An Adaptive Algorithm Selection Framework

Hao Yu, Dongmin Zhang, Francis Dang, and Lawrence Rauchwerger *
{h0y8494,dzhang,fhd4244,rwerger}@cs.tamu.edu

Technical Report TR04-002
Parasol Lab
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

March 01, 2004

Abstract

Irregular and dynamic memory reference patterns can cause performance variations for low level algorithms in general and for parallel algorithms in particular. We have previously shown that parallel algorithms (reductions, sorting) are input sensitive and can benefit from adaptation to their inputs and environment. Here we present an *Adaptive Algorithm Selection Framework* which can collect and interpret the inputs of a particular instance of a parallel algorithm and select the best performing one from a an existing library. In this paper we use our framework to select dynamically the best parallel reduction algorithm. We first introduce a set of high-level parameters that can model and rank the performance of different parallel reduction algorithms. Then we describe an off-line, systematic process to generate predictive models which can be used for run-time algorithm selection. A compiler generated instrumentation code dynamically characterizes a reduction loop's reference pattern and selects the most appropriate method for parallelizing it. Our experiments show that our framework: (a) Selects the most appropriate algorithms in 85% of the cases studied (b) When the best possible algorithm was not selected the performance is still within 2% of the optimal scheme's performance. (c) Adaptively selects the best algorithms for each phase of a dynamic program resulting in better overall performance (d) Adapts to the underlying machine architecture. (tested on IBM Regatta and HP V-Class).

*Research supported in part by NSF CAREER Awards CCR-9624315 and CCR-9734471, NSF Grants ACI-9872126, EIA-9975018, EIA-0103742, and by the DOE ASCI ASAP program grant B347886.

1 Introduction

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 of detecting and exploiting 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 the cases when performance is independent of the input data of the applications. When codes are irregular (memory references are irregular) and/or dynamic (change during the same program instance execution) it is very likely that important performance affecting program characteristics are input and environment dependent. Many important (frequently used and time consuming) algorithms used in such programs are indeed input dependent. We have previously shown [31] that, for example, parallel reduction algorithms are quite sensitive to their input memory reference pattern and system architecture. In [3] we have shown that parallel sorting algorithms are sensitive to architecture, data type, size, etc. 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. In [31] and [3] we have shown that performance can be significantly improved if we can select dynamically the best suited algorithm for each program instance.

In this paper, we present a general framework to automatically and adaptively select, at run-time, the best performing, functionally equivalent algorithm for each of their instantiations. The adaptive framework can select, at run-time, the best parallel algorithm from an existing library. The selection process is based on an off-line automatically generated prediction model and algorithm input data collected and analyzed dynamically. For this paper we have concentrated our attention on the automatic selection of reduction algorithms. For brevity we will not show the dynamic selection of sorting.

Reductions (aka updates) are important because they 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. More formally, a reduction variable is a variable whose value is used in one associative and possibly commutative operation of the form $x = x \otimes exp$, where $\otimes$ is the operator and x does not occur in exp or anywhere else in the loop. 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 [14, 16]. In the case of irregular, sparse programs, we define *irregular reductions* as reductions performed on data referenced in an irregular pattern, usually through an `index` array. For these *irregular reductions*, the access pattern traversed by the sequential reduction in a loop is often sensitive to the input data and/or computation. Therefore, as we have shown in [31], not all parallel reduction algorithms and/or implementations are equally suited as substitutes for the original sequential code. Each access pattern has its own characteristics and will best be parallelized with an appropriately tailored algorithm.

In [31] we have presented a small library of parallel reduction algorithms and shown that the best performance can be obtained only if we dynamically select the most appropriate one for the instantiated input set (reference pattern). Also in [31] we have presented a taxonomy of reduction reference patterns and sketched a decision tree based scheme (the tree was built by hand).

We now continue this work and introduce a systematic and automatic process to generate predictive models that match the parallel reduction algorithms to execution instances of reduction loops. After establishing a small set of parameters that can characterize irregular memory reference patterns and setting up a library of parallel reduction algorithms ([31], we measure their relative performance for a number of memory reference parameters in a factorial experiment. This is achieved by running a synthetic loop which can generate reduction references with the memory reference patterns selected by our 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 our previously extracted map).

This paper's main contribution is a framework for 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, parallel reductions being just on important example. The same approach could also be used for various other compiler transformations that cannot be easily analytically modeled. Our experiments an IBM Regatta and HP V-Class show that our framework: (a) selects the best performing algorithms in 85% of the cases studied; (b) when the best possible algorithm was not selected performance was still within 2% of the optimal scheme's per-

formance; (c) achieves better performance by adaptively selecting the best algorithm for each phase of a dynamic program; (d) adapts to the underlying machine architecture.

