 500–521.

[50] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Data dependence and data-flow analysis of arrays," in *Proceedings of the 5th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'92)*, New Haven, CN, Aug. 1992, pp. 434–448.

[51] Z. Li, "Array privatization for parallel execution of loops," in *Proceedings of the 6th ACM International Conference on Supercomputing (ICS'92)*, Washington, D.C., July 1992, pp. 313–322.

[52] L. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Ph.D. dissertation, University of California at Berkeley, Berkeley, CA, May 1975.

[53] J. Hoeflinger, "Interprocedural parallelization using memory classification analysis," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, Aug. 1998.

[54] Y. Paek, J. Hoeflinger, and D. A. Padua, "Simplification of array access patterns for compiler optimizations," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, June 1998, pp. 60–71.

[55] E. A. Brewer, "High-level optimization via automated statistical modeling," in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, Santa Barbara, CA, Aug. 1995, pp. 80–91.

[56] X. Li, M. J. Garzaran, and D. A. Padua, "A dynamically tuned sorting library," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, CA, Mar. 2004, pp. 111–122.

[57] J. Xiong, J. Johnson, R. Johnson, and D. A. Padua, "SPL: A language and compiler for dsp algorithms," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*, Snowbird, UT, 2001, pp. 298–308.

[58] M. Mock, C. Chambers, and S. J. Eggers, "Calpa: a tool for automating selective dynamic compilation," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO33)*, Monterey, CA, Dec. 2000, pp. 291–302.

[59] *MIPSpro Compiling and Performance Tuning Guide*, Silicon Graphics, Inc., 1999.

[60] *XL Fortran for AIX User's Guide, Version 8*, IBM Corp., 2002.

[61] *HP C/HP-UX Programmer's Guide, Ninth Edition*, Hewlett-Packard Company, 2000.

[62] N. M. Amato, J. Perdue, A. Pietracaprina, G. Pucci, and M. Mathis, "Predicting performance on SMPs. a case study: The SGI power challenge," in *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, Cancun, Mexico, May 2000, pp. 729–737.

[63] P. C. Diniz and M. C. Rinard, "Dynamic feedback: An effective technique for adaptive computing," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI'97)*, Las Vegas, NV, May 1997, pp. 71–84.

[64] M. Byler, J. Davies, C. Huson, B. Leasure, and M. Wolfe, "Multiple version loops," in *Proceedings of the 1987 International Conference on Parallel Processing (ICPP'87)*, St. Charles, IL, Aug. 1988, pp. 312–318.

[65] R. Gupta and R. Bodik, "Adaptive loop transformations for scientific programs," in *Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing(SPDP'95)*, San Antonio, TX, Oct. 1995, pp. 368–375.

[66] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad, "Fast, effective dynamic compilation," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI'96)*, Philadelphia, PA, May 1996, pp. 149–159.

[67] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi,

"Tempo: specializing systems applications and beyond," *ACM Computing Surveys*, vol. 30, no. 3es, p. 19, 1998.

[68] M. J. Voss and R. Eigenmann, "High-level adaptive program optimization with ADAPT," in *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, Snowbird, UT, June 2001, pp. 93–102.

[69] V. Bala, E. Duesterwald, and S. Banerjia, "DYNAMO: a transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, Vancouver, BC, Canada, May 2000, pp. 1–12.

[70] M. P. Plezbert and R. K. Cytron, "Does 'just in time' = 'better late than never?'," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, Paris, France, Jan. 1997, pp. 132–145.

[71] E. Gutierrez, O. G. Plata, and E. L. Zapata, "A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors," in *Proceedings of the 14th ACM International Conference on Supercomputing (ICS'00)*, Santa Fe, NM, May 2000, pp. 78–87.

[72] H. Han and C.-W. Tseng, "A comparison of parallelization techniques for irregular reductions," in *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, CA, Apr. 2001, p. 27.

[73] G. M. Zoppetti, G. Agrawal, and R. Kumar, "Compiler and runtime support for irregular reductions on a multithreaded architecture," in *CDROM/Abstracts*

Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, FL, Apr. 2002.

[74] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz, "Querying very large multi-dimensional datasets in ADR," in *CDROM Proceedings of the 1999 ACM/IEEE Conference on Supercomputing — SC'99 (Conference on High Performance Networking and Computing)*, Portland, OR, Nov. 1999.

[75] U. Banerjee, *Dependence Analysis for Supercomputing.* Norwell, MA: Kluwer Academic Publishers, 1988.

[76] W. Pugh, "The Omega test: A fast and practical integer programming algorithm for dependence analysis," in *Proceedings Supercomputing '91*, Albuquerque, NM, Nov. 1991, pp. 4–13.

[77] W. Blume and R. Eigenmann, "The Range test: A dependence test for symbolic, non-linear expressions," in *Proceedings Supercomputing '94*, Washington D.C., Nov. 1994, pp. 528–537.

[78] J. Gu, Z. Li, and G. Lee, "Experience with efficient array data flow analysis for array privatization," in *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'97)*, Las Vegas, NV, June 1997, pp. 157–167.

[79] Y. Lin and D. A. Padua, "Compiler analysis of irregular memory accesses," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, Vancouver, BC, Canada, June 2000, pp. 157–168.

[80] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and

M. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, vol. 29, no. 12, pp. 84–89, Dec. 1996.

[81] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. A. Padua, P. Petersen, W. M. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, "Polaris: The next generation in parallelizing compilers," in *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'94)*, Ithaca, NY, Aug. 1994, pp. 141–154.

[82] M. Gupta and R. Nim, "Techniques for speculative run-time parallelization of loops," in *CDROM Proceedings of the 1998 ACM/IEEE Conference on Supercomputing — SC'98 (Conference on High Performance Networking and Computing)*, San Jose, CA, Nov. 1998.

[83] S. Rus, L. Rauchwerger, and J. Hoeflinger, "Hybrid analysis: Static & dynamic memory reference analysis," in *Proceedings of the 16th ACM International Conference on Supercomputing (ICS'02)*, New York, NY, June 2002, pp. 274–284.

VITA

Hao Yu was born on December 22, 1972 in Datong, Shanxi Province, PR China. In 1989, he entered Tsinghua University and received his B.S. and M.S. degrees in Computer Science in 1994 and 1997, respectively. He began pursuing a Ph.D. degree in Computer Science at Texas A&M University in 1997. Since then, he has worked as a graduate research assistant for Dr. Lawrence Rauchwerger and in the Texas A&M Supercomputing Center.