Prediction and Adaptation in Active Harmony[1]

Jeffrey K. Hollingsworth and Peter J. Keleher
University of Maryland
{hollings,keleher}@cs.umd.edu

We describe the design and early functionality of the Active Harmony global resource management system. Harmony is an infrastructure designed to efficiently execute parallel applications in large-scale, dynamic environments.

Harmony differs from other projects with similar goals in that the system automatically adapts ongoing computations to changing conditions through online reconfiguration. This reconfiguration can consist of system-directed migration of work at several different levels, or automatic application adaptation through the use of tuning options exported by Harmony-aware applications.

We describe early experience with work migration at the level of procedures, processes and lightweight threads.

1. Introduction

Meta-computing, the simultaneous and coordinated use of semi-autonomous computing resources in physically separate locations, is increasingly being used to solve large-scale scientific problems. By using a collection of specialized computational and data resources located at different facilities around the world, work can be done more efficiently than if only local resources were used. However, the infrastructure to support this type of global-scale computation is not yet available.

Both meta-computer environments and the applications that run on them can be characterized by distribution, heterogeneity, and changing resource requirements and capacities. These attributes make static approaches to resource allocation unsuitable. Systems need to dynamically adapt to changing resource capacities and application requirements in order to achieve high performance in such environments.

We are designing and building Active Harmony, a software architecture that manages distributed execution of computational objects in such environments. Our primary focus is on the following three areas:

- **Support for dynamic execution environments**: Dynamic adaptation to network and resource capacities, both when computational objects are created, and when application requirements or resource capacities change. Active Harmony attempts to maximize *data affinity* (moving computation near its data) and load balancing through intelligent resource allocation and object migration. Harmony supports an extensible *metric interface* that permits the sharing of resource capacity and utilization information among components in a distributed system.
- **Application adaptation**: A measurement and feedback system that adapts computational objects and their execution environment to improve the overall performance of the distributed system via runtime adaptation of algorithms, data distribution, and load balancing. Active Harmony exports a detailed metric interface to applications, allowing them to access processor, network, and operating system parameters. Applications export tuning options to the system, which can then automatically optimize resource allocation. Measurement and tuning therefore become first class objects in the programming model. Programmers can write applications that include ways to adapt computation to observed performance and changing conditions.
- **Shared-data interfaces**: Active Harmony supports single-system semantics among computational objects regardless of location, including consistent shared data segments. Shared data segments allow both peer-to-peer and client-server computations to exploit the simplified programming model and fine-grained sharing permitted by shared-memory environments. Innovations include support for heterogeneity of both data and program code, and support for the dynamic execution environment. Harmony's shared data interface also includes methods of measuring the *data affinity* between arbitrary objects.

The Active Harmony system is targeted at long-lived and persistent applications. Examples of long-lived applications include scientific code and data mining applications. Persistent applications include file servers,

[1] Supported in part by NSF awards ASC-9703212, CCR-9624803 and ACI-9711364.

information servers, and database management systems. We target long-lived applications because they persist long enough for the global environment to change, and hence have higher potential for improvement.

Our emphasis on long-lived applications allows us to focus on relatively expensive operations such as object migration. The cost of these operations can be amortized across the life of the object. Additionally, in order to implement a feed-back loop that does not oscillate out-of-control, changes must be implemented gradually in order to allow the effects of one change to settle before other changes are made.

Inherent to our assumption that we can adapt long-lived parallel applications is the belief that the past performance of an application is a good predictor of future behavior. We feel this is true because many programs tend to be cyclic or iterative in nature. As a result, if we can measure a program for one cycle, we can then adapt it for the next cycle and hope to improve its performance. Examples of applications that have this behavior include most scientific codes and periodic requests to information and database severs.

