SmartApps: An Application Centric Approach to High Performance Computing*

Lawrence Rauchwerger[1], Nancy Amato[1], and Josep Torrellas[2]

[1] Department of Computer Science
Texas A&M University
{rwerger,amato}@cs.tamu.edu
[2] Department of Computer Science
University of Illinois
torrella@cs.uiuc.edu

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 the achievements to date: Run-time optimizations, performance modeling, and moderately reconfigurable hardware. The paper concludes with a short description of current and future development work.

1 Introduction

Many important applications are becoming large consumers of computing power, data storage and communication bandwidth. For example, applications such as ASCI multi-physics 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

* Research supported in part by NSF CAREER Award CCR-9734471, NSF CAREER Award CCR-9624315, NSF Grant ACI-9872126, NSF-NGS EIA-9975018, DOE ASCI ASAP Level 2 Grant B347886, and Hewlett-Packard Equipment Grants

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 too 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 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.

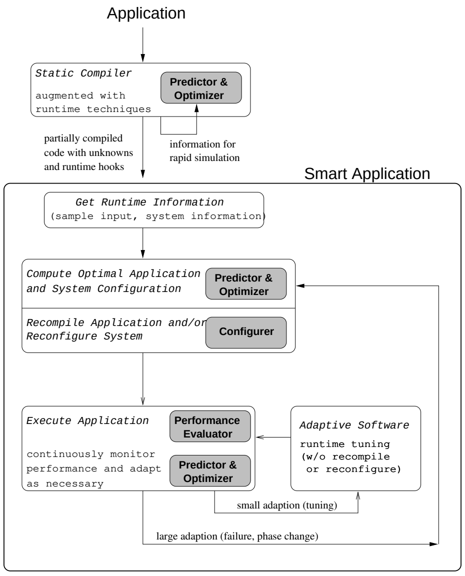


Figure 1. Smart Application.

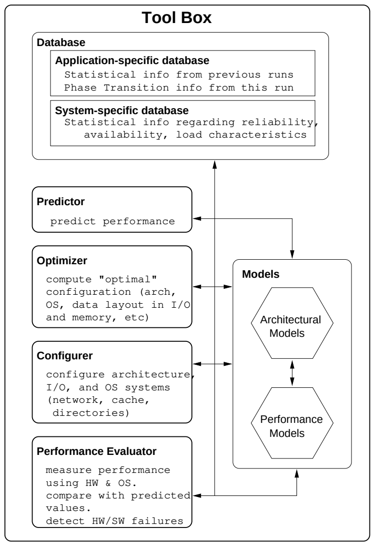


Figure 2. ToolBox.

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. In the following sections we discuss in more detail those components that are in a more advanced state.

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, or 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.

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 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 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 [35].

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 [24, 27, 26]. Load balancing will be achieved through *feedback guided blocked scheduling* [11] 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 technique 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 enable the program to 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). During this code generation phase, 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 can perform architectural modifications when feasible. As we show in Section 5 we have simulated the possibility of customizing communication protocols (e.g., specialized cache coherence protocols). In the future we hope to be able specialize processors for computing or communication and distribute the workload between 'classical' processors and processors in memory (IRAM).

In the next sections we elaborate on some of the currently implemented components of the presented SMARTAPPS architecture.

3 Compiler Generated Run-Time Optimizations

Efficiently exploiting parallel machines in general and heterogeneous machines in particular depends upon the degree 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 non-intrusive 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. In the following sections we detail some of these optimization methods and show how they can be incorporated into an integrated adaptive system for dynamic, heterogeneous computing.

3.1 Run-time Parallelization

We have developed several techniques [24–27] 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). We now briefly describe the utility of some of these techniques; details of their design can be found in [25–27, 11] and other related publications.

Partially Parallel Loop Parallelization. We have previously developed a run–time technique for finding an optimal parallel execution schedule for a partially parallel loop [23, 24]. Given the original loop, the compiler generates *inspector* code that performs run–time preprocessing (based on a sorting algorithm) of the loop's access pattern, and *scheduler* code that schedules (and executes) the loop iterations. The inspector is fully parallel, uses no element-wise synchronization, and can implement at run–time array privatization and reduction parallelization. Unfortunately this method is not generally applicable because a proper, side-effect free inspector cannot be extracted from a loop where address and data computation form a dependence cycle.

