SmartApps, an Application Centric Approach to High Performance Computing: Compiler-Assisted Software and Hardware Support for Reduction Operations*

Francis Dang*, María Jesús Garzarán§, Milos Prvulovic‡, Ye Zhang‡, Alin Jula*, Hao Yu*,
Nancy Amato†, Lawrence Rauchwerger*, and Josep Torrellas‡

Abstract

State-of-the-art run-time systems are a poor match to diverse, dynamic distributed applications because they are designed to provide support to a wide variety of applications, without much customization to individual specific requirements. Little or no guiding information flows directly from the application to the run-time system to allow the latter to fully tailor its services to the application. As a result, the performance is disappointing. To address this problem, we propose application-centric computing, or SMART APPLICATIONS. *In the executable of smart applications, the compiler embeds most run-time system services, and a performance-optimizing feedback loop that monitors the application's performance and adaptively reconfigures the application and the OS/hardware platform. At run-time, after incorporating the code's input and the system's resources and state, the* SMARTAPP *performs a global optimization. This optimization is* instance *specific and thus much more tractable than a global generic optimization between application, OS and hardware. The resulting code and resource customization should lead to major speedups. In this paper, we first describe the overall architecture of* SMARTAPPS *and then present some achievements to date, focusing on compiler-assisted software and hardware techniques for parallelizing reduction operations. These illustrate* SMARTAPPS *use of adaptive algorithm selection and moderately reconfigurable hardware.*

1 Introduction

Many important applications are becoming large consumers of computing power, data storage and communication bandwidth. For example, applications such as ASCI multiphysics simulations, real-time target acquisition systems, multimedia stream processing and geographical information systems (GIS), all put tremendous strains on the computational, storage and communication capabilities of the most modern machines. There are several reasons why the performance of current distributed, heterogeneous systems is often disappointing. First, they are difficult to fully utilize because of the heterogeneity of the processing nodes (usually with different capabilities) which are interconnected through a non-homogeneous network with different inter-node latencies and bandwidths. Secondly, the system may change dynamically while the application is running. For example, nodes may fail or appear, network links may be severed, and other links may be established with different latencies and bandwidths. Finally, in order to obtain decent performance, the work has to be partitioned in a balanced manner.

Current distributed systems have a fairly compartmentalized approach to optimization: applications, compilers, operating systems and even hardware configurations are designed and optimized in isolation and without the knowledge of input data. There is little information flow across these boundaries and no global optimization is even attempted. For example, many important activities managed by the operating system like paging activity, virtual-to-physical page mapping, I/O activity or data layout in disks are provided with little or no application customization. Since the compiler's analysis can discover much about an application's needs, performance could be boosted significantly if the OS provided hooks for the compiler, and possibly the user, to customize or tailor OS activities to the needs of a particular application. Current hardware is built for general purpose use to lower costs and has almost no tunable parameters that allow the compiler or the OS adjust it to specific application characteristics.

In addition to this lack of compiler/OS/hardware cooperation, a second important problem is that compilers do not necessarily know fully at compile time how an application will behave at run time. The reason is that the run-time behavior of an application may partly depend on its input

*Research supported in part by NSF CAREER Award CCR-9734471, NSF Grant ACI-9872126, NSF-NGS EIA-9975018 and EIA-0103742, NSF ITR ACI-0113971, DOE ASCI ASAP Level 2 Grant B347886, and Hewlett-Packard Equipment Grants

§Universidad de Zaragoza, http://www.cps.unizar.es/deps/DIIS/gaz
‡University of Illinois at Urbana-Champaign, http://iacoma.cs.uiuc.edu
*Texas A&M University, http://www.cs.tamu.edu/faculty/rwerger
†Texas A&M University, http://www.cs.tamu.edu/faculty/amato

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

data. Consequently, compilers may generate conservative code that does not take advantage of characteristics of the program's input data. This precludes many aggressive optimizations related to code parallelization, parallel algorithm substitution (when possible), and redundancy elimination. Moreover, we can only use expensive, generic methods for load balancing and memory latency hiding. If, instead, the compiler inserted code that, after reading the input data to the program at run-time, adaptively made optimization decisions, performance could be boosted significantly. Furthermore, at a higher level, the compiler may have the possibility of selecting an algorithm or a specific implementation of an algorithm from a library of functionally equivalent modules. If this choice is made based on the specific instance of an application then large-scale gains can be obtained. For example, if the code calls for a sorting routine, the compiler can specialize this call to a specific parallel sort that matches both the input data to be sorted as well as the architecture on which it will be executed.

Our ultimate goal is the overall minimization of execution time of dynamic applications in parallel systems. Instead of building individual, *generally optimized* components (compilers, run-time systems, operating systems, hardware) that can work acceptably well with any application, we will subordinate the whole optimization process to the particular needs of a specific application. We will drive the optimization with the requirements of an individual program and for a specific set of input data. Moreover, the optimization will be carried out continuously to adapt to the dynamic, time varying needs of the application. The final form of the executable of an application will take shape only at run-time, after all input data has been analyzed. The resulting Smart Application (SMARTAPP) will monitor its performance and, when necessary, restructure itself and the underlying OS and hardware to its new characteristics. Our approach promises to drastically reduce the generally intractable problem of global optimization because we optimize only a particular instance of an application. While this method may cost some additional overhead for every execution the resulting customized performance can more than pay off for long running codes.