Resource management in meta-computing environments is a complex task. This paper focuses on the problem of developing metrics to measure changing computational needs and the techniques to react to them. Section 2 describes the overall architecture of the Harmony System. Section 3 describes Harmony's use of LBF, a performance metric designed to predict the impact of migrating both process-level and procedure-level computation tasks. Section 4 describes Harmony's approach to communication minimization through thread migration. Finally, Section 5 discusses related work and Section 6 concludes.

2. Scheduling and adaptation

Traditional approaches to resource management, such as space sharing and gang scheduling, rely on the system being static and under the full control of a centralized scheduler. They also rely on all system jobs being homogenous, at least to the level of the types of information that can be obtained from the jobs. Harmony's target environment is dynamic, heterogeneous, and can contain sub-systems that are not fully under Harmony's control. Furthermore, we intend to exploit application-specific information to improve resource utilization.

For example, the underlying system can obtain sharing information about threads in distributed shared memory (DSM) applications. Another application might explicitly export tuning options to the system. These options might allow the system to adapt the application to available memory and CPU resources. Such disparate types of information present inherently different challenges and opportunities to schedulers, and could probably be best handled in a decentralized fashion. However, global schedulers provide a single point where broad policy decisions can be made

2.1 Adaptation policies

The primary motivation of Harmony's application and environment characterizations is their use in improving resource utilization and throughput. Traditional approaches to scheduling, such as space sharing and gang-scheduling, rely on the system being static and under the full control of the scheduler. Our target environment is neither. Among the issues that we address are the following:

1) **centralized versus decentralized control** - Centralized control usually implies better decision-making because of more complete information. However, up-to-date information is expensive to collect in large or dynamic systems. Since our target environments are both, we expect the ability to profitably use slightly stale system information to be important.
2) **data-shipping versus function-shipping** - At any point where computational objects interact with either remote data sources or objects, the system can potentially improve performance by migrating the objects closer together. The systems must determine when this is appropriate, and in what fashion any migration would be accomplished.
3) **throughput versus response time** - While throughput is likely to be the most important measure of system performance, the application mix will include applications for which response time is important, and the system might also be forced to coexist with applications that are both outside the control of the system, and whose performance must not be degraded.

Harmony supports both domain-specific decision processes and global policies through hierarchical resource management. The *adaptation controller* exports broad policy choices through a domain-independent interface to *domain schedulers*. Similarly, domain-independent resource and capacity information is passed from the domain schedulers up to the adaptation controller. Each domain scheduler translates global policies into policies specific to the type of application it controls.

2.2 Harmony's structure

Harmony's structure is shown in Figure 1. The major components are the following:

Adaptation controller

The adaptation controller is the heart of the system. The controller must gather relevant information to be used as

input, project the effects of proposed changes (such as migrating an object) on the system, and weigh competing costs and expected benefits of making various changes.

The design of the adaptation controller has yet to be finalized. However, the system will probably use a pattern-matching approach. Patterns will be used categorize potential problems into specific problem *domains*, and to apply sophisticated domain-specific optimizations. For example, if the adaptation controller contained a predicate that indicates the system is suffering from poor load balance, the pattern that recognizes the situation might trigger one or more load balancing algorithms. Together, the problem-recognition patterns and the domain-specific techniques will form a complete resource management system.

Active Harmony provides mechanisms for applications to export tuning options, together with information about the resource requirements of each option, to the adaptation controller. The adaptation controller then chooses among the exported options based on more complete information than is available to individual objects. A key advantage of this technique is that the system can tune not just individual objects, but also entire collections of objects. Possible tuning criteria include network latency and bandwidth, memory utilization, and processor time. Since changing implementations or data layout could require significant time, we propose to include a cost function that can be used by the tuning system to evaluate if a tuning option is worth the effort required.

Metric interface

The metric interface provides a unified way to gather data about the performance of applications and their execution environment. Data about system conditions and application resource requirements flow into the metric interface, and on to both the adaptation controller and individual applications.