The Recursive LRPD Test. In previous work we have introduced the LRPD test for `DOALL` parallelization which speculatively executes a loop in parallel and tests subsequently if any data dependences could have occurred [25, 26]. If the test fails, the loop is re–executed sequentially. To qualify more parallel loops, *array privatization* and *reduction parallelization* can be speculatively applied and their validity tested after loop termination. The test uses *shadow structures* to analyze the loop's access pattern. It can be shown that if the LRPD test passes, i.e., the loop is in fact fully parallel, then a significant portion of the ideal speedup of the loop is obtained. The drawback of this method is that if the test fails a slowdown equal to the parallel speculative execution of the loop may be experienced.

We have now developed a new technique that can extract the maximum available parallelism from a partially parallel loop and that removes the limitations of our previous methods (for partially parallel loops), i.e., it can be applied to any loop (even in the case when no proper inspector can be extracted) and requires less memory overhead. The main idea of the Recursive LRPD test [11] 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, which can represent a significant saving over the previous LRPD test method (which would re-execute the whole loop sequentially).

To re-execute the fraction of the iterations assigned to the processors that may have worked off erroneous data we need to repair the unsatisfied dependences. This can be accomplished by initializing their privatized memory with the data produced by the lower ranked processors. Alternatively, we can commit (i.e., copy-out) the correctly computed data from private to shared storage and use on-demand copy-in during re-execution. We then re-apply recursively the fully parallel LRPD test on the remaining iterations until all processors have correctly finished their work. For loops with few cross-processor dependences we can expect to finish in only a few parallel steps. We have used two different strategies when re-executing a part of the loop: We can re-execute only on the processors that have incorrect data and leave the rest of them idle (NRD), or, we can redistribute at every stage the remainder of the work across all processors (RD). There are pros and cons for both approaches. Through redistribution of the work we employ all processors all the time and thus the execution time of every stage decreases (instead of staying constant, as in the NRD case). The disadvantage is that we may uncover new dependences across processors which were satisfied before by executing on the same processor. Moreover, there is a 'hidden' but potentially large cost associated with work redistribution: more remote misses during loop execution due to data redistribution between the stages of the test. The worst case time complexity for no redistribution (NRD) is the cost of a sequential execution. There are at most p steps performing n/p work, where p is the number of processors and n is the number of iterations. In the RD (with redistribution) case we will take progressively less time because we execute in p processors decreasing the amount of work. The number of steps is heavily dependent on the distribution of data dependences of the loop. For example, if we assume that at every step we perform correctly 1/2 the work then the total time is less than twice the fully parallel execution time of the loop. In practice we have obtained better results by adopting a hybrid method which redistributes until the predicted execution time of the remaining work is less than the overhead associated with re-distribution. In other words, we redistribute until the potential benefit of using more processors is outweighed by the cost. From that point on we continue without redistribution. A potential drawback is that the loop needs to be statically block scheduled in increasing order of iteration. The negative impact of this limitation can be reduced through dynamic feedback guided scheduling [9]. By applying this new method exclusively we can remove the uncertainty or unpredictability of execution time associated with the LRPD test – we can guarantee that a speculatively parallelized program will run at least as fast as its sequential version with some additional (minor) testing overhead.

We have implemented the Recursive LRPD test in both RD and NRD flavors and applied it to the three most important loops in TRACK, a Perfect code. The implementation is partially done by our run-time pass in Polaris (to automatically apply the simple LRPD test) and then additional code has been inserted manually. Our experimental testbed is a 16 processor ccUMA HP-V2200 system running HPUX11. It has 4Gb of main memory and 4Mb single level caches.

The main loops in TRACK are EXTEND_400, NLFILT_300 and FPTRACK_300. They account for $\approx 90\%$ of sequential execution time. We have increased the input set to increase the execution time as well as all associated data structures. The degree of parallelism in the loop from NLFILT is very input sensitive and ranges from fully parallel to a significant number of cross-processor dependences. All considered loops are very load imbalanced and thus, until our feedback guided load balancing is fully implemented, causes low speedups. Figures 3 (a-c) show the speedups for individual loops and Figure 3(d) shows the speedup for the entire program. Previous to this technique this code was considered sequential.