2 System Architecture

We now give a general overview of our system which includes components at various levels of development. Some features of SMARTAPPS have been implemented, others have been studied but have not yet been prototyped while others are still in early stages. We give this high level architectural description that includes both accomplishments as well as work in progress in order to put our work in perspective.

The adaptive run-time system, shown in Figures 1 and 2, consists of a nested multi-level adaptive feedback loop that monitors the application's performance and, based on the magnitude of deviation from expected performance, compensates with various actions. Such actions may be run-time software adaptation, re-compilation, operating system and hardware reconfiguration. The system shown in Figure 1 uses techniques from a TOOLBOX shown in Figure 2. The TOOLBOX contains application and system specific databases and algorithms for performance evaluation, prediction and system reconfiguration. The tools are supported by architectural and performance models.

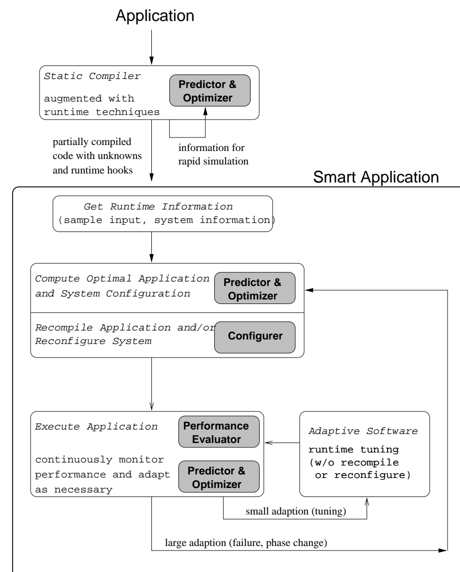

Figure 1. Smart Application.

The **first stage** of preparing a dynamic application for execution occurs outside the proposed run-time system. It is a pre-compilation in which all possible static compiler optimizations are applied. However, for many of the more aggressive and effective code transformations, the needed information is not statically available. For example, if the code solves a sparse adaptive mesh-refinement problem, the initial mesh is read from an input file only at the beginning of the execution and is therefore not available for static compilation. In this case, the compiler may use *speculative transformations* which will have to be validated at run-time. We will generate an intermediate code that will contain all the necessary compiler-internal information statically available, which will be combined with execution-time information to finish possible optimizations. This additional information will be packaged so that the application could in fact be executed, albeit sub-optimally, without passing through the second run-time compilation stage (the current level of development). Calls to generic algorithms or, when possible, parallel algorithm recognition and substitutions will be either left in their most general form or specialized to

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

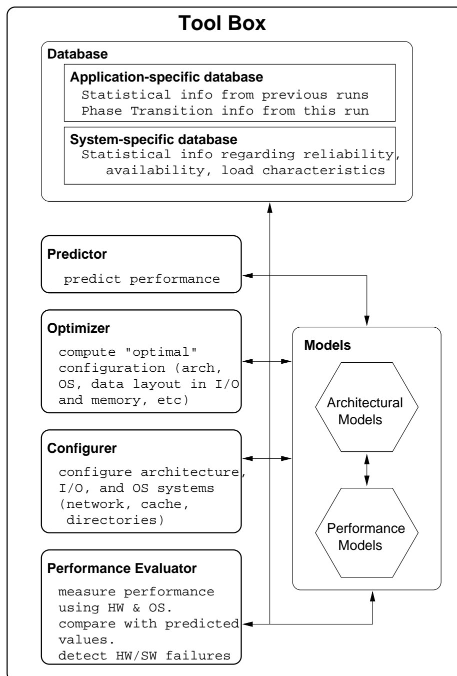

Figure 2. ToolBox.

the extent permitted by static compiler analysis, e.g., type analysis. For example, when a reduction operation is recognized or specifically called by the program, the compiler will possibly decide between the 'standard' parallel equivalent or 'histogram reductions' if enough knowledge can be extracted from the code [20].

The **second stage** in an application's life is driven by the run-time system. It starts by reading in and/or sampling the input data which are relevant to the 'unfinished' optimizations. This 'relevant' data is analyzed with fast, approximative methods and essential characteristics are extracted. The result of this analysis will place the instance of this application in a certain 'functioning domain' which represents the possible universe of forms that an application can take at run-time. Calls to routines that perform certain standard functions will be specialized by selecting from a linked library the algorithms and/or their implementations that match the 'functioning domain' (code and data) of this particular instantiation of the program. In addition, the run-time system provides information about the type and resource availability of the system on which the application will be executed. Performance monitoring instrumentation is added to the code based on its intrinsic structure as well as that of the run-time environment. Different architectural and operating system features will dictate which parameters are important, and which can be measured.