2 Framework Overview

In this section we give an overview of our general framework for adaptive and automatic low level algorithm selection, the details of which are presented in the remainder of this paper 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. More 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 algorithm choice 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.

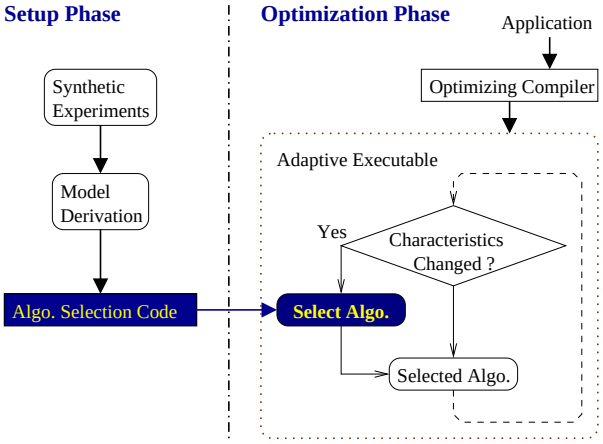

Figure 1: Overview of Adaptive Algorithm Selection.

The **setup phase** occurs once for each computer system and thus, implicitly tailors our process to a particular architecture. We then establish the domain of our inputs and the domain of possible optimizations.

In the particular case presented in this paper, i.e., parallel reductions, the **input domain** is the universe of all possible and realistic memory reference patterns – because, as previously shown [31], they crucially impact the obtained performance. Architecture type is also important but is used implicitely. 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 it. The domain of possible optimizations is composed, in our case, by the different parallel reduction algorithms collected in a library.

We then need to explore our input domain and find a mapping to the outcome domain. In our case we establish a mapping between different points in the input parameter space (memory reference patterns) and a relative performance ranking 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 of covering our input domain. For each of these data points we measure, on the particular architecture, the relative performance of our algorithms and rank them accordingly.

We should mention here that we have also tried other methods of exploring the data space. In [3] we have used a machine learning algorithm to explore the input space that defines the performance of sorting algorithms.

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 used the information obtained during the setup phase to find its corresponding output. In our case we use a statistical regression which will eventually select the appropriate best performing parallel reduction algorithm. In [3] we used a statically generated decision tree which is dynamically traversed to select the best sorting algorithm.

In this paper we specialize our adaptive framework to parallel reduction algorithm selection. In the next sections we present our library of reduction algorithms and a set of parameters that can characterize an irregular memory reference pattern. Then we describe the factorial experiment that explores our input space. For each parameter set we will execute a synthetic parameterized loop that generates a memory reference pattern on which we can evaluate and rank the different algorithms in our library. A regression method is used to compute predictive 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 operaton in question and use them in our pre-computed models to select the best algorithmic option. We evaluate our framework with experimental results for static and dynamic reference patterns.

3 Reduction Algorithm Library

Our parallel reduction algorithm library contains several methods suitable for a range of reference patterns. For brevity, we provide here only a high level description of the methods. See [31, 29] and other cited references for

additional details.

3.1 Parallel Reduction Algorithms

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 loop execution, and *private accumulation and global update methods*, which accumulate in private storage during loop execution and update shared variables with each processor's contribution afterwards. Direct update methods include the classical *recursive doubling* [14, 16], *unordered critical sections* [9, 32] and *local write* [11]; our library only includes local write because the others are not competitive for arrays. Private accumulation methods in our library include *replicated buffer* [14, 16, 24, 18], *replicated buffer with links* [31], and *selective privatization* [31].

Replicated Buffer (REPBUF). The loop executes as a `doall`, accumulating partial results in private reduction variables, and later accumulating results across processors.

Replicated Buffer With Links (REPLINK). To avoid traversing the unused (but allocated) elements during REPBUF's cross-processor reduction phase, a linked list records the processors that access each shared element.

Selective Privatization (SELPRIV). SELPRIV only privatizes array elements that have cross-processor contention. By excluding unused privatized elements, SELPRIV maintains a dense private space where almost all elements are used. Since the private space does not align to the shared data array, the reduction's index array is modified to redirect accesses to the selectively privatized elements.

Local Write (LOCALWR). As originally described in [11], LOCALWR first collects the reference pattern in an *inspector loop* [25] and partitions the iteration space based on the "owner–computes" rule. Memory locations referenced on multiple processors have their iterations replicated on those processors as well. To execute the reduction(s), each processor goes through its local copies of the iterations containing the reduction(s) and operating on the processor's local data.

3.2 A Qualitative Comparison

In Table 1, we present a qualitative comparison of the algorithms described above.