3.2 Adaptive Algorithm Selection: Choose the Right Method for Each Case

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. For example, they might 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

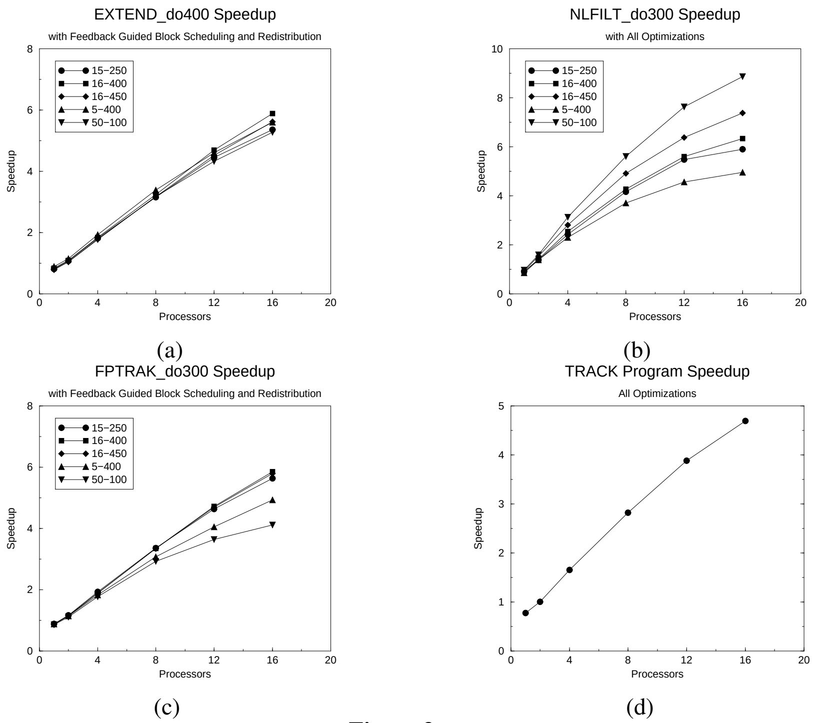


Figure 3. .

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 [35]. 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

[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.

APP	MO	DIM	SP	CON	CHR	Recom. Scheme	Experimental Result
Irreg	2	100,000	25	100	0.92	rep	rep > ll > sel > lw
- DO 100		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	1	25,600	25	200	0.25	ll	sel > ll > rep > lw
- DO 50		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	2	16,384	23.94	95.75	0.41	rep	rep > ll > sel > lw
- ComputeForces loop		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	2	332,288	35.88	17.9	0.14	sel	ll > sel > rep > lw
- DO 78			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	28	186,943	0.14	0.04	0.125	hash	hash > ll > rep
- bjt100		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 4. The data has been obtained from the execution of the applications 8 processors. DIM: 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.

number of distinct memory elements referenced by the loop [17]. 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 parallelization method to each component. We have used the following novel and previously known parallel reduction algorithms: local write (**lw**) [17] (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.4 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 hash tables. 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.

4 The Toolbox: Modeling, Prediction, and Optimization

In this section, we describe our current results in developing a performance PREDICTOR whose predictions will be used to select among various algorithms, and to help diagnose inefficiencies and identify potential optimizations.

4.1 High-Level Organizational Models

These models are based on bandwidth/latency models (e.g., BSP [34], LogP [10], or CGM [12]), which incorporate system specific (measured) parameters accounting for communication costs such as bandwidth and synchronization. Although much progress has been made, we still lack adequate tools for predicting actual algorithm performance on real machines. Our work [3] indicates that further progress towards this goal requires a tighter coupling of the cost model to the architecture, and specifically, to the memory system (e.g., caching, memory, I/O). In particular, we have shown that high accuracy can be attained when the application's interaction with the memory system is known (e.g., accessing an array with a constant stride, or with a known degree of inter-processor contention). In such cases, we show that simple BSP-like performance models based on counts of reads (loads) and writes (stores) can provide quite accurate predictions (maximum errors less than 5% for a variable number of processors and access patterns on the SGI PowerChallenge [3]). Our best BSP-like models include components accounting for a variable number of processors and memory distributions (e.g., cache sizes), and thus can be used to predict performance on different system configurations.