Tuning interface

The tuning interface provides a method for applications to export tuning options to the system. Each tuning option defines the expected consumption of one or more system resources. The options are intended to be "knobs" that the system can use to adjust applications to changes in the environment. The main concern in designing the tuning interface is to ensure that it is expressive enough to describe the effects of all application tuning options.

3. Adaptation metrics

To make informed choices about adapting an application, Harmony needs metrics to predict the performance implications of any changes. To meet this need, we have developed a metric called Load Balancing Factor (LBF) to predict the impact of changing where computation is performed. This metric can be used by the system to evaluate potential application reconfigurations before committing to potentially poor choices.

We have developed two variants of LBF, one for process level migration, and one for fine-grained procedure level migration. Process Load Balancing Factor (LBF) predicts the impact of changing the assignment of processes to processors in a distributed execution environment. Our goal is to compute the potential improvement in execution time if we change the placement. Our technique can also be used to predict the performance

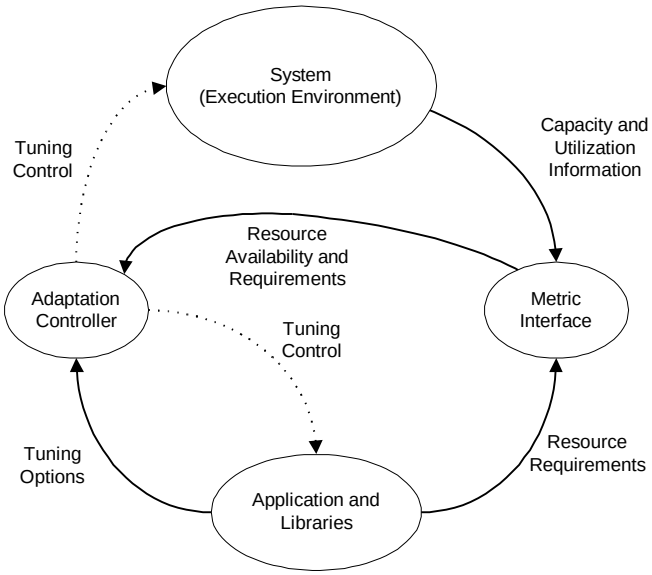

Figure 1: Major Components of Active Harmony

gain possible if new nodes are added. Also, we are able to predict how the application would behave if the performance characteristics of the communication system were to change.

To assess the potential improvement, we predict the execution time of a program with a virtual placement, during an execution on a different one. Our approach is to instrument application processes to forward data about inter-process events to a central monitoring station that simulates the execution of these events under the target configuration.

The details of the algorithm for process level-LBF are described in [1]. Early experience with process-LBF is encouraging. Figure 2 shows a summary of the measured and predicted performance for a TSP application, and four of the NAS benchmark programs [2]. For each application, we show the measured running time for one or two configurations and the predicted running time when the number of nodes is used. For all cases, we are able to predict the running time to within 6% of the measured time.

While process LBF is designed for course-grained migration, procedure-level LBF is designed to measure the impact of fine-grained moved of work. The goal of this metric is to compute the potential improvement in execution time if we move a selected procedure, F, from the client to the server or visa-versa.

Application Target	Meas. Time	Pred.	Error		Pred.	Error	
TSP			4/1			4/1	
4/4	85.6	85.5	0.1	(0.1%)	85.9	-0.3	(-0.4%)
4/1	199.2	197.1	2.1	(1.1%)	198.9	0.3	(0.2%)
EP - class A			16/16			16/8	
16/16	258.2	255.6	2.6	(1.0%)	260.7	-2.5	(-1.0%)
FT- class A			16/16			16/8	
16/16	140.9	139.2	1.7	(1.2%)	140.0	0.9	(0.6%)
IS- class A			16/16			16/8	
16/16	271.2	253.3	17.9	(6.6%)	254.7	16.5	(6.0%)
MG- class A			16/16			16/8	
16/16	172.8	166.0	6.8	(4.0%)	168.5	4.3	(2.5%)

Figure 2: Measured and predicted time for LBF. For each application, we show 1-2 target configurations. The second column shows the measured time running on this target configuration. The rest of the table shows the execution times predicted by LBF when run under two different actual configurations.

