Adaptive algorithms for managing a distributed data processing workload

by J. Aman
C. K. Eilert
D. Emmes
P. Yocom
D. Dillenberger

Workload management, a function of the OS/390™ operating system base control program, allows installations to define business objectives for a clustered environment (Parallel Sysplex™ in OS/390). This business policy is expressed in terms that relate to business goals and importance, rather than the internal controls used by the operating system. OS/390 ensures that system resources are assigned to achieve the specified business objectives. This paper presents algorithms developed to simplify performance management, dynamically adjust computing resources, and balance work across parallel systems. We examine the types of data the algorithms require and the measurements that were devised to assess how well work is achieving customer-set goals. Two examples demonstrate how the algorithms adjust system resource allocations to enable a smooth adaptation to changing processing conditions. To the customer, these algorithms provide a single-system image to manage competing workloads running across multiple systems.

Although there has been an important role for a computing environment that has a single machine and a single copy or image of an operating system, a number of factors have converged to motivate use of multiple machines, especially when connected in a parallel fashion. These machines could be logical or physical configurations of what might otherwise be deemed a single machine, but each is controlled by a separate copy of an operating system.

Among the reasons for this interest in parallelism are:

- To increase total computing power available over a single image so as to reduce individual response time or to handle larger volumes of work, or both
- High availability due to the expectation that failure of any single component, at whatever level, will not cause the loss of all computing capabilities
- Access to lower-cost technology to use as building blocks for a larger system
- Ability to grow the total computing power in small increments to address needs as they arise with small incremental cost and no outage required

There is an obvious increase in complexity with the introduction of multiple images. It is natural to want to view them as cooperating and sharing resources. Considerable simplification results by seeing these multiple images as a single computing environment and having one set of controls rather than separate

0018-8670/97/$5.00 © 1997 IBM

controls for each image. System parameters previously requiring a human operator to monitor and set them are now controlled by the workload management (WLM) algorithms described in this paper. Workloads are dynamically balanced across images. WLM tracks those factors needed to best place incoming work and provides interfaces to make workload-balancing recommendations.

In a parallel environment, the WLM objective to simplify performance management while effectively using all computing resources poses a number of design problems that must be addressed. Given that some external controls are needed to reflect business goals and importance, but that low-level controls are not provided, the system must decide which resources to allocate to which work requests. It is up to the system to calculate how much of those resources to give and for how long a time. With due consideration for the danger of thrashing, it is up to the system to determine how often to make those changes and whether all the changes should be made at the same time.

With respect to the problem of balancing work across a parallel environment, the system must choose where to run each work request given the following constraints:

- Goals need to be achieved.
- Goals may not be known in advance.
- Resource requirements are unknown.
- Other work requests will be concurrently demanding resources in competition with new work requests that are also unknown.

Another problem addressed by the workload management algorithms is maximizing the use of resources across the parallel environment, especially where there are diverse machine sizes—the problem of configurational heterogeneity discussed in Reference 1. Finally, the underlying configuration must be concealed from end users and changes made transparent to them while allowing load balancing across equivalent servers.[2]

In this paper, the next section describes related work in resource management and workload balancing. Then WLM concepts and the system model are described. The section following that one describes the WLM algorithms used for goal-oriented resource management. The two subsequent sections describe the WLM approach for balancing work across the parallel environment and the products that cooperate to realize the benefits of the WLM philosophy. The paper concludes with a summary of the current state of the art and some outstanding problems that are yet to be addressed.

Related work

A number of alternatives[3] exist in deciding how system resources, such as access to the processor or processor storage, or both, should be allocated among eligible work. One possibility is that *no controls* are offered at all—the system queues and dispatches work automatically. The absence of external controls offers maximal simplicity and may well be adequate if the system is dedicated to a small number of work requests and is of sufficient capacity to handle all work quickly enough to satisfy the appropriate parties. This approach may also be sufficient if the system implements techniques to modify access to system resources as individual work requests "age," i.e., are observed to consume higher levels of resources. However, satisfaction with this approach will depend on how closely the system anticipates and implements the wishes or expectations of the end user(s) or installation in the absence of any external control. As the mixture of work in the system becomes increasingly diverse, with more complex human expectations on what should happen, the absence of any human control becomes less tenable.

A second approach is to *keep the system available for an "owner,"* thus protecting access of this special user to the system. This approach is more suitable to small systems but has a number of implementations. Condor[1] is a system that allows workstations to be used by others when idle but it checkpoints and preempts "foreign" work when the "owner" wants access to the machine. The Butler[1] system has a similar philosophy and will actually terminate "foreign" work when the "owner" wants access to the machine. Utopia, from the University of Toronto, also provides an option that allows the system to reject remote work when the "owner" needs the system back.[1] As a category, these implementations provide a limited partitioning of work as either "owner" or "nonowner," with no finer granularity for ranking work within or across these groupings.

A third approach is to *optimize system resources* so as to "keep the machine busy." Utopia[1] allows specification of a threshold beyond which "foreign" work is not accepted but otherwise is happy to offer service to all. This approach is an extension of the prior technique where a threshold of zero would be used

for the amount of "nonowner" work that coexists with "owner" work.

A fourth approach is to *minimize response time*. This method is the implicit control in Reference 4. Although minimizing response time may seem desirable, it does not address conditions where not all work is equally important and misses the opportunity to make trade-offs to optimize some work at the expense of other work.