The scenario covered by our models is precisely the situation in SmartApps. In particular, at run-time, when our predictions will be made, the application's access pattern will be characterized, enabling us to select an appropriate model. In cases in which we cannot characterize the access pattern, we will provide a pair of best-case and worst-case models whose predictions will contain the actual execution time (a *prediction interval* [3]).

4.2 Low-level models

Significant work has been done in low-level analytical models of computer architectures and applications [33, 1, 32, 22]. While such analytical models had fallen out of favor, being replaced by comprehensive simulations, they have recently been enjoying a resurgence due the need to model large-scale NUMA machines and the availability of hardware performance counters [18, 7]. However, these models have mainly been used to analyze the performance of various architectures or system-level behaviors. That is, they have not been considered as competitive approaches to models such as the *BSP*.

In [2], we propose a cost model that we call *F*, which is based on values commonly provided by hardware performance monitors, that displays superior accuracy to the BSP-like models (our results on the SGI PowerChallenge use the MIPS R10000 hardware counters [31]). Function *F* is defined under the assumption that the running time is determined by one of the following factors: (1) the accesses issued by some processor at the various levels of the hierarchy, (2) the traffic on the interconnect caused by accesses to main memory, or (3) bank contention caused by accesses targeting the same bank. For each of the above factors, we define a corresponding function (*F1*, *F2*, and *F3*, resp.) which should dominate when that behavior is the limiting factor on performance. That is, we set *F*$=\max\{$ *F1*, *F2*, *F3*$\}$. The functions are linear relations of values measurable by hardware performance counters, such as loads/stores issued, L1 and L2 misses and L1 and L2 write-backs, and the coefficients are determined from micro-benchmarking experiments designed to exercise the system in that particular mode.

A complete description of *F*, including detailed validation results, can be found in [2]. We present here a synopsis of the results. The function *F* was compared with three *BSP*-like cost functions based, respectively, on the *Queuing Shared Memory* (*QSM*) [15] and the (d,x)-*BSP* [8], which both embody some aspects of memory contention, and the *Extended BSP* (*EBSP*) model [19], which extends the *BSP* to account for unbalanced communication. Since the *BSP*-like functions do not account for the memory hierarchy, we determined an optimistic (min) version and a pessimistic (max) version for each function. The accuracy of the *BSP*-like functions and *F* were compared on an extensive suite of synthetic access patterns, three bulk-synchronous implementations of parallel sorting, and the NAS Parallel Benchmarks [14]. Specifically, we determined measured and predicted times (indicated by T_M and T_P, respectively) and calculated the prediction error as ERR $= \frac{\max\{T_M, T_P\}}{\min\{T_M, T_P\}}$, which indicates how much smaller or larger the predicted time is with respect to the measured time.

A summary of our findings regarding the accuracy of the *BSP*-like functions and *F* is shown in Tables 1-3, where we report the maximum value of ERR over all runs (when omitted, the average values of ERR are similar). Overall, the *F* function is clearly superior to the *BSP*-like functions. The validations on synthetic access patterns (Table 1) underscore that disregarding hierarchy effects has a significant negative impact on predictive accuracy. Moreover, *F*'s overall high accuracy suggests that phenomena that were disregarded when designing it (such as some types of coherency overhead) have only a minor impact on performance. Since the sorting algorithms (Table 3) exhibit a high degree of locality, we would expect the optimistic versions of the *BSP*-like functions to perform much better than their pessimistic counterparts, and indeed this is the case (errors are not shown for $EBSP_{\min}$ and $DXBSP_{\min}$ because they are almost identical to the errors for $QSM_{\min}$). A similar situation occurs for the MPI-based NAS benchmarks (Table 2).

Synthetic Access Patterns – ERRs			
Function		AVG ERR	MAX ERR
QSM	MIN	24.15	88.02
	MAX	53.85	636.79
EBSP	MIN	24.15	88.02
	MAX	27.29	648.35
DXBSP	MIN	6.36	31.84
	MAX	34.8	411.46
F		1.19	1.91

Table 1. Synthetic Access Patterns.