The algorithm used to compute procedure is based on the Critical Path (CP) of a parallel computation (The longest process time weighted path through the graph formed by the inter-process communication in the program). The idea of procedure LBF is to compute the new CP of the program if the selected procedure was moved from one process to another[2].

In each process, we keep track of the original CP and the new CP due to moving the selected procedure. We compute procedure LBF at each message exchange. At a send event, we subtract the accumulated time of the selected procedure from the CP of the sending process, and send the accumulated procedure time along with the application message. At a receive event, we add the passed procedure time to the CP value of the receiving process **before** the receive event. The value of the procedure LBF metric is the total effective CP value at the end of the program's execution. Procedure LBF only approximates the execution time with migration since we ignore many subtle issues such as global data references by the "moved" procedure. Figure 3 shows the computation of procedure LBF for a single message send. Our intent with this metric is to supply initial feedback to the programmer about the potential of a tuning alternative. A more refined prediction that incorporates shared data analysis could be run after our metric but before proceeding to a full implementation.

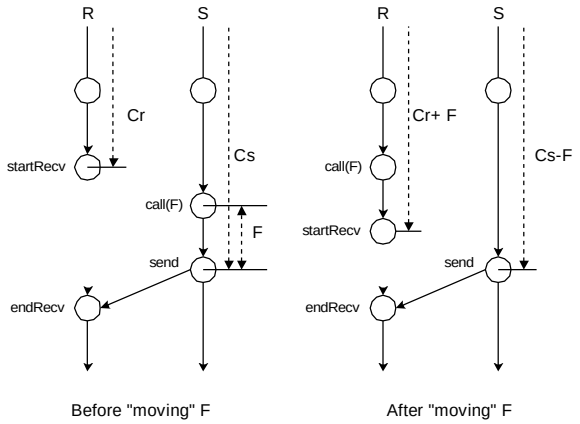

Figure 3: Computing procedure LBF - The PAG before and after moving the procedure F. The time for the procedure F is moved from the sending process (which is on the application's critical path) to the receiving one (which is not).

We created a Synthetic Parallel Application (SPA) that demonstrates a workload where a single server becomes the bottleneck responding to requests from three clients. In the server, two classes of requests are processed: *servBusy1* and *servBusy2*. *ServBusy1* is the service requested by the first client and *servBusy2* is the service requested by the other two clients.

[2] Our metric does not evaluate *how* to move the procedure. However, this movement is possible if the application uses Harmony's shared data programming model.

The results of computing procedure LBF for the synthetic parallel application are shown in Figure 4. To validate these results, we created two modified versions of the synthetic parallel application (one with each of *servBusy1* and *servBusy2* moved from the server the clients) and measured the resulting execution time[3]. The results of the modified programs are shown in the third column of Figure 4. In both cases, the error is small indicating that our metric has provided good guidance to the application programmer.

Procedure	Procedure LBF	Measured Time	Difference
ServBusy1	25.3	25.4	0.1 (0.4%)
ServBusy2	23.0	23.1	0.1 (0.6%)

Figure 4: Procedure LBF accuracy.

For comparison to an alternative tuning option, we also show the value for the Critical Path Zeroing metric [3]. CP Zeroing is a metric that predicts the improvement possible due to optimally tuning the selected procedure (i.e., reducing its execution time to zero) by computing the length of the critical path resulting from setting the time of the selected procedure to zero. We compare LBF with Critical Path Zeroing because it is natural to consider improving the performance of a procedure itself as well as changing its execution place (processor) as tuning strategies.