Once a wide variety of work requests can be in concurrent execution, it may not be sufficient to merely keep the machine busy. This probability suggests that the system administrator may want or need to control the priority of access to resources. One approach is to allow *specification of low-level "how to" performance controls,* exemplified by releases of MVS prior to Multiple Virtual Storage/Enterprise Systems Architecture System Product 5.1 (MVS/ESA* SP5.1) and by compatibility mode in MVS/ESA Version 5. Virtual Machine/Enterprise Systems Architecture* (VM/ESA*)[5] is a second operating system using this approach. Utopia also allows specification of priority controls.[1]

The second major approach in organizing multiple images is one of shared data and shared work.

The preceding paragraphs discussed how resources would be managed on behalf of work requests in the system. We now discuss alternatives for organizing multiple images on behalf of a workload. There seem to be two primary choices in this regard.

The first major scheme is to partition individual images into clusters, based on some attribute. One approach is to cluster images so that each cluster runs similar work or even the same "job." In the Scalable POWERparallel System 2 (SP2*),[6] some images function as server nodes, whereas others run individual work requests. Each node is separately configurable in terms of I/O, memory, and CPU capability. SP2 allows a system administrator to define separate pools of machines that are available to parallelize a particular job, run interactive users, or run nonparallelized jobs. Currently there is no support for time sharing or preemptive scheduling.

A second clustering approach is based on data affinity wherein each image is given ownership of a distinct set of persistent data (files, databases, etc.). The Tandem system and NCR 3600** system and Reference 4 all embody such an approach. The limitations of this approach are discussed in other papers.[7]

The second major approach in organizing multiple images is one of shared data and shared work. For example, while Utopia assumes global file access,[1] it uses geographic proximity (sometimes virtual proximity) to cluster images in the network. Specific resource requirements are kept in a system-provided file, which must be managed by system administrators, presumably with input from application owners who are aware of their own requirements. Reference 7 also assumes a data-sharing environment.

Other platforms need to assume that system capacity is configured for peak load, due to data affinity and the natural imbalance that will occur for real-world computing environments. This implies that those platforms require considerable excess capacity at off-peak times, which yields substantial advantage to WLM where trade-offs can be made that reflect the intended use of computing resources according to business needs.

Once a parallel environment where multiple images are capable of handling a given work request exists, the question arises as to which image should be chosen. The decision as to where each work request should be placed and how to best choose the target image involves a number of trade-offs between what information is available and what resource management philosophy and controls are provided.

In the SP2 world, interactive users may be spread across nodes that are lightly loaded. Batch jobs may be submitted via IBM LoadLeveler* or NQS/MVS* (Network Queuing System/Multiple Virtual Storage), although parallel jobs may only be submitted by the former. LoadLeveler[2] attempts to balance work across a set of SP2 nodes by using:

- Job classes—Defined by the system administrator, jobs can be classified as short running, long running, etc.

- Job priority—How important a job is as defined by the owner's group, userid, and class. The priority of a job will determine whether LoadLeveler will schedule this job ahead of or behind existing queued jobs.

The LoadLeveler component—Interactive Session Support (ISS)—balances log-ins and application sessions across multiple servers based on factors such as link speed, number of connections, overall system load, and, optionally, machine speed. Although this is periodically reevaluated, there is no feedback to ensure that the recommendation reflects actual responsiveness.

Utopia performs load balancing under a dynamic algorithm that uses load indices for CPU queue lengths, free memory, disk I/O transfer rates, disk space, and number of concurrent users.[1] Other metrics may be used at the discretion of the installation, and applications are free to use their own metrics, although it seems that using different metrics would cause problems since different programs may be at cross-purposes in their routing approaches. A further challenge to Utopia's support is how to combine metrics into a single usable measure vs the more complex load vector proposed.

Utopia utilizes a "master" image to coordinate load data and in some schemes to make load decisions.[1] After placing each new work request, Utopia incorporates a load adjustment factor to account for latent demand. General resource demands are described in a system-provided file, though usually on an exception basis. It is unlike WLM, where resource demands are not assumed to be known in advance. Utopia is intended to balance across potentially thousands of hosts, at which point the projected overhead is estimated to be 1 percent. With up to dozens of hosts, the overhead for balancing under Utopia is less than 0.5 percent.

Reference 4 assesses several alternatives to route work based on some knowledge of data access patterns and evaluates the sensitivity of the algorithms to incorrect information. The base algorithm against which all others are compared involves tracking where each transaction is routed, by class, and projecting what its expected response time will be on the basis of system parameters and static transaction attributes and then choosing the image that minimizes the expected response time. The paper shows that this algorithm is quite sensitive to these values, which is disconcerting in view of the practical difficulty of ascertaining these values and their tendency to change over time. The algorithm has a further tendency to overlook the cost of routing to an image that does not own the data used by the transaction.

The first alternative investigated in Reference 4 applies a threshold so that data affinity is enforced in routing unless the target image is overloaded, i.e., its projected response time is above the estimated optimal choice by a certain threshold percentage. The threshold approach is always superior to using data affinity as the sole determinant in routing. Under some conditions, using data affinity alone can cause the queues to become unbounded in length. However, choosing the best threshold is somewhat problematical since it must be sensitive to system utilization. WLM, by contrast, tracks the actual response time delivered with no assumptions on transaction attributes.