Then, a fast RUN-TIME COMPILER, which will be developed from an existing restructurer, will finish the compilation process and generate a highly optimized and adaptable code, the SMART APPLICATION. This executable will include code for adaptive run-time techniques that allow the application to make on-the-fly decisions about various optimizations. To this end, we will use our techniques for detecting and exploiting loop level parallelism in various cases encountered in irregular applications [15, 18, 17]. Load balancing will be achieved through *feedback guided blocked scheduling* [5] which allows highly imbalanced loops to be block scheduled by predicting a good work distribution from previous measured execution times of iteration blocks.

For certain simple algorithms, which can be automatically recognized, e.g., reductions, the compiler will insert code that can substitute the sequential version with a parallel equivalent that best matches the data access pattern of the application. This adaptive parallel algorithm substitution can be implemented either through multi-version code (library calls) as is currently done, or through recompilation.

The result of static and dynamic compiler analysis of the application will also produce a set of requirements of desirable features for the operating system. These requests are embedded in the user code and can call upon a tunable, modular OS to change some of its parameters (e.g., page mapping) and to perform some simple modification of the underlying architecture (e.g., type and/or number of system components). Furthermore, the compiler will generate (statically or at run-time) a list of specifications for the run-time environment. These application-level specifications are passed to the system configuration optimizer. The PREDICTOR and OPTIMIZER tools will use the application requirements and characteristics to compute an 'optimal' architectural configuration and tune the environment accordingly. In addition to the OS tuning we will perform architectural modification when feasible. This may range from the customization of communication protocols (e.g., specialized cache coherence protocols) to the specialization of processors for computing or communication. In the latter case the SMARTAPP will distribute the workload between 'classical' processors and processors in memory (IRAM).

In the following sections, we first briefly review some of the implemented compiler-generated run-time optimizations of the presented SMARTAPPS architecture, and then describe in more detail compiler-assisted software [20] and hardware [8] techniques for parallelizing reduction operations. These illustrate SMARTAPPS use of adaptive algorithm selection and moderately reconfigurable hardware.

3 Compiler Generated Run-Time Optimizations

Efficiently exploiting parallel machines in general and heterogeneous machines in particular depends upon the de-

gree to which a program has been optimized to execute on a given architecture. We believe that all optimization techniques, whether performed by compiler or programmer, are derived from three fundamental optimization principles: (i) maximizing parallelism while minimizing overhead and redundant computation, (ii) minimizing wait-time due to load imbalance, and (iii) minimizing wait-time due to memory latency.

The SMART APPLICATION mainly consists of a run-time library embedded by the compiler in the application and which can dynamically select compiler optimizations based on the above three principles (e.g., loop parallelization or scheduling for load balance). Some architectural reconfiguration and operating system level tuning may also be employed to obtain fast, low overhead performance improvement. We plan to integrate such adaptive techniques into the application by extending current static and run-time technologies and by developing completely new ones.

We have developed several techniques [16, 17, 15, 18] that can detect and exploit loop level parallelism in various cases encountered in irregular applications: (i) a speculative method to detect fully parallel loops (The LRPD Test), (ii) an inspector/executor technique 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). Details can be found in [16, 17, 18, 5].

We have recently developed a new technique that can extract the maximum available parallelism from a partially parallel loop and that removes limitations of previous methods (for partially parallel loops), i.e., it can be applied to any loop (even if no proper inspector can be extracted) and requires less memory overhead. The main idea of the Recursive LRPD test [5] is that in any block-scheduled loop executed under the processor-wise LRPD test with copy-in, the chunks of iterations that are less than or equal to the source of the first detected dependence arc are always executed correctly. Only the processors executing iterations larger or equal to the earliest sink of any dependence arc need to re-execute their portion of work. Thus only the remainder of the work (of the loop) needs to be re-executed. We have implemented the Recursive LRPD test and applied it to the three most important loops in TRACK, a Perfect code. As detailed in [5], we obtained very encouraging speedups – prior to this technique, TRACK was considered sequential.

4 Software Support for Reductions: Adaptive Algorithm Selection

Memory accesses in irregular programs take a variety of patterns and are dependent on the code itself as well as on their input data. Moreover, some codes are of a dynamic nature, i.e., they modify their behavior during their execution because they simulate position dependent interactions between physical entities.

A special and very frequent case of loop dependence pattern 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.[1]

It is difficult to a find a reduction parallelization algorithm (or for that matter, other optimizations) that will work well in all cases. We have designed an adaptive scheme that will detect the type of reference pattern through static (compiler) and dynamic (run-time) methods and choose the most appropriate scheme from a library of already implemented choices [20]. To find the best choice we establish a taxonomy of different access patterns, devise simple, fast ways to recognize them, and model the various old and newly developed reduction methods in order to find the best match. The characterization of the access pattern is performed at compile time whenever possible, and otherwise, at run-time, during an inspector phase or during speculative execution.