The length of the new CP due to the movement of *servBusy1* is 25.4 and the length due to *servBusy2* is 16.1 while the length of the original CP is 30.7. With the Critical Path Zeroing metric, we achieve almost the same benefit as tuning the procedure *ServBusy1* by simply moving it from the server to the client. Likewise, we achieve over one-half the benefit of tuning the *ServBusy2* procedure by moving it to the client side.

Procedure	LBF	Improvement	CP Zeroing	Improvement
ServBusy1	25.3	17.8%	25.4	17.4%
ServBusy2	23.1	25.1%	16.1	47.5%

Figure 5: Procedure LBF vs. CP Zeroing.

4. Adaptation via thread migration

LBF allows Harmony to evaluate the computational effects of moving procedures and processes among distinct nodes. This section discusses Harmony's ability to gather and use somewhat analogous information from shared-memory applications.

Harmony provides a shared memory abstraction to parallel applications running on networks of workstations. Systems with such support are commonly termed distributed shared memory (DSM) systems. DSM applications are multi-threaded, and assumed to have many more threads than the number of nodes used by any one application. Overall performance depends on parallelism, load balance, latency tolerance, and communication minimization. In this paper, we focus on communication minimization.

Communication results primarily from data sharing between threads. Hence, co-locating communicating threads on the same nodes can reduce communication. Harmony obtains sharing information through an *active correlation-tracking* mechanism [4]. Previous systems obtained page-level access information by tracking existing page faults. Page faults occur when local threads access invalid shared pages. Invalid pages are re-validated by fetching the latest version of the shared page from the last node that modified it. The underlying system keeps track of the thread that caused each page fault, slowly building up a pattern of the pages accessed by each thread. The *correlation* between a pair of threads is the number of shared pages that they access in common.

The problem is that there are usually multiple threads running on each machine, and these threads share state. Once the first thread on a node re-validates a given page, all other local threads can access the page without invoking the DSM system.

Hence, the system only gains partial information about the sharing behavior of local threads. Any migration decisions are therefore made with only partial information, often leading to bad long-term choices. Bad choices are discovered only after the threads have been migrated to other processors. Once a thread migrates off of a local host, the interactions between that thread and those left behind become visible in the form of network page faults (unless masked by the actions of other threads on the new node). These faults can be used to identify threads that should then be moved back to their original position, resulting in ping-ponging of threads across the system.

Active correlation-tracking avoids these problems through a one-time correlation-tracking phase. Briefly, the algorithm is as follows:

1) All pages are marked invalid.
2) At each access fault caused by the above step (a *correlation fault)*, the page is noted and the page's protection is returned to its original state.

[3] Since Harmony's shared data programming model is not yet fully implemented, we made these changes by hand.

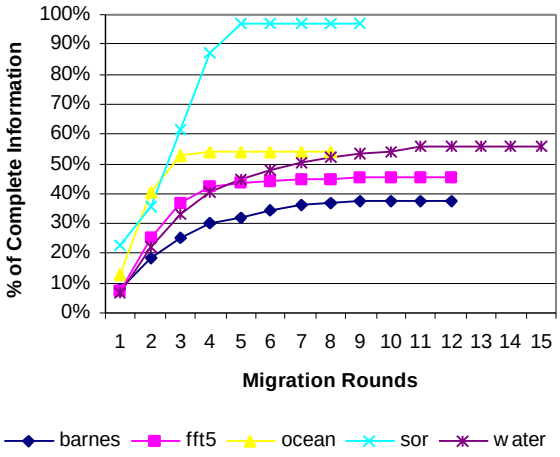

Figure 6: Passive information-gathering

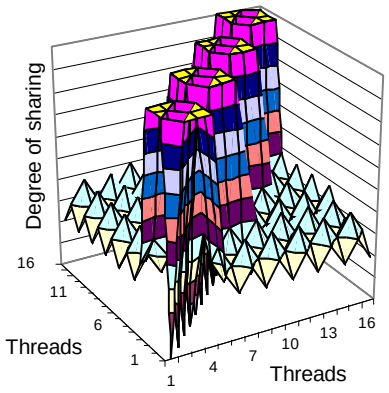

Figure 7: 16-thread FFT

3) At the next barrier, untouched pages are returned to their previous protection level.