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 temporary and spatial locality. With respect to overhead, REPLINK, SELPRIV and LOCALWR all have auxiliary data structures that depend on the access pattern, and which must be updated when it changes. Their overhead costs can be reduced with schedule reuse [25].

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`

Table 1: Qualitative comparison of parallel reduction algorithms. `M` is the number of iterations; `N` is the size of the reduction array; `P` is the number of processors.

4 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. Below, we first define the identified parameters. We 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.

4.1 The Parameters

Below, we enumerate the parameters in no specific order.

N is the *number 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 same access pattern; here **N** includes the 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 [11]; there, the underlying data structures (corresponding to the irregular reductions) represent graphs $G = (V, E)$ and the authors defined **Connectivity** 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

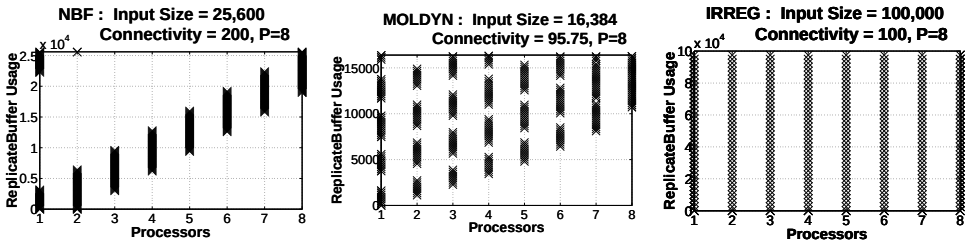

Figure 2: Memory access patterns of *Replicated Buffer* parallel reduction algorithm.

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 instrumented a parallel loop transformed for the *Replicated Buffer* algorithm using light-weight timers ($\sim$ 100 clock cycles).

SP, the *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**N**). Intuitively, **SP** indicates if *Replicated Buffer* is efficient.

CLUS, the *Number of Clusters*, reflects spatial locality and measures if the used private elements in the *Replicated Buffer* are scattered or clustered on each processor. Fig. 2 shows three memory access patterns which can be classified as *clustered*, *partially-clustered*, and *scattered*, respectively. 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.

4.2 Decoupled Effects of The Parameters

We have 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. The trends summarized in Table 2 are based on a comprehensive set of experiments on multiple architectures that are reported in [30, 29]. The trends for REPLINK are similar to REPBUF and are not listed here.

The effect of **N** is straight–forward, comparing 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 will indicate 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.

parameters	REPBUF	SELPRIV	LOCALWR
N	↗	↑	↑
CON	↑	↗	–
MOB	↑	↘	↓
OTH	↑	↑	↗
SP	↗	↓	↘
CLUS	↗	↑	–

Table 2: Summary of the decoupled effects of the characteristic parameters: on the performance of different reduction parallel algorithms. (↑: positive effect; ↓: negative effect; ↗: little positive effect; ↘: little negative effect; –: no effect.)

Large **MOB** values (many 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, and for LOCALWR, because large **MOB** values often result in high iteration replication. Large values of **OTH** often indicate good performance for REPBUF and SELPRIV because the first private accumulation loop has a larger iteration body; since LOCALWR replicates "other work" it will not benefit as much.

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

5 Adaptive Reduction Selection

In this section we will elaborate on the **setup** and on the **dynamic selection** phases of our adaptive scheme. The **setup** 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 now explain how these methods have to change in order to optimize dynamic programs, i.e., codes that change their characteristics during execution.

To reduce overhead 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 instantiation we can directly reuse the previously selected algorithm, thus saving run-time selection overhead. This is accomplished with standard compiler technology.

5.1 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. 4.

The domain of values that can be taken by the input parameters is explored by setting up a factorial experiment [13]. 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 of the different parameters. The chosen parameter values for our reduction experiment are shown in Table 3.

To measure the performance of different reduction patterns, we have created a synthetic reduction loop. The structure of the loop is shown in Fig 5, with C-like pseudocode. 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 with an additional pre–processing step. The dynamic pattern depends strictly on the `index` array, which is generated automatically to correspond to the parameters **SP** and **CLUS**. *Other Work* is a dynamically measured parameter and represents the ratio between the time spent performing reductions and the rest of the loop. The contents of the `index[*]` array are generated via a randomized process which satisfies the requirement specified by the parameters.

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. 4.

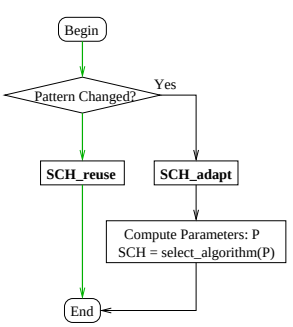

Figure 3: Adaptive Reduction Parallelization at Run-time.