Reference 4 includes the interesting observations that optimization for a single work request can negatively affect overall results and that load balancing becomes more important as the overall load increases.

The second alternative investigated in Reference 4 assumes that transactions fall into either a short or long duration, and routes the former using the base algorithm, but routes the latter based on data affinity. This approach makes the further assumption that which category a given work request lies in can be readily determined at run time. The idea is to take advantage of idle capacity when the risk of making a mistake is low, but to force data affinity when the cost is high. This algorithm does better than the base algorithm and the first alternative, but the improvement is sensitive to utilization and communication costs. This alternative would require some sort of external specification by the system administrator, unlike WLM, which makes no assumption that the duration of a work request can be determined upon its arrival.

The adaptive approach discussed in Reference 4 uses feedback to adjust for incorrect information. This approach enhances the base algorithm by tracking actual response time values and uses this value to adjust the estimated response time formula.

Reference 7 uses lock contention in a shared-data environment as a technique to determine how to route work requests. In particular, groups of transactions that access the same data are routed to a

given image to reduce the lock contention time. The basic objective is to ensure that each machine is kept at a "safe" utilization rate and to decide how to change the routing when any image is above its "safe" threshold. The algorithm depends on knowledge of factors such as:

- Threshold utilization
- CPU cost to process lock conflict when parties are on the same or different images
- Arrival rate of each transaction type and its CPU cost
- Affinity matrix representing the average number of times each transaction type waits for a lock held by each other type

WLM system model

The complexity of specifying low-level controls to tune system resources leads to a natural desire to offer the system administrator the capability to specify goals for work in the system in business terms, rather than using low-level controls. The operative principle is that the system should be responsible for implementing resource allocation algorithms that allow these goals to be met. WLM is unique in offering externals that capture business importance and goals and implements them on behalf of the system administrator.

Two primary concepts and facilities that WLM provides need to be introduced at this point. The first is the ability to partition the universe of work requests into mutually disjoint groups, called *service classes*. This partitioning is called *classification* and is based on the attributes of an individual work request, which might include the userid that submitted the request, related accounting information, the transaction program to be invoked or the job to be submitted, the work environment or subsystem to which the request was directed, and so forth. Installations are able to specify which service class is associated with each work request by specifying the value for one or more attributes and the corresponding service class. Defaults and other techniques may be used to group work requests into each service class.

Each service class represents work requests with identical business performance objectives. To address the fundamental problem that the resource demands of most work requests are unknown at the outset and can vary depending on parameters that may be known only at execution time, there is a need to allow the business objectives to change based on the resource demands of the work request. This is quite different from the requirement in other implementations that the resource demands be known in advance.

A service class is comprised of a sequence of *periods*, with a value defined by the installation to express how long a work request is considered to belong to each period. This "duration" is a measured amount of service consumed that incorporates time spent actually running instructions on a processor along with other components of service defined by the installation. Each work request starts in period 1 and is managed according to the first period goal (to be described in the next few paragraphs) until enough service is consumed to exceed the first period "duration." The work request is then moved to the second period and managed according to the second period goal, and so forth.

Each period has an associated *goal* and an associated *importance*, as alluded to above. Note that the durations may be assigned different values for distinct service classes, even when comparing the same period. In the same way, the goals for a given period in different service classes may be distinct. An installation may specify explicitly three major goal types for work requests. Certain activities associated with system work may be managed implicitly and are accorded special treatment and do not require installation specification. The goal types provided by WLM are *response time, discretionary*, and *velocity*. These types of goals are now described in turn.

Response time goals indicate a desire for internal elapsed time to be, at most, a certain value. "Internal" refers to the fact that the time is measured from the point where the work request is recognized by the system to the point where the work request is considered complete. Note that elapsed time refers to wall-clock time and, hence, includes delays when programs are not running on behalf of the work request. Use of wall-clock time is desirable since it reflects the impact on a user awaiting completion of the work request. The precise definition of when the clock starts or stops ticking to capture the elapsed time is documented for each particular environment and so is not elaborated in this paper.

The second goal type, discretionary, indicates that there is no business requirement for the work to complete within a certain predetermined elapsed time, and the system should use its discretion in giving resources to such work when it is ready to run. In an

unconstrained environment, discretionary work will use available resources. In a constrained environment, discretionary work may be denied resources in favor of work requests with other goal types. Optional controls not described in this paper allow the installation to ensure that discretionary work makes progress in a constrained environment.

The third goal type is velocity. Work requests that are not considered discretionary and do not have a set response time objective nevertheless may need further control to reflect the degree of delay that is tolerable once the work request becomes ready to run. Such work requests may be long-running (possibly "never-ending") and want to run periodically or intermittently, during which time the work request needs access to resources. Velocity goals address this category of work requests.

WLM is designed for a data-sharing environment.

A final concept associated with periods, which was mentioned above, is that of *importance*. Importance is merely a relative ranking of work and is only a factor in constrained environments where the algorithms must make choices as to whose goals will be attended to first when system resources are reallocated. The algorithms attend to the goals of work at the highest importance before attending to those at lower importance levels.

The concept of period was introduced to demonstrate a fundamental behavior of WLM of work that addresses the variability of resource demands. WLM does not require the system administrator to know these demands in advance. Goals are allowed to change based on their cost. The term "period" is not used subsequently in order to avoid certain technical discussions and difficulties that are not central to the theme of this paper. The more general concept of "service class" will be used in the remainder of the paper. For a more complete description of WLM externals, please refer to Reference 8.