From the point of view of optimizing the parallelization of reductions (i.e., selecting the best parallel reduction algorithm) we recognize several characteristics of memory references to reduction variables. **CH** is a histogram which shows the number of elements referenced by a certain number of iterations, and **CHD** is the CH distribution. **CHR** is the ratio of the total number of references (or the sum of the **CH** histogram) and the space needed for allocating replicated arrays across processors, and the set of **CHR**s which have a high degree of contention is referred to as **HCHR**. **CON**, the Connectivity of a loop, is a ratio between the number of iterations of the loop and the number of distinct memory elements referenced by the loop [10]. The **Mobility (MO)** per iteration of a loop is directly proportional to the number of distinct elements that an iteration references. The **Sparsity (SP)** is the ratio of referenced elements to the dimension of the array. The **DIM** measure gives the ratio between the reduction array dimension and cache size. If the program is dynamic then changes in the access pattern will be collected, as much as possible, in an incremental manner. When the changes are significant enough (a threshold that is tested at run-time) then a re-characterization of the reference pattern is needed.

Our strategy is to identify the regular components of each irregular pattern (including uniform distribution), isolate and group them together in space and time, if this is not already the case, and then apply the best reduction paral-

[1] A reduction variable is a variable whose value is used in one associative and 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.

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

APP	MO	DIM	SP	CON	CHR	Recom. Scheme	Experimental Result
Irreg - DO 100	2	100,000	25	100	0.92	rep	rep > ll > sel > lw
		500,000	5	20	0.71	lw	lw > rep > ll > sel
		1,000,000	1.25	5	0.40	lw	lw > rep > ll > sel
		2,000,000	0.25	1	0.26	sel	sel > lw > ll > rep
Nbf - DO 50	1	25,600	25	200	0.25	ll	sel > ll > rep > lw
		128,000	6.25	50	0.25	sel	sel > ll > rep > lw
		256,000	0.625	5	0.25	sel	sel > ll > rep > lw
		1,280,000	0.25	2	0.25	sel	sel > ll > rep > lw
Moldyn - ComputeForces loop	2	16,384	23.94	95.75	0.41	rep	rep > ll > sel > lw
		42,592	7.75	31	0.36	rep	rep > ll > sel > lw
		70,304	1.69	6.75	0.33	ll	ll > rep > sel > lw
		87,808	0.375	1.5	0.29	ll	ll > rep > sel > lw
Spark98	1	30,169	0.625	5	0.18	sel	sel > ll > rep > lw
- smvpthread() loop		7,294	0.6	4.8	0.2	sel	ll > sel > rep > lw
Charmm - DO 78	2	332,288	35.88	17.9	0.14	sel	ll > sel > rep > lw
			17.94	8.97	0.15	sel	ll > sel > rep > lw
		664,576	1.12	4.48	0.13	sel	ll > sel > rep > lw
Spice - bjt100	28	186,943	0.14	0.04	0.125	hash	hash > ll > rep
		99,190	0.20	0.06	0.125	hash	hash > ll > rep
		89,925	0.16	0.05	0.125	hash	hash > ll > rep
		33,725	0.16	0.05	0.126	hash	hash > ll > rep

Figure 3. The data has been obtained from the execution of the applications 8 processors. INPUT: number of reduction elements; SP: sparsity; CON: connectivity; CHR: ratio of total number of references to space needed for per processor replicated arrays; MO: mobility.

lelization method to each component. We have used the following novel and previously known parallel reduction algorithms: local write (**lw**) [10] (an 'owner compute' method), private accumulation and global update in replicated private arrays (**rep**), replicated buffer with links (**ll**), selective privatization (**sel**), sparse reductions with privatization in hash tables (**hash**).

Our main goal, once the type of pattern is established, is to choose the appropriate reduction parallelization algorithm, that is, the one which best matches these characteristics. To make this choice we use a decision algorithm that takes as input measured, real, code characteristics, and a library of available techniques, and selects an algorithm for the given instance.

The table shown in Fig.3 illustrates the experimental validation of our method. All memory reference parameters were computed at run-time. The result of the decision process is shown in the "Recommended scheme" column. The final column shows the actual experimental speedup obtained with the various reduction schemes which are presented in decreasing order of their speedup. For example, for Irreg, the model recommended the use of Local Write. The experiments confirm this choice: *lw* is listed as having the best measured speedup of all schemes.

In the experiment for the SPICE loop, the hash table reduces the allocated and processed space to such an extent that, although the setup of a hash table is large, the performance improves dramatically. It is the only example where hash table reductions represent the best solution because of the very sparse nature of the references. We believe that codes in C would be of the same nature and thus benefit from it. There are no experiments with the Local Write method because iteration replication is very difficult due to the modification of shared arrays inside the loop body.

5 Hardware Support for Reductions: Private Cache-Line Reduction (PCLR)

We have proposed a new scheme *Private Cache-Line Reduction (PCLR)* which adds architectural support for performing reduction operations in scalable shared-memory multiprocessors.

The essence of PCLR is that each processor participating in the reduction uses *non-coherent lines in its cache* as temporary private storage to accumulate its partial results of the reduction. Moreover, if these lines are displaced from the cache, their value is *automatically accumulated* onto the shared reduction variable in memory. Finally, since the cache lines are non-coherent, cache misses are satisfied from within the local node by returning a line *filled with neutral elements*. Figure 4 shows a representation of the scheme.