NAS Parallel Benchmarks – MAX ERR					
	QSM		*EBSP*	*DXBSP*	
NPB	MIN	MAX	MAX	MAX	*F*
CG	2.46	258.31	210.10	166.91	1.46
EP	2.42	262.53	252.32	169.64	1.02
FT	2.05	309.40	245.64	199.92	1.63
IS	1.57	404.81	354.47	261.57	1.39
LU	2.15	295.01	236.80	190.62	1.32
MG	1.57	403.48	289.11	260.71	1.73
BT	2.77	229.68	189.08	148.41	1.05
SP	2.13	298.69	194.21	193.00	1.05

Table 2. NAS Parallel Benchmarks.

Sorting Programs – MAX ERR						
		QSM		*EBSP*	*DXBSP*	
Sort	**SStep**	MIN	MAX	MAX	MAX	*F*
Radix	SS1	2.12	400.14	320.48	258.55	1.41
	SS4	2.72	321.56	302.11	207.78	1.39
Sample	SS2	2.17	320.95	252.25	207.38	1.15
	SS4	2.89	287.72	247.31	185.91	1.11
	SS5	2.58	361.36	327.08	233.49	1.26
Column	SS1	3.44	268.13	205.23	173.25	1.06
	SS2	2.46	268.13	264.49	173.25	2.05
	SS3	2.88	230.37	228.11	148.85	1.88
	SS4	2.61	245.56	247.10	158.67	2.09
	SS5	1.36	484.93	280.03	313.34	1.16

Table 3. Sorting algorithms for selected supersteps.

Sort	**SStep**	**Errors for *F* (Measured)**		**Errors for *F* (Estimated)**	
		AVG	MAX	AVG	MAX
Radix	SS1	1.22	1.32	1.01	1.09
	SS4	1.11	1.16	1.13	1.16
Sample	SS2	1.05	1.09	1.20	1.21
	SS4	1.06	1.11	1.03	1.04
	SS5	1.13	1.17	1.24	1.26
Column	SS1	1.12	2.49	1.17	1.72
	SS2	1.80	1.89	2.02	2.12
	SS3	1.66	1.69	1.84	1.87
	SS4	1.78	1.83	2.05	2.06
	SS5	1.16	1.17	1.88	1.90

Table 4. Sorting algorithms: comparison of *F*'s accuracy with measured vs. estimated counters.

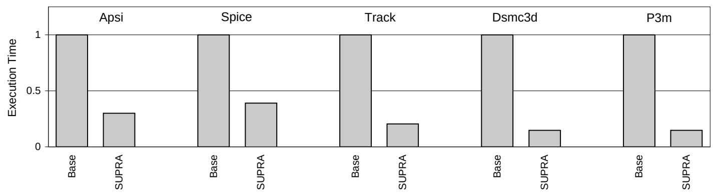
Performance predictions from a HW counter-based model. One of the advantages of the *BSP*-like functions over the counter-based function *F*, is that, to a large extent, the compiler or programmer can determine the input values for the function. While the counter-based function exhibits excellent accuracy, it seems that one should actually run the program to obtain the required counts, which would annihilate its potential as a performance predictor. However, if one could guess the counter values in advance with reasonable accuracy, they could then be plugged into *F* to obtain accurate predictions. For example, in some cases meaningful estimates for the counters might be derived by extrapolating values for large problem sizes from pilot runs of the program on small input sets (which could be performed at run-time by the adaptive system). To investigate this issue, we developed least-squares fits for each of the counters used in *F* for those supersteps in our three sorting algorithms that had significant communication. The input size n of the sorting instance was used as the independent variable. For each counter, we obtained the fits on small input sizes ($n/p = 10^5 \cdot i$, for $1 \leq i \leq 5$), and then used the fits to forecast the counter values for large input sizes ($n/p = 10^5 \cdot i$, for $5 < i \leq 10$). These estimated counter values were then plugged in *F* to predict the execution times for the larger runs. The results of this study are summarized in Table 4. It can be seen that in *all* cases, the level of accuracy of *F* using the extrapolated counter values was not significantly worse than what was obtained with the actual counter values. These preliminary results indicate that at least in some situations a hardware counter-based function does indeed have potential as an *a priori* predictor of performance. Currently, we are working on applying this strategy to other architectures, including the HP V-Class and the SGI Origin 2000.