The WLM philosophy for resource adjustment is described in some detail in subsequent sections, but it is essentially a receiver-donor loop with respect to adjusting resources. The fundamental principle on which its success is based is that the system need not determine the optimal change at any given point. It is sufficient that the system makes an improvement when adjustments are made. This principle allows WLM to avoid the trap of over-analysis where system overhead may balloon in search of optimal solutions. By working only on a single problem at a time, the algorithms leave intact resource allocations that are working well.

With the description of how WLM addresses resource controls completed, the second major consideration is to describe what requirements and assumptions WLM makes in how the images of the parallel environment are organized. The section on related work described the two major approaches as clustering vs sharing.

WLM assumes that each image is potentially capable of running any application. Any configuration requirements are the responsibility of the installation. WLM requires no intervention to reconfigure the images based on workload so as to fully utilize capacity.

WLM is designed for a data-sharing environment. Specific resource requirements are not currently incorporated into WLM, e.g., configurations that are asymmetric with respect to devices, vector, or cryptographic facilities, etc. This asymmetry is currently assumed to be handled by subsystems or dynamically managed by operating system or subsystem cooperation. For example, certain routing techniques described in the section on balancing work across a parallel environment can be used to group servers that have identical data and facility access capabilities. These routing techniques include generic resource and sysplex routing and allow the installation to group like servers without WLM awareness of what their common capabilities might be.

The third consideration for WLM to address is the question of how to route work requests among the images of the parallel environment.

Reference 9 describes a number of approaches for work balancing, among which WLM can be described as an adaptive model. Static models are not sufficiently robust for commercial environments, given the expected variability in arrival rate, resource re-

quirements, and so forth. The WLM structure possesses several desirable attributes described in the paper. First, it is important to ensure that the overhead associated with keeping the necessary data and the related calculations is low so as to avoid losing all advantage to extra system overhead. Second, the algorithms are not overly sensitive to inaccuracies in the data used to drive it. Third, simple approaches to load balancing prevail over complex algorithms. Finally, WLM will not move a work request that has already begun execution, since this is too expensive.

As will be discussed in subsequent sections, WLM also uses feedback to correct its view of how well each server is performing against actual business goals when deciding whether each server is a proper choice. However, WLM does not require nor use knowledge of data affinity in making its decisions. This is important for situations where this knowledge is unavailable or where the same cost is associated with accessing data from any candidate image as in a data-sharing environment. As noted in Reference 4, staying current on data affinity in the face of changing applications and usage patterns can make accuracy of data affinity assumptions problematical (and costly).

The WLM philosophy is to use actual measured results, which incorporate delays in all categories, and other indicators, without attempting to determine specific delays that cannot be directly controlled. Unlike Reference 7, which focuses on lock contention, WLM does not assume that data affinity can be determined on the basis of the attributes of an arriving work request. Of course, lock contention is not the only delay that must be considered in routing work requests.

The general philosophy adopted by WLM for balancing work across parallel systems is to place work where it has the "best" chance of meeting its goals, whatever they may be. This approach is superior to trying to fill up one machine prior to going to the next. It also addresses the problem of how to maximize use of resources across a parallel environment, especially where there are diverse machine sizes—the problem of configurational heterogeneity discussed in Reference 1.

The WLM design philosophy for routing consists of independent cooperating images with shared state data and uses a "push" model. A push model is one in which work requests are directed (pushed) to a given image for processing and is in contrast to a "pull" model wherein each image requests work explicitly. Unlike the approach described in Reference 9, WLM does not need to probe potential target images to see whether they are capable of absorbing new work as the shared state data are sufficient to make this determination. Note that in an OS/390* (formerly known as MVS) operating system environment, WLM can easily manage systems that are running at 90+ percent of capacity, whereas Reference 9 describes a model that works well in a range no higher than 70 to 80 percent of capacity. The approach of WLM is intended for dozens to hundreds of hosts, with overhead measured to be containable within 0.5 percent for several systems.

A number of benefits surface from the WLM philosophy of goal-oriented performance management. The most obvious of these benefits is the *simplification in defining performance objectives* and initialization states to the system. The system administrator is able to specify business objectives directly to the system in business terms. These objectives reflect both goals and business importance and apply to the entire parallel environment controlled by the business policy. It is still the responsibility of the system administrator to ensure that each service class contains work with similar goals, business importance, and resource requirements to acquire the maximum benefit from WLM. Placing work with similar goals but diverse resource requirements into the same service class will limit the ability of WLM to make effective resource trade-offs, to correctly project resource needs, and to project the effects of resource adjustments.

First, the system administrator does not have to understand low-level technical controls. There is no fiddling with dispatch controls. The system administrator does not have to understand trade-offs for setting dispatch priorities when a machine has a single very fast processing engine vs a single slower engine vs multiple slower engines vs multiple very fast engines. The system administrator does not have to individually set storage isolation targets (amount of processor storage that should be protected or restricted for a given address space), tune for the worst case, and then worry that the working set changes according to goal. The WLM algorithms will monitor and set the appropriate values automatically on behalf of the system administrator. *Effective use of capacity* is assured by the management algorithms.