With this approach, the processors are relieved of the initialization and merge-out work. Also, since the approach is still based on computing partial results and combining

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

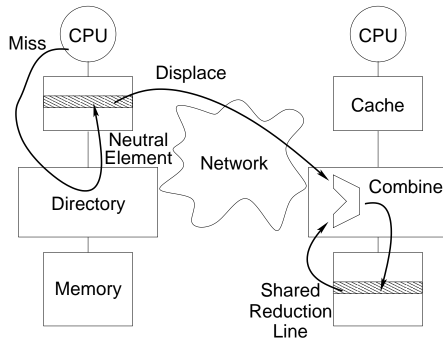

Figure 4. Representation of how PCLR works.

them, the reduction is performed with no critical sections.

The initialization phase is avoided by initializing the reduction lines on demand, as they are brought into the cache on cache misses. Since the cache is used as private storage to accumulate the partial results, there is no need to allocate any private array in memory. On a cache miss to a reduction line, the local directory controller intercepts the request and services it by supplying a line of neutral elements.

The merging phase is avoided by combining the reduction cache lines in the background as they are displaced from the cache during parallel loop execution. As each displaced reduction line reaches the home of the shared reduction variable, the directory controller combines its contents with the shared reduction variable in memory. Meanwhile, the processors execute the loop without interruption.

When the parallel loop ends, some partial results may remain in the caches. They must be explicitly flushed so that they are correctly combined with the shared data before any further code is executed. This flush step takes much less time than an ordinary merging phase. This is because it has less combining to perform, as most of it has already been performed through displacements during the loop execution. In fact, the work is at worst proportional to the size of the cache, rather than to the size of the shared array. It is also more efficient because the processor issues no remote loads. Instead, it simply sends all the partial results to their homes, where the directory controller combines the data.

With PCLR support, the parallelized reduction code is shown in Figure 5. Note that we have added a call to a function that configures the machine for PCLR before the loop execution. This example is simplified by using static scheduling and omitting the forking and joining code.

5.1 Implementation of PCLR

Any implementation of PCLR has to consider the following issues: differentiation of reduction data (Sections 5.1.1 and 5.1.5), support for on-demand initialization (Section 5.1.2) and combining (Section 5.1.3) of lines and con-

```
1    ConfigHardware(arguments);
// The range 0..Nodes is split among the processors
2    for(i=MyNodesBegin;i<MyNodesEnd;i++)
3        w[x[i]]+=expression;
4    CacheFlush();
5    barrier();
```

Figure 5. PCLR Parallelized reduction code.

figuration of the hardware (Section 5.1.4). We discuss these issues in this section.

In the following discussion, we assume a CC-NUMA architecture such as the one in Figure 4. Each node in the machine has a directory controller that snoops and potentially intervenes on all requests and write-backs issued by the local cache, even if they are directed to remote nodes.

5.1.1 Differentiating Reduction Data

While the data used in reduction operations remain in the cache, they are read and written just like regular, non-reduction data. However cache misses and displacements of reduction data require special treatment. Consequently, any implementation of PCLR has to provide a way to distinguish reduction data from regular data.

A simple way of doing so is to use special load and store instructions for “reduction” accesses. Cache lines accessed by these special instructions are marked as containing reduction data by putting them into a special “reduction” state. In this state, a processor can read and write the line without sending invalidations, even though other processors may be caching the same memory line. Misses by reduction loads and displacements of lines in the reduction state cause special transactions that are recognized by the local and home directories, respectively.

Note that we assume that reduction and regular data never share a cache line. Although it would be possible to enhance our scheme to support line sharing, alignment of reduction data on cache line boundaries is beneficial even without PCLR. Consequently, we assume that the compiler guarantees no line sharing.

In the following, we explain the rest of PCLR assuming this simple approach to differentiating reduction data. In Section 5.1.5, we propose a more advanced scheme for reduction data differentiation that allows using unmodified or slightly modified processors and caches.

5.1.2 On-Demand Initialization of Reduction Lines

When a reduction load misses in the cache, a specially-marked cache line read transaction is issued to the memory system. The local directory controller intercepts the request

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

and satisfies it by returning a line initialized with *neutral elements* for the particular reduction operation. The line is loaded into the cache in the reduction state.

A reduction load may hit in the cache on a line that is not in the reduction state. This may occur if the line had been accessed prior to the reduction loop with plain accesses and happened to linger in the cache. In this case, if the line is in state dirty, it is written back to memory in a plain write-back. Irrespective of its state, the line is then invalidated. Finally, the cache issues a reduction read miss as indicated above.

5.1.3 On-Demand Combining of Partial Results

When a line in the reduction state is displaced from the cache, a specially-marked write-back transaction is issued to the memory system. Once the write-back arrives at its home, the directory controller reads the previous contents of the line from memory, combines it with the newly-arrived partial result, and stores the updated line back to memory. The combining of the lines is done according to the reduction operator in the code, and is performed for every single element in the line. Note that those elements of the displaced line that were not accessed by the processor still contain the neutral element, so the effect of merging them with memory content is that the memory content is unchanged.

To combine the lines, the directory controller has to be enhanced with execution units that support the required reduction operators. Since a cache line contains several individual data elements, such execution units may become a bottleneck if their performance is too low. Luckily, all the elements of a line can be processed in parallel or in a pipelined fashion. Consequently, it is not too difficult to improve the performance by pipelining these execution units or adding more units.