After the tracking phase has ended, each processor has a complete record of all pages accessed by local threads.

We measure the performance impact of correlation-tracking on several applications from several standard DSM applications[4]. The tracking phase's primary overhead results from the correlation faults. However, because the faults are incurred in parallel across all nodes in the system, they cause an overhead of less than 20%. Furthermore, this overhead only occurs during the tracking phase. Since the tracking phase usually only occurs once at the beginning of an application's execution, this cost has a negligible effect on overall performance.

Note that active correlation-tracking gives Harmony complete sharing information without network communication. Hence, a new configuration can be implemented in only a single round of thread migrations. By contrast, Figure 6 shows that the passive approach requires an average of more than six rounds of mass thread migrations before the amount of information stabilizes. Each point shows the percentage of total information learned by the passive approach at that round. Even at the end of the migrations, the passive tracking only comes close to obtaining complete information for sor, by far the least complex of our applications.

Information obtained from the active correlation-tracking mechanism is used to create *correlation maps*, which summarize sharing information among all threads in the system. Figure 7 shows a 16-thread correlation map of a 3-D FFT with 64 by 64 by 16 data points. The map shows a well-defined structure in which all of the high peaks are concentrated along the diagonal. Extension along the *z* axis represents the correlation (in terms of the number of pages shared between threads *x* and *y*) between the two threads identified by the *x* and *y* coordinates. The majority of FFT's data sharing occurs inside four-thread groupings. The lowest peak represents sharing among threads 1-4, the next peak represents sharing between among nodes 5-8, etc. This map implies that a mapping of four threads to each of four processors would avoid network communication for the majority of the sharing between threads. However, an eight by two mapping would exclude the majority of the peaks, implying that it would cause much more communication mapping. What is not clear from the map is to what extent this communication advantage would translate into a performance advantage.

Given complete information on the sharing between threads, Harmony attempts to reduce communication by co-locating thread pairs that share the most data.

This problem is NP-complete, so we evaluated several heuristics for mapping specific threads to nodes. Briefly, we looked at both leader-based and leaderless variants of AscEdge, DesEdge, and DesNode [4]. AscEdge treats threads and their communication (or correlation) as nodes and edges of a weighted graph, respectively. We map threads to nodes by sorting edges by weight (communication cost) in ascending order. The threads representing the endpoints of each edge are put onto distinct nodes, if possible. Each thread is mapped to the node with which the thread has the highest aggregate correlation. DesEdge differs in that the edges are sorted in descending order, and threads are placed on the same nodes, if possible. DesNode sorts threads by aggregate correlation, and maps each thread to the node with which it has the highest aggregate correlation.

Figure 8 shows the communication costs that result from running each of the heuristics on information gathered through active correlation-tracking. The first five columns give communication costs in the number of pages shared by pairs of threads. The second set of five

[4] Please see [5, 6] for details of our test applications.

columns gives the number of kilobytes communicated per iteration, and the last five columns give the number of messages sent per iteration. The heuristics AscEdge, DesEdge, and DesNode are abbreviated 'ae', 'de', and 'dn', respectively. Leader-based variants are identified by '-l' suffixes. Additionally, we also show the communication costs of the optimal configuration ('opt'), and of a random configuration ('r').

There is a large amount of variation across the different applications, but de-l generally performs the best, and random the worst. On average, de-l obtains a solution within 0.3% of optimal. This performance appears to rest on two advantages. First, de-l is usually able to group threads connected by high-cost nodes together on the same nodes. Second, the leader-based approaches appear to help ensure that the nodes are filled at the same pace, rather than having the first node fill completely, then the second, etc. The problem with filling nodes unevenly is that it increases the chance that a high-cost edge has to be split across nodes because of one node being filled.