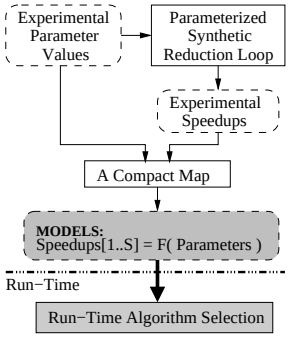

Figure 4: Setup phase of Adaptive Algorithm Selection.

```
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;
```

Figure 5: Parameterized Synthetic Reduction Loop

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		

Table 3: Values of parameters for the factorial design

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 *model* that can then be used by an application at run-time. We applied two methods, *general linear regression* and *decision tree learning*. However, due to space constraints, in this paper we will describe only the modeling process using *general linear regression*.

As illustrated in Fig. 4, 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 [20]. We have specified a maximal 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 maximal model.

The samples are first separated into training data and testing data. When adding a term into the model, the train-

ing data are used to fit (least square fit) the coefficients and the fitted model is evaluated using the testing data. If the test error is higher than the test error 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 its automatically generated models have produced very 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 experimental samples with less synthetic experimentation time and avoiding the term explosion. For parameter **MOB**, we have only chosen 2 values for the experiment, we have excluded the terms containing a non-linear **MOB** factor from the maximal model. For **OTH** and **CLUS**, though we specified few values, the values used in the map are measured and computed from the generated `index` array, respectively.

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.

5.2 Dynamic Selection Phase

During the dynamic phase, which is executed every time a reduction loop is instantiated (unless results can be reused because the memory reference characteristics have not changed) a compiler generated routine (the prediction code) will analyze the collected memory reference parameters and select the appropriate algorithm. This is accomplished by letting the pre-generated model evaluation routines estimate speedups of all algorithms, rank them and select the best one. The evaluation of the polynomial models is fast as each of final models contains only about 30 terms.

5.3 Selection Reuse for Dynamic Programs

In the previous section we presented a systematic process to generate predictive models that could recommend the best irregular reduction parallelization algorithm by collecting a set of static and dynamic parameters. We mentioned that if the memory characteristics parameters do not change we can reuse our decision and thus reduce runtime 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 will 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 the access pattern of the irregular reductions changes at instance t and the next change of the 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 *re-usability* 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 *re-usability* instantiations.

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

In the following we will attempt to include *re-usability* into our previously developed models, predict the 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.

The phase–wise speedup of a parallel algorithm can be formulated as $\frac{(R-1)T_{par}+(1+O)T_{par}}{T_{seq}}$). The best algorithm is the algorithm that has the smallest value of $\frac{R+O}{Speedup}$. In above formulas, R is the *re-usability*, 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 experimental process described in previous sections, we have generated models to compute the O (setup overhead ratio). Together with the predicted speedup (excluding the setup overhead), we were able to evaluate $\frac{R+O}{Speedup}$ at run–time and select the best scheme for a dynamic execution phase.

We estimate the *re-usability* **R** at the beginning of each phase by a regression model on the previously collected values of **R**. Alternatively we could use profiling information.

To further reduce the modeling overhead, instead of applying the default (usually slow) scheme, *Replicated Buffer*, we simply instrument parameter collection codes on the fast parallel algorithms, e.g., *Selective Privatiza-*

Program	Lang.	Lines	Source	Loop	Coverage	# inp
Irreg – CFD kernel using FE methods	F77	223	[11]	apply elements' forces (do 100) – traverse elements	∼ 90%	4
Nbf – MD kernel (from GROMOS)	F77	116	[11], [26]	non-bounded force calc. (do 50) – traverse neighbor list	∼ 90%	4
Moldyn – synthetic MD program	C	688	[11], [7], [19]	non-bounded force calc. – traverse interaction list	∼ 70%	4
Charmm – MD kernel (from CHARMM)	F77	277	[12]	non-bounded force calc. – traverse neighbor list	∼ 80%	3
Spark98 – unstructured FE simulation	C	1513	SPEC'2000	smvp loop – Symmetric MV Product	∼ 70%	2
Spice – circuit simulation	F77	18912	SPEC'92	bjt loop – traverse BJT devices, update circuit mesh	11–45%	4
Fma3d – 3D FE method for solids	F90	60122	SPEC'2000	scatter_element_nodal_forces_platq loop – scatter forces	∼ 30%	1

Table 4: Scientifi c Applications and Reduction Loops.

tion and *Adaptive Local Write*. This way, we do not have to switch back to the *Replicated Buffer* scheme to collect the parameters when the pattern changes, which reduces the execution time for the loop instances where pattern changes.

6 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 predictive models with the other algorithms.