These execution units should include an integer ALU for integer operations. For floating-point operations, having a full floating-point unit would be more general, but would also increase the complexity of the directory controller significantly. Our experience with the applications in Section 6.2 suggests that multiplication is rarely used as a reduction operator. Thus, for floating-point operations, having a floating-point adder and comparator is sufficient.

Finally, it is possible that the reduction data had been accessed prior to the reduction loop with plain accesses, and still lingers in several caches when the reduction loop starts. To handle this case, when the home directory controller receives a write-back for the line, it always checks the list of sharer processors for the line in the directory. Note that misses due to the reduction accesses do not go to the home. Thus, the home only has sharing information about non-reduction sharers. If the line is in a (non-reduction) dirty state in a cache, the controller recalls the line and writes it back to shared memory before performing any combining. The controller also sends invalidations to all (non-reduction) sharer processors. After the first reduction write-back of a line, the list of sharers at its home is empty for the remainder of the reduction loop and causes no further invalidation or recall messages.

5.1.4 Configuring the Hardware

The configuration of the hardware is controlled by the compiler which inserts the necessary code into the SMARTAPP. Before executing a reduction loop, each processor issues a system call to inform the directory controller in its node about the data type and the operation of the reduction. This is shown in line 1 of Figure 5. With this simple approach, we can only support one type of reduction operation per parallel section. In our example of Figure 5, the controller must be configured to perform double-precision floating-point addition when it receives a reduction write-back.

Any loop that performs several types of reduction operation must be distributed into multiple loops, so that each loop performs only one type of reduction operation. Fortunately, loops with multiple types of reduction operation are rare.

Finally, the operating system knows if different, time-shared processes want to use different types of reduction operations. If this is the case, the operating system flushes the reduction data from the caches when a process is preempted, and reprograms the directory controller when the process is re-scheduled.

5.1.5 Advanced Differentiation of Reduction Data

In Section 5.1.1 we explained a simple mechanism to distinguish reduction data from regular data and then explained the rest of PCLR using that simple mechanism. Now we propose a more advanced, but equivalent, mechanism that eliminates the need to modify the processor, the caches, or the coherence protocol.

In this scheme, instead of using special instructions, cache states, and protocol transactions to identify reduction data, such data are identified by using *Shadow Addresses* [4]. The scheme works as follows. In the reduction code, we use a *Shadow Array* instead of the original reduction array. For example, in Figure 5, we would use array *w_redu* instead of *w*. This shadow array is mapped to physical addresses that do not contain physical memory. However, such addresses differ from the corresponding physical addresses of the original array in a known manner. For example, they can have their most significant bit flipped. As a result, when a directory controller sees an access that addresses nonexistent memory, it will know two things. First, it will know that it is a reduction access. Second, from the

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

physical address, it will know what location of the original array it refers to.

With this approach, we do not need to modify the hardware of the processor, caches, or coherence protocol. The only requirement is that the machine must be able to address more memory than physically installed. Then, when a directory controller sees a read miss from the local processor to nonexistent memory, it simply returns a line of neutral elements to the processor. Furthermore, when a directory controller sees the write-back of a line from the local processor to nonexistent memory, it will forward it to the home of the corresponding element of the original array. Finally, when a directory controller receives the write-back of a line from a remote processor, it translates its address to the address of the corresponding element in the original array and combines the incoming data with the data in memory.

This approach requires modest compiler and operating system support. The compiler modifies the reduction code to access a shadow array instead of the original array. It also declares the shadow array and inserts a system call to tell the operating system which array is shadow of which. The operating system has to support the mapping of pages for the shadow array. Specifically, on a page fault in the shadow array, it assigns a nonexistent physical page whose number bears the expected relation to the number assigned to the corresponding original array page. Moreover, if the latter does not exist yet, it is allocated at this time.

5.2 Summary

The PCLR scheme addresses many of the problems of performing parallel reductions in scalable shared-memory multiprocessors. PCLR has two main advantages. First, it uses cache lines as the only private storage and initializes them on demand. As a result, there is no need to allocate private data structures or to perform a cache-sweeping initialization loop. Second, it performs the combining of the partial results with their shared counterparts on demand, as the reduction loop executes. As a result, there is no need for a costly merging step that involves sweeping the cache and many remote misses. All that is needed is to flush the reduction data from the caches at the end of the loop. These two advantages are particularly important when the reduction access patterns are sparse.

Most PCLR modifications are in the directory controllers, which perform special actions on read misses and write backs. With the use of shadow addresses, the only modification to the processor and caches is the ability to pin and unpin lines in the caches through the *load&pin* and *store&unpin* instructions. It can be argued that these instructions could also be useful for other functions in modern processors.

6 Evaluation

We evaluate the PCLR scheme using simulations driven by several applications.

6.1 Simulation Environment

We use an execution-driven simulation environment based on an extension to MINT [19] that includes a dynamic superscalar processor model [12]. The architecture modeled is a CC-NUMA multiprocessor with up to 16 nodes. Each node contains a fraction of the shared memory and the directory, as well as a processor and a two-level cache hierarchy with a write-back policy. The processor is a 4-issue dynamic superscalar with register renaming, branch prediction, and non-blocking memory operations. Table 1 lists the main characteristics of the architecture. Contention is accurately modeled in the entire system, except in the network, where it is modeled only at the source and destination ports.