The history of performance tuning has given rise to a number of heuristics to address different performance problem areas. Unfortunately, these "rules

of thumb" are often wrong. For example, paging may be tolerated so long as goals are being met. In the past, the system administrator might set some control value, hear that users are unsatisfied, and then have to retune, all the time having to balance the needs of conflicting workloads. Performance tuning with WLM does not require that the system administrator readjust resources, a process that is iterative and expensive. Effective use of capacity is assured by the management algorithms.

Next, the system administrator does not have to worry about the placement of work to the best image and best server within the parallel environment. There is no requirement to define the resource requirements of work requests to the system. Effective use of capacity is assured by the management algorithms.

Finally, the business policy defined to WLM handles mixed workloads, e.g., interactive, batch, transaction processing, data mining environments, and so forth. The system is responsible for resource management of work in execution and for the management of delays and their impact on attaining goals. There is no need to partition the images or nodes of the parallel environment for each separate workload. The system administrator does not have to specify the resource demands of work in advance. Effective use of capacity is assured by the management algorithms.

The second major benefit of the WLM philosophy is to allow granular *growth* to be transparent to the installation. Transparency simplifies the problem of scaling the environment as the workload grows. WLM supports dynamic changes in adding or removing images, subsystems, and applications, as well as variability in workload characteristics and resource demands. Reconfiguration need not affect performance objectives. If there is insufficient capacity to meet all goals, the business policy determines the relative business importance in meeting each goal. The addition of new applications need not cause revision of old objectives, with the attendant rebalancing of low-level controls. WLM dynamically adjusts to all these changes. WLM will also dynamically adjust to short-term changes, including spikes in demand.

The third major benefit is to support high *availability* objectives. This support includes rebalancing work when an image is removed and advising in the placement of restarting subsystem environments when their host system is removed. Change management is also simplified since a strategy of rippling hardware or software changes, or both, across images in the parallel environment while managing existing workload on remaining images allows for continuous operation 24 hours per day, seven days per week.

The fourth major benefit of the WLM philosophy is to require *no changes at the application level*. Support is provided by the operating system and major subsystem environments. It contrasts to an implementation such as Utopia,[1] where there are no kernel changes, but some major applications are assumed to change to be sensitive to routing considerations.

WLM algorithms for resource management

The Multisystem Goal-Driven Performance Controller (MGDPC) contains the resource management algorithms of WLM. The MGDPC is responsible for allocating computer system resources so that the customer's performance goals are met to the extent that the goals are achievable. The MGDPC must manage work across multiple systems. It must manage multiple types of work, from short transactions to processor-intensive batch transactions. It must manage client/server workloads, where resources must be allocated to servers to address the performance of the clients. It must manage workloads that vary, detecting performance problems and reallocating resources. It must manage multiple resources. And it must do all of this efficiently. The MGDPC must act like a very good systems programmer. The following subsections describe how it is done.

The code in the MGDPC combines the performance management approaches of an experienced systems programmer with analytic algorithms. The systems programmer in the MGDPC has the advantages of a wealth of data, analytic algorithms that run at machine speeds, the opportunity to make resource changes every ten seconds, and updated data and feedback on previous decisions every ten seconds. The MGDPC can be thought of as a data collection and analysis system, resource adjustment, and feedback loop extending across a set of interconnected, cooperating, independent computer systems.

The MGDPC collects performance data, measures the achievement of goals, selects the service classes that need their performance improved, selects bottleneck resources, selects donors of the resources, assesses the impact of making resource reallocations, and makes the reallocations if there is a net benefit to the changes. The MGDPC is invoked once every ten

seconds, referred to as a policy interval, performing detection and correction of actual or anticipated performance problems so as to make the operating system adaptive and self-tuning.

Independent and cooperating. The MGDPC is responsible for managing the performance of a workload that is distributed across a set of interconnected, cooperating, independent computer systems. These computer systems are said to be cooperating in the sense that each is exchanging operational measurement data with the other computer systems in the set. They are said to be independent in the sense that each is an entirely separate, wholly functional computer system whose resources are controlled by its own copy of the operating system. Each system operates independently and considers itself the local system. To each system, the remote systems are all the other systems being managed. Each system considers itself local and all other systems remote. The MGDPC is implemented as distributed intelligence. No system considers itself the master.

The primary objective of the MGDPC is to meet performance goals across all the systems being managed. This objective is met without any centralized control. Instead, each system receives performance data from all the other systems being managed and, based on its view of how the entire distributed workload is doing, makes resource allocation decisions to best meet sysplex (System/390* Parallel Sysplex* [10])-wide goals. A secondary objective of the MGDPC is to meet performance goals on its local system, in which case resource allocation decisions are made using local and remote data.

Each local MGDPC collects data on its local system, periodically broadcasts its view to the other systems in the sysplex, and implements mechanisms that can run independently on each system so that each system knows which class of work to help, by how much, and in what order, and knows the effects that resource reallocations on the local system will have on the sysplex performance of each class of work.

Each system's understanding of the sysplex effects of resource reallocations is the key to each system being able to independently make local resource trade-offs to achieve sysplex performance goals. Each system must also understand which portion of the problem it must solve so multiple systems do not all try to solve all parts of the problem at the same time.

Another feature of the MGDPC allows the systems to reallocate resources to help work that is doing poorly on the local system even though the work is doing well from a sysplex perspective. This local optimization is allowed as long as it does not adversely affect the relative sysplex performance of other classes of work. If an individual system determines that there is nothing it must do to assist work to achieve sysplex performance goals, it is free to work on local performance problems to the extent that sysplex goals are not adversely affected. It has enough data to project the effect of local resource reallocations on sysplex goals.