6.1 Experimental Setup

We selected seven programs from a variety of scientifi c domains (see Table 4). All Fortran codes were automatically instrumented to collect information about the access pattern using the Polaris compiler [4]; the C programs were done manually. For most programs, we chose or specifi ed multiple inputs to explore input-dependent behavior. While specifying the inputs, *we have tried, where possible, to use data sets that exercise the entire memory hierarchy of our parallel machine.*

We have studied two parallel systems: an UMA HP V-Class with 16 processors [1] and a NUMA IBM Regatta p690 system with 32 processors [2]. The machine confi gurations are briefly described in Table 5. In the IBM Regatta system, each POWER4 chip contains 2 processors and a multi-chip module (MCM) contains 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 signifi cantly 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 and a 16-processor subsystem of the IBM, which was suffi cient for us to exercise the architectural characteristics of these two systems. Additionally, while we ran all 22 application/input combinations on the HP, due to limitations on our single user time allocation, we did not obtain results for `FMA3D` on the IBM, and so we only show results on the IBM for 21 application/input cases.

	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

Table 5: Target machine confi gurations.

6.2 Adaptive Algorithm Selection Results

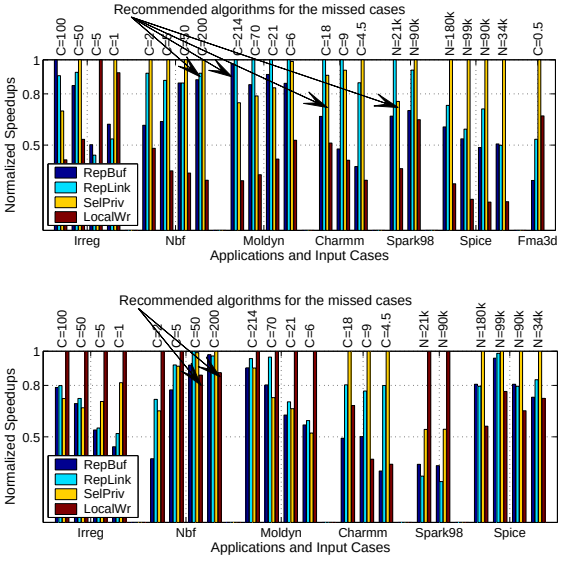

Figure 6: Relative Performance of Parallel Reduction Algorithms. Obtained on HP V-Class with P=8, and IBM Regatta p690 with P=16, respectively.

Fig. 6 presents the results obtained on the HP V-Class

system and the IBM Regatta 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 in 18/22 cases on the HP and 19/21 cases on the IBM. As the arrows in the graphs show, in all the mis-predicted cases the regression model identified the second best algorithm, and moreover, there was little performance difference between the best and second best algorithm in all such cases.

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 have compared the effectiveness of our predictive 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 in `OpenMP 2.0` [22].
- **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 *Selective Privatization*(SELPRIV) and *Local Write*(LOCALWR) as the default algorithms for the HP and IBM systems, respectively.

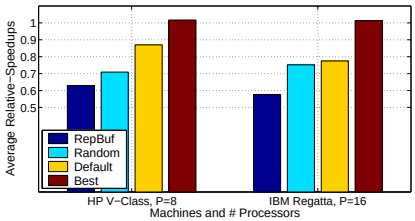

Figure 7: Average relative-speedups.

Fig. 7 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 *speedup ratio* value, the better the relative effectiveness of our predictive models. Here, the comparisons show that our automatically generated predictive 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 predictive models can improve the performance of irregular reductions significantly.

As mentioned in Fig. 3, our instrumentation introduces extra computation in the *Replicated Buffer* algorithm to measure the parameters, **OTH**, **SP** and **CLUS**. We have measured the run-time overhead (normalized to the *Replicated Buffer* 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 in 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, respectively, 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. 6, especially for the bars corresponding to the results for `Irreg` and *Charmm* on the HP system, the best schemes for different inputs change and our technique predicts the best schemes correctly by collecting and utilizing the dynamically collected parameters.

6.3 Validation for Regular Reduction

To test the robustness and generality of our predictive models, we have 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 relative small and every element of the vector is 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 predictive models have given the correct recommendations. Hence, our *adaptive algorithm selection* technique is generally applicable to all reductions.

7 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(s) dynamically and thereby improve the overall performance of adaptive irregular programs. While adaptive and dynamic computation are widely used in scientific programs, there is no such benchmark program available to us. Therefore, we have used two synthetic programs which were derived from heavily used applications and algorithms. In this section, `DynaSel` represents applying *Adaptive Algorithm Selection* dynamically for every computation-phase (described in Section 5.3). Our experiments were carried out on the 8-processor subsystem of our HP V-class.