5 Hardware

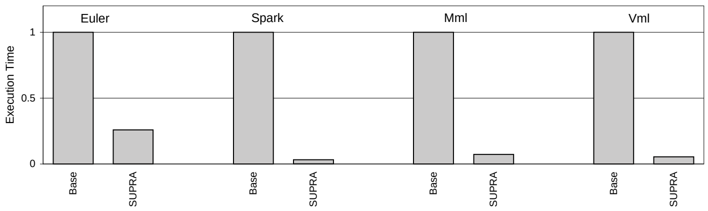
Smart applications exploit their maximum potential when they execute on reconfigurable hardware. Reconfigurable hardware provides some hooks that enable it to work in different modes. In this case, smart applications, once they have determined their true behavior statically or dynamically, can actuate these hooks and conform the hardware to its most desirable state for the application. The result is large performance improvements.

A promising area for reconfigurable hardware is the hardware cache coherence protocol of a CC-NUMA multiprocessor. In this case, we can have a base cache coherence protocol that is generally high-performing for all types of access patterns or behaviors of the application. However, the system can also support other specialized cache coherence protocols that are specifically tuned to certain application behaviors. Applications should be able to select the type of cache coherence protocol used on a code section basis. We provide two examples of specialized cache coherence protocols here. Each of these specialized protocols is composed of the base cache coherence transactions plus some additional transactions that are suited to certain functions. These two examples are the *speculative parallelization protocol* and the *advanced reduction protocol*.

The speculative parallelization protocol is used profitably in sections of a program where the dependence structure of the code is not analyzable by the compiler. In this case, instead of running the code serially, we run it in parallel on several processors. The speculative parallelization protocol contains extensions that, for each protocol transaction, check if a dependence violation occurred. Specifically, a dependence violation will occur if a logically later thread reads a variable before a logically earlier thread writes to it. The speculative parallelization protocol can detect such violations because it tags every memory location that is read and written, with the ID of the thread that is performing the access. In addition, it compares the tag ID before the access against the ID of the accessing thread. If a dependence violation is detected, an interrupt runs, repairs the state, and restarts execution. If such interrupts do not happen too often, the code executes faster in parallel with the speculative parallelization protocol than serially with the base cache coherence protocol. More details can be found in [36–38].



(a)



(b)

Figure 5. Execution time improvements by reconfiguring the cache coherence protocol hardware to support (a) speculative parallelization, and (b) advanced reduction operations.

The advanced reduction protocol is used profitably in sections of a program that contain reduction operations. In this case, instead of transforming the code to optimize these reduction operations in software, we simply mark the reduction variables and run the unmodified code under the new protocol. The protocol has extensions such that, when a processor accesses the reduction variable, it makes a privatized copy in its cache. Any subsequent accumulation on the variable will not send invalidations to other privatized copies in other caches. In addition, when a privatized version is displaced from a cache, it is sent to its original memory location and accumulated onto the existing value. With these extensions, the protocol reduces to a minimum the amount of data transfer and messaging required to perform a reduction in a CC-NUMA. The result is that the program runs much faster. More details can be found in [38].

We now see the impact of cache coherence protocol recongurability on execution time. Figure 5 compare the execution time of code sections running on a 16-processor simulated multiprocessor like the SGI Origin 2000 [30]. We compare the execution time under the base cache coherence protocol and under a reconfigurable protocol called SUPRA. In Figure 5(a), SUPRA is reconfigured to be the speculative parallelization protocol, while in Figure 5(b), SUPRA is reconfigured to be the advanced reduction protocol. In both charts, for each application, the bars are normalized to the execution time under Base.

From the figures, we see that the ability to reconfigure the cache coherence protocol to conform to the individual application's characteristics is very beneficial. The code sections that can benefit from the speculative parallelization protocol (Figure 5(a)), run on average 75% faster under the new protocol. The code sections that can benefit from the advanced reduction protocol (Figure 5(b)) run on average 85% faster under the new protocol.