Fundamental concepts. In this subsection, the fundamental concepts of MGDPC operation are discussed.

Data histories. The MGDPC algorithms require efficient access to large quantities and varieties of performance data. Individual MGDPC algorithms need data summarized over different periods of time. Since individual algorithms also need different levels of statistical confidence in the data, they need to be able to look at different minimum numbers of data points. The use of the data determines how far back in time it is necessary to look or the minimum number of data points required to get a valid representation of a phenomena, or both. It is therefore important to maintain the number of data points represented in the performance data, and it is not always sufficient to merely keep a single summary value. Keeping all the individual observations of all the types of performance data in virtual storage, and searching and summarizing on demand, would consume far too much storage. Accessing the data from disk would require far too much time.

The MGDPC solved the problem with data histories. A data history is a mechanism to collect and analyze data over time. By using data histories the MGDPC can use data that have enough samples to be representative without using data so old that the data might be out of date. A data history contains n rows of data and a roll counter that determines when data should roll out of each row. Each row represents data from a range of time in history. Row 1 contains data from the most recent period only. Subsequent rows contain varying ranges of older data. Values for the number of rows have been found that have been proven to be effective for the OS/390 environment. The roll counter controls when to roll a row of data from one time range to another further back in time. The roll counter is incremented each policy inter-

val. Each row has associated with it a number that corresponds to the roll counter value specifying when the data in the row should be rolled into the next row. If the counter value of row m is 1, it means row m is rolled into row $m + 1$ every interval. If the counter of row m is 4, it means row m is rolled every fourth interval.

Data are added to the history as follows. New data are added to row 1. At the end of each policy interval the oldest row whose roll counter value evenly divides into the current roll counter value is found. The content of that row is added to the next numerically higher row. The content of all the numerically lower rows are moved up one row, leaving row 1 empty. When it is time to roll data out of the last row in the history, the data are discarded. To obtain data from a data history, the data from rows 1 through p are added together. The value of p is chosen such that the data used were gathered over a long enough interval with enough samples to be representative.

Given the ability to summarize data for varying ranges of time by simply including data from different rows of the history, data can be summarized for a minimum time or a minimum number of observations, or a combination of these criteria. The MGDPC uses the data history facility extensively. Histories are used for state samples, response time distributions, processor consumption, performance index calculations, service consumption per transaction, server topology determination, and other purposes.

Performance index. A fundamental problem with trying to meet performance goals and make trade-offs among different work with different goals is knowing how work is doing relative to its goals and relative to other work. The solution used by the MGDPC is the performance index. The calculation of the performance index for a class with a response time goal is:

$$\text{performance\_index} = \frac{\text{actual\_response\_time}}{\text{goal\_response\_time}}$$

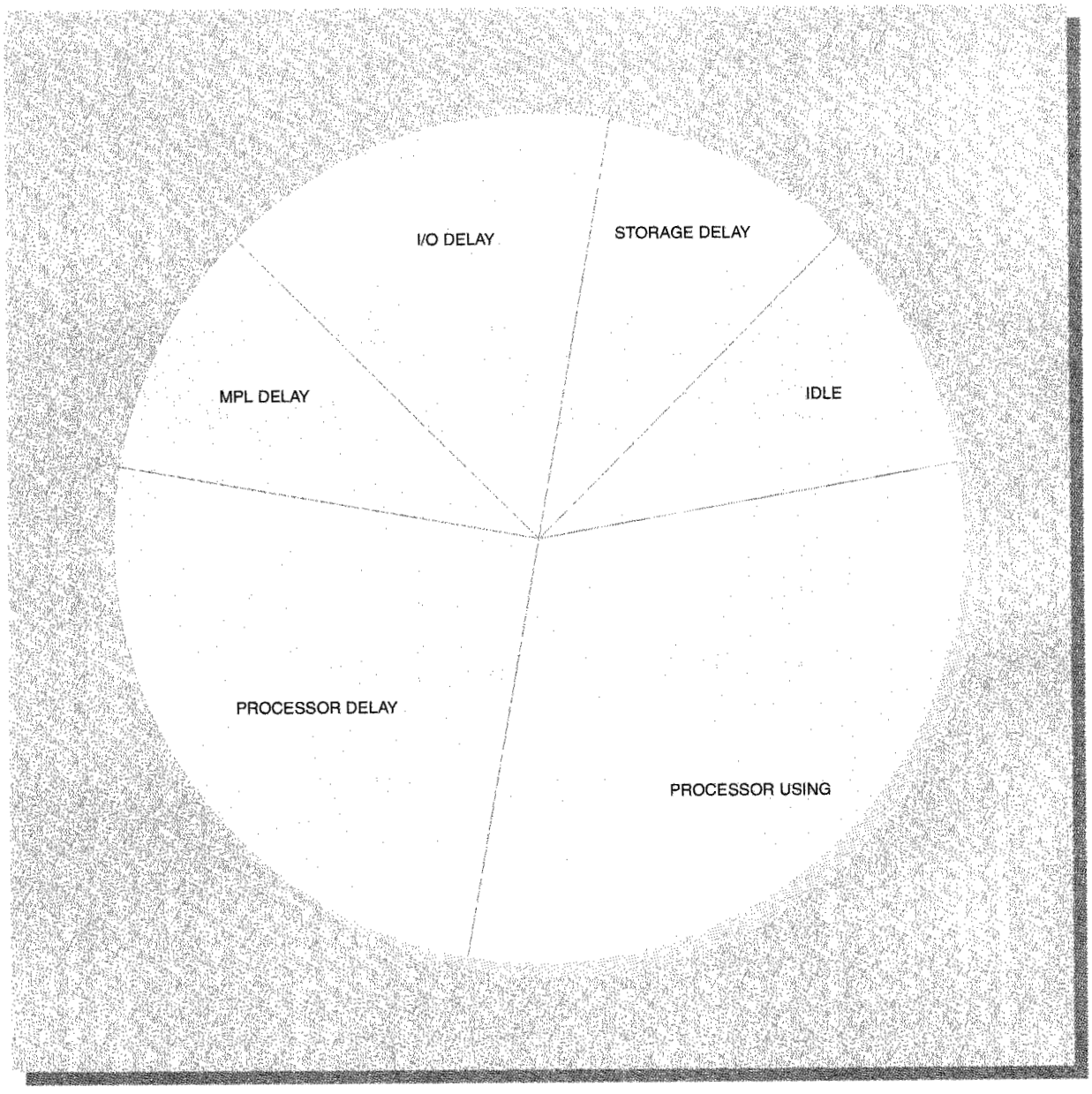
It is a calculated measure of how well work is meeting its defined performance goals. The performance index allows comparisons between work with different goals. A performance index of 1.0 indicates the class is exactly meeting its goal. A performance index greater than 1.0 indicates the class is performing worse than its goal, and a performance index less than 1.0 indicates the class is performing better than its goal.