7.1 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)* [21, 6]. Fig. 8 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 doesn't 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.
```

Figure 8: High level description of `AmrRed2D`.

We numbered 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 renumbered. Each mesh adaptation (refinement and coarsening) indicates that start of a new dynamic computation phase. In terms of data distribution, we originally distribute the nodes in a blocked manner (each processor hosts a block of nodes with contiguous node IDs). This way, we manage to change the characteristics of the reductions across adaptation phases. In particular, the inter-processor communication pattern of the reductions might change. For instance, suppose one node and its neighbor were distributed on two contiguous processors, after one or more steps of refinement, the node and its neighbor may end up being distributed on processers further apart.

For `AmrRed2D`, we have 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). 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 each phase contain 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.

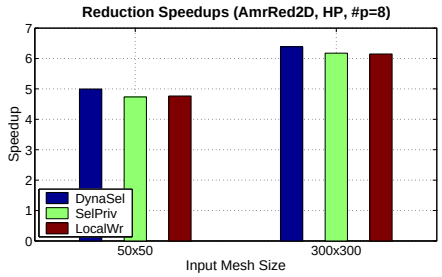

Figure 10: Speedups of adaptively selecting algorithms on `AmrRed2D`

.

Fig. 9 shows 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 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 has performed very poorly for `AmrRed2D`, we chose to not plot its execution time 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 couple observations from these graphs. First, the best parallel algorithms change dynamically during execution. Secondly, for most dynamic phases, `DynaSel` makes the right decision and applies the appropriate transformation algorithms. That is, `DynaSel` capitalized on the opportunity for run-time optimization.

Fig. 10 gives the overall loop speedups of the different parallel reduction algorithms. The conclusions here are that using *Adaptive Algorithm Selection* to select and apply the best parallel reduction algorithm for every dynamic phase has little overhead, it out performs applying any single other algorithm, and it improves overall perfor-

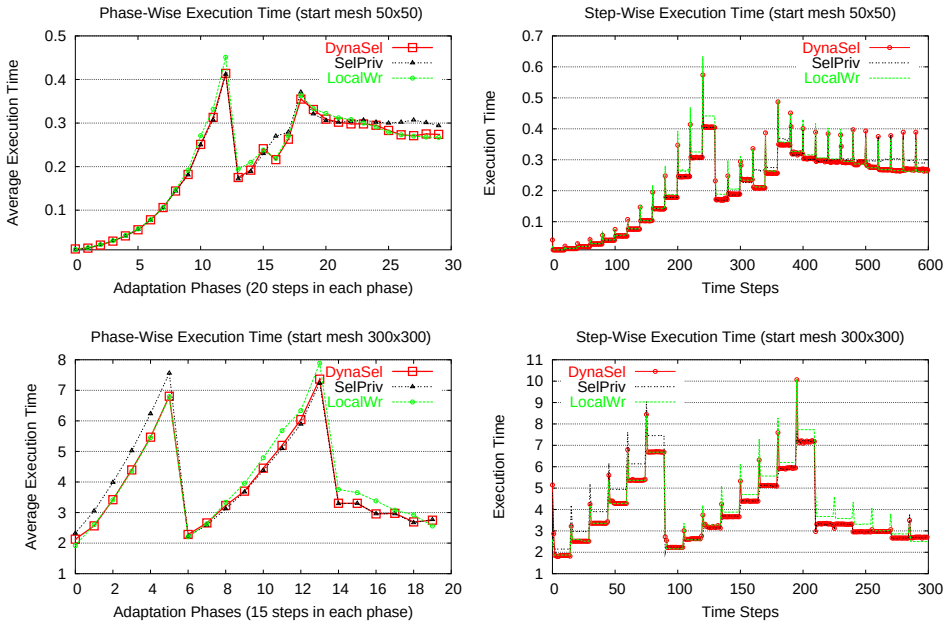

Figure 9: Phase-wise and Step-wise effects of dynamically selecting algorithms `AmrRed2D` with initial mesh sizes specified in graph title.

mance by $\sim 5\%$.

7.2 Molecular Dynamics (`Moldyn`)

`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 [11, 7, 19].

A high-level description of `Moldyn` is given in Fig. 12. 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 a quasi-linear time step which 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 an algorithm using a link-cell data structure to generate the neighbor lists [23]. 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 has been significantly reduced so that it is comparable to the time required for Step 3 for a system with a realistic number of particles. Since step 3 is executed many more times than Step 2, the execution time of the modified `Moldyn` is now truly dominated by the force computation, which is the irregular reduction we would like to optimize.