6 Conclusions and Future Work

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.

One problem assigned to the OPTIMIZER is to compute how the application's data should be laid out in the memory and I/O systems of a given configuration of the system to minimize latencies, and therefore, execution time. The system we are developing is based on the *FORUM* system [16, 28, 29] developed by the Storage Systems Program (SSP) group at Hewlett-Packard Laboratories. This system produces an assignment of workloads to large storage devices such as disks and disk arrays. The assignment is made by an analytical constraint solver that attempts to minimize execution time (latencies) and/or to minimize the number (or expense) of storage devices required. Clearly, there is a large similarity between this problem, and the broader memory and I/O system layout problem considered here. The key challenge in generalizing the system lies in the characterization of the access patterns.

The performance of parallel applications is very sensitive to the type and quality of operating system services. We therefore propose to further optimize SmartApps by interfacing them with an existing customizable OS. While there have been several proposals of modular, customizable OSs, we plan to use the K42 [4] experimental OS from IBM, which represents a commercial-grade development of the TORNADO system [21, 5]. Instead of allowing users to actually alter or rewrite parts of the OS and thus raise security issues, the K42 system allows the selective and parametrized use of OS modules (objects). Additional modules can be written if necessary but no direct user access is allowed to them. This approach will allow our system to configure the type of services that will contribute to the full optimization of the program.

So far we have presented various run-time adaptive techniques that a compiler can safely insert into an executable under the form of multiversion code, and that can adapt the behavior of the code to the various dynamic conditions of the data as well as that of the system on which it is running. Most of these optimization techniques have to perform a test at run-time and decide between multi-version sections of code that have been pre-optimized by the static compilation process. The multi-version code solution may, however require an impractical number of versions. Applications exhibiting partial parallelism could be greatly sped up through the use of selective, point-to-point synchronizations and whose placement information is available only at run-time. Motivated by such ideas we plan on writing a two stage compiler. The first will identify which performance components are input dependent and the second stage will compile at run time the best solution. We will target what we believe are the most promising source of performance improvement for an application executing on a large system: increase of parallelism, memory latency and I/O management. In contrast to the run-time compilers currently under development [20, 6, 13] which mainly rely on the benefits of partial evaluation, we are targeting very high level transformations, e.g., parallelization, removal of (several levels) of indirection and algorithm selection.

References

1. S. Adve, V. Adve, M. Hill, and M. Vernon. Comparison of Hardware and Software Cache Coherence Schemes. In *Proc. of the 18th ISCA*, pp. 298–308, June 1991.
2. N. M. Amato, J. Perdue, A. Pietracaprina, G. Pucci, and M. Mathis. Predicting performance on smps. a case study: The sgi power challenge. In *Proc. IPDPS*, pp. 729–737, May 2000.
3. N. M. Amato, A. Pietracaprina, G. Pucci, L. K. Dale, and J. Perdue. A cost model for communication on a symmetric multiprocessor. TR 98-004, Dept. of Computer Science, Texas A&M University, 1998. A preliminary version was presented at the *SPAA'98 Revue*.
4. M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm. Customization lite. In *Proc. of 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, May 1997.
5. J. Appavo B. Gamsa, O. Krieger and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. of OSDI*, 1999.
6. et. al. B. Grant. An evaluation of staged run-time optimizations in Dyce. In *Proc. of the SIGPLAN 1999 PLDI, Atlanta, GA*, May 1999.
7. D. Bhandarkar and J. Ding. Performance Characterization of the Pentium Pro Processor. In *Proc. of HPCA III*, February 1997.
8. G.E. Blelloch, P.B. Gibbons, Y. Mattias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Trans. Par. Dist. Sys.*, 8(9):943–958, 1997.
9. J. Mark Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *EUROPAR98*, Sept., 1998.
10. D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. ACM SIGPLAN PPOPP*, pp. 1–12, 1993.
11. F. Dang and L. Rauchwerger. Speculative parallelization of partially parallel loops. In *Proc. of the 5th Int. Workshop LCR 2000, Lecture Notes in Computer Science*, May 2000.
12. F. Dehne, A. Fabri, and A. Rau–Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. ACM Symp. Comput. Geom.*, pp. 298–307, 1993.
13. Dawson Engler. Vcode: a portable, very fast dynamic code generation system. In *Proc. of the SIGPLAN 1996 PLDI Philadelphia, PA*, May 1996.
14. D. Bailey *et al.* The NAS parallel benchmarks. *Int. J. Supercomputer Appl.*, 5(3):63–73, 1991.
15. P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging-model for parallel computation? In *Proc. ACM SPAA*, pp. 72–83, 1997.
16. R. Golding, E. Shriver, T. Sullivan, and J. Wilkes. Attribute managed storage. In *Workshop on Modeling and Specification of I/O*, 1995.
17. H. Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *PACT'98*, Oct. 1998.
18. Ravi Iyer, Nancy Amato, L. Rauchwerger, and Laxmi Bhuyan. Comparing the memory system performance of the HP V-Class and SGI Origin 2000 multiprocessors using microbenchmarks and scientific applications. In *Proc. of ACM ICS*, pp. 339–347, June 1999.