New performance indexes are calculated for every policy interval. Performance indexes are calculated from enough recent completions to be representative of the results for the class. Both sysplex and local performance indexes are calculated for each class on each system. To operate independently, each system must have enough information to be able to calculate a performance index for each class. To provide this information, each system sends updated information to all the other systems every policy interval. The information is stored in two histories. Local information is stored in a local history, and data from the remote systems are stored in a history for data from remote systems.

A projected response time is calculated for each in-flight work unit. The projected response times for in-flight work are combined with data from the actual response completions to calculate the performance index. The local performance index represents the performance of work units associated with the class on the local system. The local performance index is calculated from data from the local response time history. The sysplex performance index represents the performance of work units associated with the class, across all the systems being managed. Each system independently combines the local and remote data histories to compute a sysplex performance index.

State sampling. The first action to be taken when trying to solve the performance problem of a service class is finding out what the problem is. The MGDPC must determine why the work is being delayed. Many delays can be measured quite precisely, but the cost is prohibitive. The MGDPC solved this problem with state sampling. Four times a second, the MGDPC samples every work unit in every system being managed. Four times was chosen as a value because it is frequent enough but not prohibitive in cost. From these samples, the MGDPC builds a picture of the work in each class. It learns where each class is spending its time. It learns how much each class is using each resource and how much each class is delayed waiting for each resource. The samples are aggregated for each policy interval, and from this picture of the work in each class, the MGDPC can determine what to do. The state sampling implemented by the MGDPC is very efficient, requiring not more than one percent of the processor time to accomplish its task. The cost of state sampling is by far the largest contributor to