```
     Initialize the coordinates of particles.
     FOR (N time steps) DO
1      Update the coordinates of particles based
          on their forces and velocities.
       if (N mod K = 0) then
2        Build a neighbor list for each particle
            that are within 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.
```

Figure 12: A high level description of `Moldyn`.

For `Moldyn`, since we have observed performance changes 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 could change due to the different setup overheads of the algorithms. This way, we could examine both the effectiveness of the predictive models and the efficiency of selecting and switching algorithms. We experimented with 5 different inputs. The input specifications are given in Table 6. Please 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.

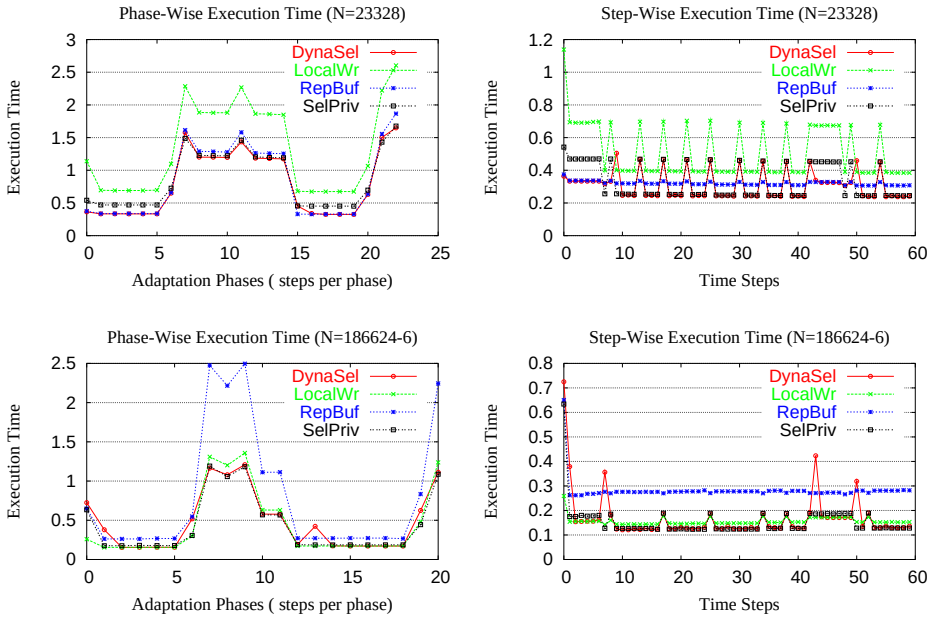

Figure 11: Phase-wise and Step-wise effects of dynamically selecting algorithms `Moldyn` (inputs #1 and #5).

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

Table 6: Specifications of dynamic inputs of `Moldyn`.

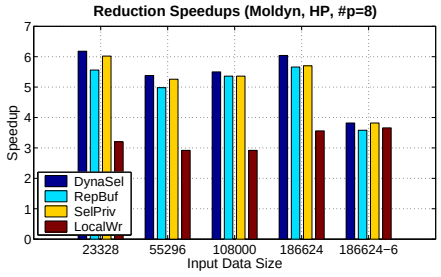

Figure 13: Speedups of adaptively selecting algorithms on `Moldyn`

.

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. 11 for two inputs (#1 and #4). Fig. 13 gives the overall loop speedups applying different reduction parallelization techniques. We note that since the number of steps of the phases may change, 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.

8 Related Work

There does not appear to be a great body of effort in the area of adaptive selection of high level algorithms. Brewer [5] 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 by, along with some code annotations, by the end user. A benchmarking phase determines the coefficients of these models, and the entire process was shown to be effective for both selecting among parallel sorting algorithms and determining data distribution for a parallel equation solver.

Li, Garzaran, and Padua [17] 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. Many previous work aim at tuning specific algorithm parameters. Examples are Spiral [28] and FFTW [10] for FFT signal processing and ATLAS [27] for matrix multiplica-

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

Another somewhat relevant approach is that of *dynamic feedback* [8] which selects code variants based on on-line profiling. There are many recent efforts to develop dynamic and adaptive compilation systems, however we feel they are out of scope for this work.

Several parallel reduction algorithms have been proposed following "owner computes" rule. The *data affiliated loop* [18] method traverses all the iterations in the reduction loop on each processor and only executes the reductions accessing local data. *Local Write* [11] generates a reusable schedule so that each processor only needs to traverse the iterations that have reductions accessing local data. The disadvantages of these techniques are that they may heavily replicate unnecessary computation (residing in iterations) and load balance is not sustained by the methods (unlike "data replication" based methods).

Zoppetti and Agrawal [33] proposed a parallel reduction algorithm that overlaps computation and communication for multi-threaded architectures. They also implemented an incremental inspector is used to update the computation schedule efficiently. The potential drawback of this technique is that it may introduce unnecessary communication (e.g., pipelining data sections to irrelevant computation threads).

Adaptive Data Repository (ADR) infrastructure [15] 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. Their experiments have shown that none of the three strategies worked the best for various query patterns and a predictive model was desired.

9 Conclusion

In this paper, we presented an *Adaptive Algorithm Selection Framework* that can automatically adapt to the input data, environment and machine and select the best performing algorithm. We have applied our framework to the adaptive selection of parallel reduction algorithms. We have identified a few high-level, architecture independent parameters characterizing a program's static structure, dynamic data access patterns and candidate transformations. We applied an off-line synthetic experimental process to automatically generate predictive models which are used to dynamically select the most appropriate optimization transformation among several functionally equivalent candidates.

Through experimental results, we have shown that our technique can select the most appropriate parallel reduction algorithms at run–time with very low overhead. When this technique is applied to dynamic programs selecting new algorithms for each of their dynamic phases performance is improved even further.