Processor Parameters	Memory Parameters
4-issue dynamic, 1 GHz	L1, L2 size: 32 KB, 512 KB
Int, fp, ld/st FU: 4, 2, 2	L1, L2 assoc: 2 way, 4 way
Inst. window: 64	L1, L2 size: 64 B, 64 B
Pending ld, st: 8, 16	L1, L2 latency: 2, 10 cycles
Branch penalty: 4 cycles	Local memory latency: 104 cycles
Int, fp rename regs: 64, 64	2-hop memory latency: 297 cycles

Table 1. Architectural characteristics of the modeled CC-NUMA. The latencies shown measure contention-free round trips from the processor in processor cycles.

The system uses a directory-based cache coherence protocol along the lines of DASH [14]. Each directory controller has been enhanced with a single double-precision floating-point add unit. Both the directory controller and the floating point-unit are clocked at 1/3 of the processor's frequency. The floating-point unit is fully pipelined, so it can start a new addition every three processor cycles. Its latency is 2 cycles (6 processor cycles). Floating-point addition is the only reduction operation that appears in our applications (Section 6.2).

Private data are allocated locally. Pages of shared data are allocated in the memory module of the first processor that accesses them. Our experiments show that this allocation policy for shared data achieves the best performance results for both the baseline and the PCLR system.

6.2 Applications

To evaluate the PCLR system, we use a set of FORTRAN and C scientific codes. The applications *Euler* from HPF-2 [7] and *Equake* from SPECfp2000 [11], and the kernels:

Appl.	Names of Loops	% of Tseq	# of Invocations	Iters. per Invocation	Instruc. per Iter.	Red. Ops. per Iter.	Red. Array Size (KB)	Lines Flushed	Lines Displaced
Euler	*dflux_do[100,200]* *psmoo_do20* *eflux_do[100,200,300]*	84.7	120	59863	118	14	686.6	3261	2117
Equake	*smvp*	50.0	3855	30169	550	22	707.1	742	580
Vml	*VecMult_CAB*	89.4	1	4929	135	6	40.0	168	0
Charmm	*dynamc_do*	82.8	1	82944	420	54	1947.0	1849	330
Nbf	*nbf_do50*	99.1	1	128000	1880	200	1000.0	238	1774
Average		81.2	795	61181	620	59	871.0	1251	960

Table 2. Application characteristics. In *Euler*, we only simulate dflux_do100, and all the numbers except Tseq correspond to this loop. The data in the last two columns correspond to a single loop, and are collected through simulation of a 16-processor system.

Vml from Sparse BLAS [6], *Charmm* from [3], and *Nbf* from the GROMOS molecular dynamics benchmark [9].

All of these codes have loops with reduction operations. Table 2 lists the loops that we simulate in each application and their weight relative to the total *sequential* execution time of the application (%Tseq). This value is obtained by profiling the applications on a single-processor Sun Ultra 5 workstation. The table also shows the number of loop invocations during program execution, the average number of iterations per invocation, the average number of instructions per iteration, the average dynamic number of reduction operations per iteration, and the size of the reduction array. The last two columns will be discussed in the next section.

The loops in Table 2 are analyzed by the Polaris parallelizing compiler [2] or by hand to identify the reduction statements. Then, we modify the code to implement the parallel reduction code for the software and PCLR algorithms. For PCLR, reduction accesses are also marked with special load and store instructions to trigger special PCLR operations (Section 5.1.1) in our simulator.

Next we report data, including speedups, for only the sections of code described in Table 2. Also, since there is a significant variation in speedup figures across applications, we report average results using the *harmonic mean*.

Impact of PCLR We evaluate two different implementations of our PCLR scheme. The first one is an implementation where the directory controller is hardwired. The second one utilizes a programmable directory controller, similar to the MAGIC micro-controller in the FLASH multiprocessor [13]. A programmable controller can provide the functionality required by PCLR without requiring hardware changes. These two implementations of PCLR are compared against a baseline system, i.e., a software-only reduction parallelization. The software-only approach accumulates partial results in private arrays and merges the data out when the loop is done.

Figure 6 compares the execution time of these three systems. The baseline software-only system is *Sw*. The PCLR implementation with a hardwired directory controller is *Hw*, and the implementation with a flexible programmable directory controller is *Flex*. The simulated system is a 16-node multiprocessor. For each application, the bars are normalized to *Sw*, and broken down into time spent in the initialization phase of the *Sw* scheme (*Init*), loop body execution (*Loop*), and time spent merging the partial results at the end of the loop in *Sw* or flushing the caches in *Hw* and *Flex* (*Merge*). The numbers above each bar show the speedup relative to the sequential execution of the code. In the sequential execution, all data were placed on the local memory of the single active processor.

The figure shows that the speedups in *Flex* are, on the average, only 16% lower than in *Hw* and 136% higher than in *Sw*. Therefore, implementing PCLR using a programmable directory controller is a good trade-off. Overall, for a 16-node multiprocessor, the *Hw* PCLR scheme achieves an average speedup of 7.6, while the software-only system delivers an average speedup of only 2.7. If PCLR is implemented with a programmable directory controller the average speedup is 6.4.