The relative communication cost magnitudes match up quite well with the number of bytes communicated and messages sent. However, the differences in communication cost are exaggerated in the byte and message totals. This implies that the page sharing addressed well by all of the heuristics causes relatively less communication than the pages that are handled better by some heuristics than others.

5. Related work

Process migration techniques have been investigated in depth [7-9]. Recent work has investigated using heterogeneous migration in conjunction with typesafe languages, and in situations where the type-safety of applications written in non-typesafe languages can be verified [10]. Harmony is different in that we focus on the policy issues of *when* to migrate large, long-running distributed applications; and whether to migrate the process or the data.

Farming computational objects to nodes of a distributed system has been exploited by many projects [11-19]. Several projects also allow process migration across homogeneous processor borders for purposes of load balancing. Dome [18] also allows migration across heterogeneous boundaries, but uses high-level checkpoints consisting of user-written routines that marshal and unmarshal significant data structure manually. Additionally, Dome only supports applications based on a library of parallel data structures; arbitrary parallel applications are not supported. The Harmony project is distinguished from these projects by the following factors: emphasis on dynamic environments, integration of application tuning options and semantic information into resource management algorithms, and inclusion of such into resource management algorithms.

Several studies [7] claim that load-balancing via process migration is not a viable strategy for improving performance, primarily because of large migration overhead. We believe that two changes in current systems make these results less relevant today than with systems of a decade ago. First, computationally intensive applications are now a mainstay on the type of workstations we are considering in our work. These applications have large resource demands, and sufficient execution time to amortize the cost of process migration. Second, the prevalence of parallel and client-server applications means that data affinity becomes at least as important as processor cycles for these applications. Hence, there is more potential for improved performance with current applications than with typical applications of a decade ago.

Globus [20] and Legion [21] also provide infrastructures that support program execution in a meta-computing environment. Harmony differs from these systems by focusing on developing metrics and algorithms for program adaptation, and by our use of shared-data interfaces as part of the programming model. Globus and Legion provide other essential services for meta-computing including naming, security, and communication. We are investigating using one or both of these systems as a test-bed for the resource management policies we are developing.

AppLeS [22] provides programmers with applica-

	# Shared Pages					Total KBytes Communicated					Data Request Messages				
	barnes	fft	ocean	sor	water	barnes	fft	ocean	sor	water	barnes	fft	ocean	sor	water
opt	99458	2240	12774	924	11696	21122	17096	49876	867	8165	15011	3583	25727	196	2441
ae	102121	2960	14176	928	12290	26885	29703	82810	982	13899	24649	6450	34965	224	4199
ae-l	99458	2960	14343	932	12192	21122	31748	77736	1098	13467	15011	6737	33557	252	4064
de	100918	2240	15526	924	12058	23450	17096	144028	867	11683	16873	3583	51818	196	3547
de-l	100416	2240	12774	928	11696	23347	19021	49876	982	8176	17494	3812	25727	224	2442
dn	101276	2240	17218	928	12407	24726	17190	162050	982	14555	18182	3583	61330	224	4106
dn-l	100416	2240	14447	924	12313	23348	17191	103845	867	13559	17502	3587	41049	196	4097
r	103723	3440	17067	992	12620	27632	51428	121948	2943	14979	22022	10974	40689	700	4158

Figure 8: Shared pages and performance

tion level scheduling. Harmony differs in that our focus is on providing middle-ware that provides metrics and mechanisms to alter the behavior of a program in execution. In AppLeS, the application programmer is supplied information about the computing environment [23] and given a library to let them react to the changes in available resources. In Harmony, we are trying to have the application supply alternative ways of executing the program and then let the runtime software select among these options based on observations of the environment.