The importance of this work is that the presented adaptive optimization technique can model programs with irregular and dynamic behavior and customize solutions to each program instance. It is a general framework that can adapt any number of optimizations to the program's needs.

References

[1] Parasol – HP V-Class Multiprocessor. PARASOL Lab, http://parasol.tamu.edu.

[2] Texas A&M University Supercomputing Facility, http://sc.tamu.edu/fi ers/regatta.html.

[3] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerge. Stapl: An adaptive, generic parallel c++ library. In *Proc. of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Systems*, 2001.

[4] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoefl nger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[5] E. Brewer. High-level optimization via automated statistical modeling. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 80–91, Santa Barbara, CA, Aug. 1995.

[6] J. G. Castanos and J. E. Savage. Repartitioning unstructured adaptive meshes. In *Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS'00)*, Cancun, Mexico, May 2000.

[7] C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, GA, May 1999.

[8] P. C. Diniz and M. C. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, pages 71–84, 1997.

[9] R. Eigenmann, J. Hoefl nger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

[10] M. Frigo. Spl: A language and compiler for dsp algorithms. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 169–180, Atlanta, GA, 1999.

[11] H. Han and C.-W. Tseng. Improving compiler and runtime support for adaptive irregular codes. In *Proc. of the IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Paris, France, Oct. 1998.

[12] 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*, 25(6):597–621, 1995.

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

[14] C. Kruskal. Efficient parallel algorithms for graph problems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 869–876, August 1986.

[15] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Querying very large multi-dimensional datasets in adr. In *High Performance Networking and Computing (SC '99)*, pages 13–19, Portland, Oregon, Nov. 1999.

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

[17] X. Li, M. J. Garzaran, and D. Padua. A dynamically tuned sorting library. In *Proc. of the International Symposium on Code Generation and Optimization*, Mar. 2004.

[18] Y. Lin and D. A. Padua. On the automatic parallelization of sprase and irregular fortran programs. In *Proc. of the Workshop on Languages, Compilers and Run-time Systems for Scalable Computers (LCR)*, pages 41–56, Pittsburgh, PA, May 1998.

[19] 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*, 29(3):217–247, 2001.

[20] A. Miller. *Subset Selection in Regression (Second Edition)*. Chapman & Hall/CRC, 2002.

[21] L. Oliker and R. Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 39. ACM Press, 1999.

[22] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface, Version 2.0*, 2000.

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

[24] W. M. Pottenger. *Theory, Techniques, and Experiments in Solving Recurrences in Computer Programs*. PhD thesis, CSRD, Univ. of Illinois at Urbana-Champaign, May 1997.

[25] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5), May 1991.

[26] R. v. Hanxleden. Handling irregular problems with Fortran D – a preliminary report. In *Proc. of the 4th Workshop on Compilers for Parallel Computers (CPC93)*, Delft, Netherlands, Dec. 1993.

[27] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1–2):3–25, 2001.

[28] J. Xiong, J. Johnson, R. Johnson, and D. Padua. Spl: A language and compiler for dsp algorithms. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 298–308, Snowbird, UT, 2001.

[29] H. Yu. *Run-Time Optimization of Irregular Applications*. PhD thesis, Texas A&M University, Department of Computer Science, Mar. 2004.

[30] H. Yu, F. Dang, and L. Rauchwerger. Parallel reduction: An application of adaptive algorithm selection. In *Proc. of the 15th Annual Workshop on Language and Compilers for Parallel Computing (LCPC'02)*, pages 171–185, College Park, MD, July 2002.

[31] H. Yu and L. Rauchwerger. Adaptive reduction parallelization. In *Proceedings of the 14th ACM International Conference on Supercomputing, Santa Fe, NM*, May 2000.

[32] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, New York, 1991.

[33] G. Zoppetti and G. Agrawal. Compiler and runtime support for static and irregular reductions on a multithreaded architecture. 2003. Extension of the paper in the Proceedings of International Parallel and Distributed Processing Symposium (IPDPS'02).