19. B. H. H. Juurlink and H. A. G. Wijshoff. A quantitative comparison of parallel computation models. In *Proc. of ACM SPAA*, pp. 13–24, 1996.
20. D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. TR UWCSE 91-11-04, Dept. of Computer Science and Engineering, Univ. of Washington, Nov. 1991. .
21. O. Krieger and M. Stumm. Hfs: A performance-oriented flexible file system based on building-block compositions. *IEEE Trans. Comput.*, 15(3):286–321, 1997.
22. S. Owicki and A. Agarwal. Evaluating the performance of software cache coherency. In *Proc. of ASPLOS III*, April 1989.
23. L. Rauchwerger, N. Amato, and D. Padua. Run–time methods for parallelizing partially parallel loops. In *Proc. of the 9th ACM ICS, Barcelona, Spain*, pp. 137–146, July 1995.
24. 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.
25. L. Rauchwerger. Run–time parallelization: A framework for parallel computation. TR UIUCDCS-R-95-1926, Dept. of Computer Science, Univ. of Illinois, Urbana, IL, Sept. 1995.
26. L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Par. and Dist. Systems*, 10(2), 1999.
27. L. Rauchwerger and D. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. In *Proc. of 9th IPPS*, April 1995.
28. E. Shriver. *Performance Modeling for Realistic Storage Devices*. PhD Thesis, Computer Science Dept., New York Univ., 1997.
29. E. Shriver, A. Merchant, and J. Wilkes. An analytic behavior model for disks drives with readahead caches and request reordering. In *Proc. ACM SIGMETRICS*, pp. 182–191, 1998.
30. Silicon Graphics, Corporation, http://techpubs.sgi.com/library/. *Performance Tuning Optimizaton for Origin2000 and Onyx2*.
31. Silicon Graphics Corporation 1995. *SGI Power Challenge: User's Guide*, 1995.
32. R. Simoni and M. Horowitz. Modeling the Performance of Limited Pointer Directories for Cache Coherence. In *Proc. of the 18th ISCA*, pp. 309–318, June 1991.
33. J. Torrellas, J. Hennessy, and T. Weil. Analysis of Critical Architectural and Programming Parameters in a Hierarchical Shared Memory Multiprocessor. In *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp. 163–172, May 1990.
34. L. Valiant. Bridging model for parallel computation. *Comm. of the ACM*, 33(8):103–111, 1990.
35. H. Yu and L. Rauchwerger. Adaptive reduction parallelization. In *Proc. of the 14th ACM ICS, Santa Fe, NM*, May 2000.
36. Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proc. of HPCA-4*, pp. 162–173, 1998.
37. Y. Zhang, L. Rauchwerger, and J. Torrellas. Speculative Parallel Execution of Loops with Cross-Iteration Dependences in DSM Multiprocessors. In *Proc. of HPCA-5*, Jan. 1999.
38. Ye Zhang. *DSM Hardware for Speculative Parallelization*. Ph.D. Thesis, Department of ECE, Univ. of Illinois, Urbana, IL, Jan. 1999.