Scalability of PCLR. To evaluate the scalability of PCLR, we have simulated a multiprocessor system with 4, 8, and 16 processors. Figure 7 shows the harmonic mean of the speedups delivered by the different mechanisms. It can be seen that PCLR (both *Hw* and *Flex*) scale well. However, the *Sw* scheme scales poorly. The time of the merging step in *Sw* does not decrease when more processors are available. If the main loop scales well, the merging step limits the achievable speedups according to Amdahl's law.

7 Conclusion

So far we have made good progress on the development of many the components of SmartApps. We will further develop these and combine them into an integrated system.

In this paper we have illustrated SmartApps capability by presenting two complementary techniques for adaptively optimizing an important operation for parallel programs: reductions. Our software-based approach describes

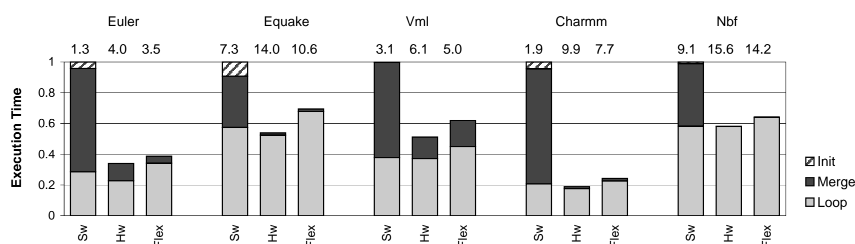

Figure 6. Execution time under different schemes for a 16-node multiprocessor. The numbers above the bars are speedups relative to the sequential execution.

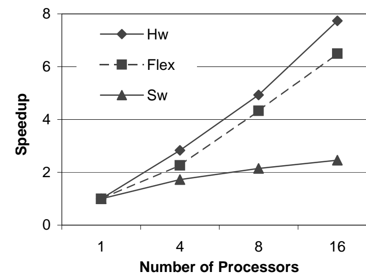

Figure 7. Speedups delivered by the different mechanisms (harmonic mean).

how a compiler can generate multi-version code that adapts to the code behavior. We have also illustrated how hardware support can be used to enable better speedups.

References

[1] N. Amato L. Rauchwerger and J. Torrellas. Smartapps: An application centric approach to high performance computing. In *Proc. 13th Workshop on Programming Languages and Compilers for Parallel Computing*,LNCS, 2000.

[2] W. Blume *et al.* Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[3] B. R. Brooks *et al.* CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *J. of Computational Chemistry*, (4):187–217, 1983.

[4] J. B. Carter *et al.* Impulse: Building a Smarter Memory Controller. In *Proc. of the Fifth Int. Symp. on High Performance Computer Architecture*, pp. 70–79, January 1999.

[5] F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. In *Proc. Int. Parallel and Distributed Processing Symp.*, April. 2002.

[6] I. Duff, M. Marrone, G. Radiacti, and C. Vittoli. A set of Level 3 Basic Linear Algebra Subprograms for Sparse Matrices. Tech. Rept. RAL-TR-95-049, Rutherford Appleton Laboratory, 1995.

[7] I. Duff, R. Schreiber, and P. Havlak. HPF-2 Scope of Activities and Motivating Applications. Technical Report CRPC-TR94492, Rice Univ., Nov. 1994.

[8] M. Garzaran, A. Jula, M. Prvulovic, H. Yu, L. Rauchwerger, and J. Torrellas, Architectural support for parallel reductions in scalable shared-memory multiprocessors. In *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[9] W. Gunsteren and H. Berendsen. GROMOS: GROningen MOlecular Simulation software. Tech. Rept., Lab. of Physical Chemistry, Univ. of Groningen, 1988.

[10] H. Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, Oct. 1998.

[11] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[12] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, Oct. 1998.

[13] J. Kuskin *et al.* The Stanford FLASH Multiprocessor. In *Proc. of the 21st Annual Int. Symp. on Computer Architecture*, pp. 302–313, April 1994.

[14] D. Lenoski *et al.* The Stanford Dash Multiprocessor. *IEEE Computer*, pp. 63–79, March 1992.

[15] L. Rauchwerger, N. Amato, and D. Padua. A scalable method for run-time loop parallelization. *Int. J. Paral. Prog.*, 26(6):537–576, July 1995.

[16] L. Rauchwerger. Run–time parallelization: A framework for parallel computation. Technical Report UIUCDCS-R-95-1926, Dept. of Computer Science, Univ. of Illinois, Urbana, Illinois, Sept. 1995.

[17] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Parallel and Distributed Systems*, 10(2), 1999.

[18] L. Rauchwerger and D. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. In *Proc. of 9th Int. Parallel Processing Symp.*, April 1995.

[19] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proc. 2nd Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecomm. Systems, pp. 201–207, Jan. 1994.*

[20] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization. In *Proc. of the 14th ACM Int. Conf. on Supercomputing, Santa Fe, NM*, May 2000.

0-7695-1573-8/02/$17.00 (C) 2002 IEEE