6. Summary and discussion

We have described Active Harmony, a new infrastructure for managing resources in large, dynamic environments. One of Harmony's major strengths is its ability to gather and exploit many types of application-specific information. This information can then be used to automatically reconfigure running applications at several levels. This paper has concentrated on two such mechanisms. First, the LBF metric predicts the performance impact of moving procedures and processes. Second, active correlation- tracking is used to gather data sharing information and drive thread migration decision heuristics.

However, Harmony's strength is not in the individual mechanisms and metrics used, but in the interfaces that allow global policy decisions and many types of application-specific information to be tied together. Harmony is a work in progress, and we are continuing to evaluate new sources of information and new mechanisms for possible inclusion in the system.

7. References

[1] H. Eom and J. K. Hollingsworth, "LBF: A Performance Metric for Program Reorganization," in *The International Conference on Distributed Computing Systems*, May 1998.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, and D. S. Browning, "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, vol. 5, pp. 63-73, 1991.

[3] J. K. Hollingsworth, "An Online Computation of Critical Path Profiling," in *SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, Philadelphia, PA, 1996.

[4] K. Thitikamol and P. Keleher, "Thread Migration and Communication Minimization in DSM Systems," *The Proceedings of the IEEE*, 1998.

[5] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[6] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[7] D. Eager, E. Lazowska, and J. Zahorjan, "The Limited Performance Benefits of Migrating Active Processes," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '88)*, May 1988.

[8] F. Douglis and J. Ousterhout, "Process Migration in the Sprite Operating System," in *Proceedings of the 7th International Conference on Distributed Computing Systems*, September 1987.

[9] J. M. Smith, "A Survey of Process Migration Mechanisms," in *Operating Systems Review*, 1989.

[10] B. Steensgaard and E. Jul, "Object and Native Code Thread Mobility Among Heterogeneous Computers," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[11] G. Weikum, C. Hasse, A. Monkeberg, and P. Zabback, "The COMFORT Automatic Tuning Project," in *Information Systems*, 1994.

[12] J. Ju, G. Xu, and K. Yang, "An Intelligent Dynamic Load Balancer for Workstation Clusters," in *osr*, January 1995.

[13] S. Zhou, J. Wang, X. Zheng, and P. Delisle, "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," *SPE*, vol. 23, pp. 1305-1336, 1993.

[14] P. Mehra and B. Wah, "Automated Learning of Workload Measures for Load Balancing on a Distributed System," in *Int'l Conference on Parallel Processing*, 1993.

[15] B. Schnor, "Dynamic Scheduling of Parallel Applications," in *Lecture Notes in Computer Science 964, Parallel Computing Technologies, Third International Conference, PaCT-95*, 1995.

[16] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective Distributed Scheduling of Parallel Workloads," in *Sigmetrics'96 Conference on the Measurement and Modeling of Computer Systems*, 1996.

[17] B. C. Neuman and S. Rao, "The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems," in *Concurrency: Practice and Experience*, 1994.

[18] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, "Dome: Parallel programming in a heterogeneous multi-user environment," Carnegie Mellon University CMU-CS-95-137, March 1995 1995.

[19] M. Litzkow and M. Livny, "Experience with the Condor Distributed Batch System," in *The IEEE Workshop on Experimental Distributed Systems*, October 1995.

[20] I. Foster, N. Karonis, C. Kesselman, G. Koenig, and S. Tuecke, "A Secure Communications Infrastructure for High-Performance Distributed Computing," in *Proc. 6th IEEE Symp. on High-Performance Distributed Computing*, 1997.

[21] M. J. Lewis and A. Grimshaw, "The Core Legion Object Model," in *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[22] F. Berman and R. Wolski, "Scheduling from the Perspective of the Application," in *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, 1996.

[23] R. Wolski, "Dynamic Forecasting Network Performance Using the Network Weather Service," UCSD, La Jolla, CA TR-CS96-494, May 1997.