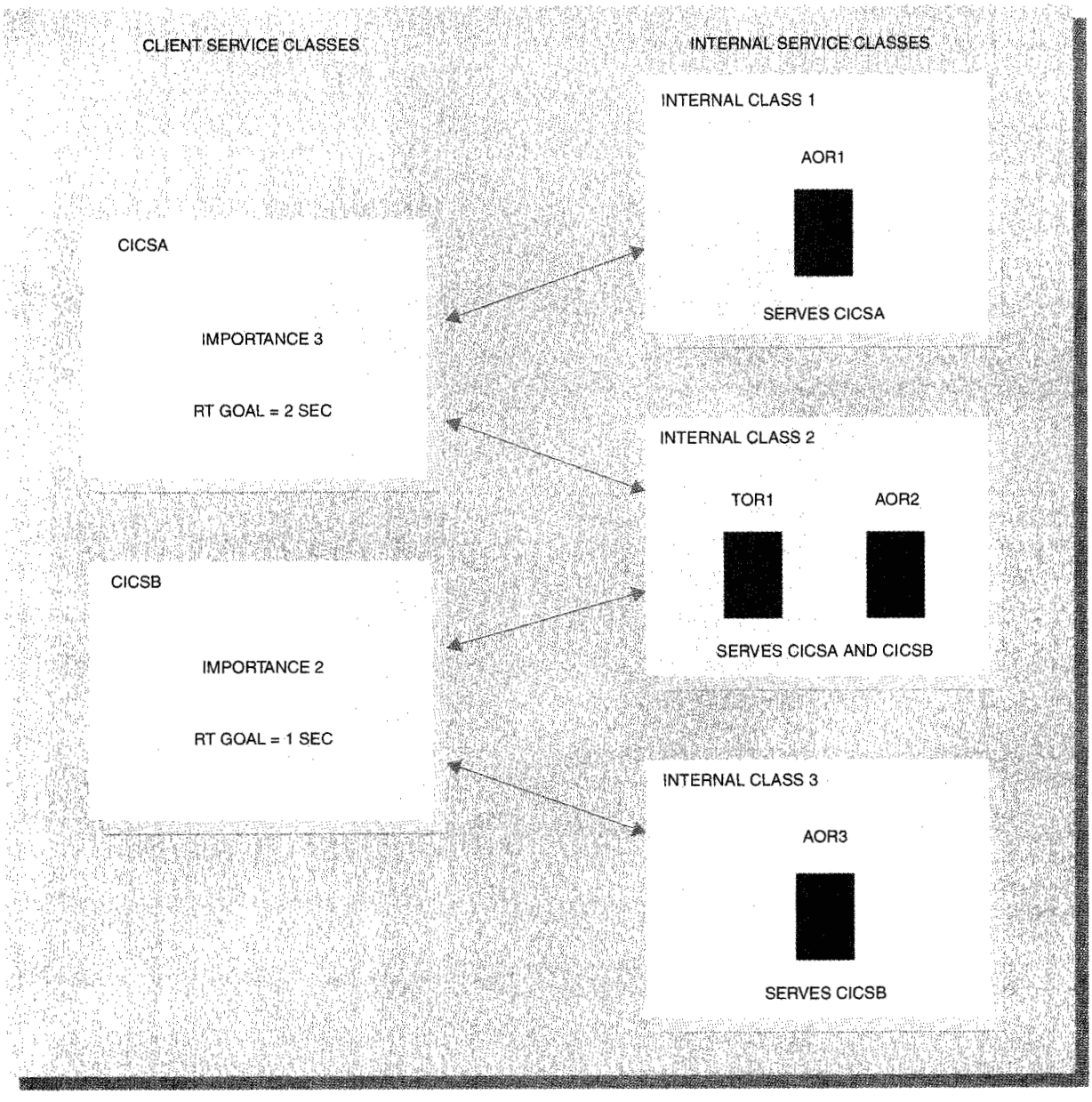
Figure 1 State sample types



system overhead among the various functions performed. Figure 1 shows the types of state samples.

Server topology. Client/server workloads introduce a further level of complexity into managing resources to meet performance goals. The client service classes[11] have the performance goals but are served by one or more server address spaces. The client service classes do not consume computer system resources. The resources are consumed by the server address spaces serving the client service classes. So computer system resources must be allocated to the server address spaces to meet the goals of the client service classes. The MGDPC must understand the

Figure 2 Client/server diagram



client/server relationships and must be able to project the effects on the client service classes of making resource adjustments to the server address spaces. The MGDPC must be able to project second-level effects.

The client service classes in the diagram of Figure 2 are labeled CICSA and CICSB (from Customer Information Control System, CICS*). Work requests classified to CICSA and CICSB receive service from several server address spaces. CICSA is served by server spaces TOR1, AOR1, and AOR2. CICSB is served by server spaces TOR1, AOR2, and AOR3. Achieving the goals of CICSA and CICSB requires that adequate computer system resources be allocated to the server address spaces—TOR1, AOR1, AOR2, and AOR3—

since resources cannot be directly attributed to or allocated to CICSA and CICSB.

The problem of learning the client/server relationships was solved by sampling. The problem of allocating computer system resources to server address spaces to meet the goals of the client service classes was solved by dynamically creating internally defined server service classes and assigning the server address spaces to them based on the client service classes they were observed serving. The problem of projecting second-level effects was solved using a proportional aggregate speed algorithm.

Four times a second, the MGDPC samples control blocks set by the server address spaces to detect which client service classes are being served. From these samples, the MGDPC learns which server address spaces serve which client service classes and in what proportion. The MGDPC reevaluates these client/server relationships once a minute so the topology built will reflect changing client/server relationships. Server address spaces are also moved among internal service classes once a minute to reflect any changes in the client/server topology.

For each distinct combination of client service classes observed being served by one or more servers, an internally defined server service class is dynamically created. In the example in Figure 2, these combinations are (CICSA), (CICSA, CICSB), and (CICSB). AOR1 serves only CICSA. TOR1 and AOR2 serve both CICSA and CICSB. AOR3 serves only CICSB. On the basis of these combinations, the MGDPC creates the corresponding internal server service classes 1, 2, and 3 and moves TOR1, AOR1, AOR2, and AOR3 to them for management. Internal classes are a mechanism for collecting data on and managing servers to meet the goals of clients. To meet the client service class goals of CICSA and CICSB, the server address spaces will be managed by managing server service classes 1, 2, and 3.

Computer system resources are allocated to these internal server service classes in order to meet the performance goals of the client service classes. The topology represents the client/server relationships and the proportion of time each server is serving each client. This learned information will adapt over time, because the relationship between clients and server address spaces is dynamic. The server topology samples are kept in a history. The history mechanisms slowly age the samples out so there are less likely to be abrupt changes based on short-term effects.

Proportional aggregate speed. In the client/server case, the MGDPC must improve the performance of the client service classes indirectly. The MGDPC must be able to assess the effect on a client service class, e.g., CICSA, from improving the performance of an internal service class, e.g., Internal Class 2. This improvement is proportional to the extent to which the client service class, e.g., CICSA, is served by the server spaces in the internal service class. To be able to project the effects on clients of the resources allocated to the servers, the concept of the proportional aggregate speed of a client class was introduced.

For a nonserved class, speed is defined as the classes' processor "using samples"[12] divided by all of the nonidle samples of the class, multiplied by 100, and results in this calculation:

$$\text{speed} = \frac{\text{processor\_using\_samples}}{\text{nonidle\_samples}} \times 100$$

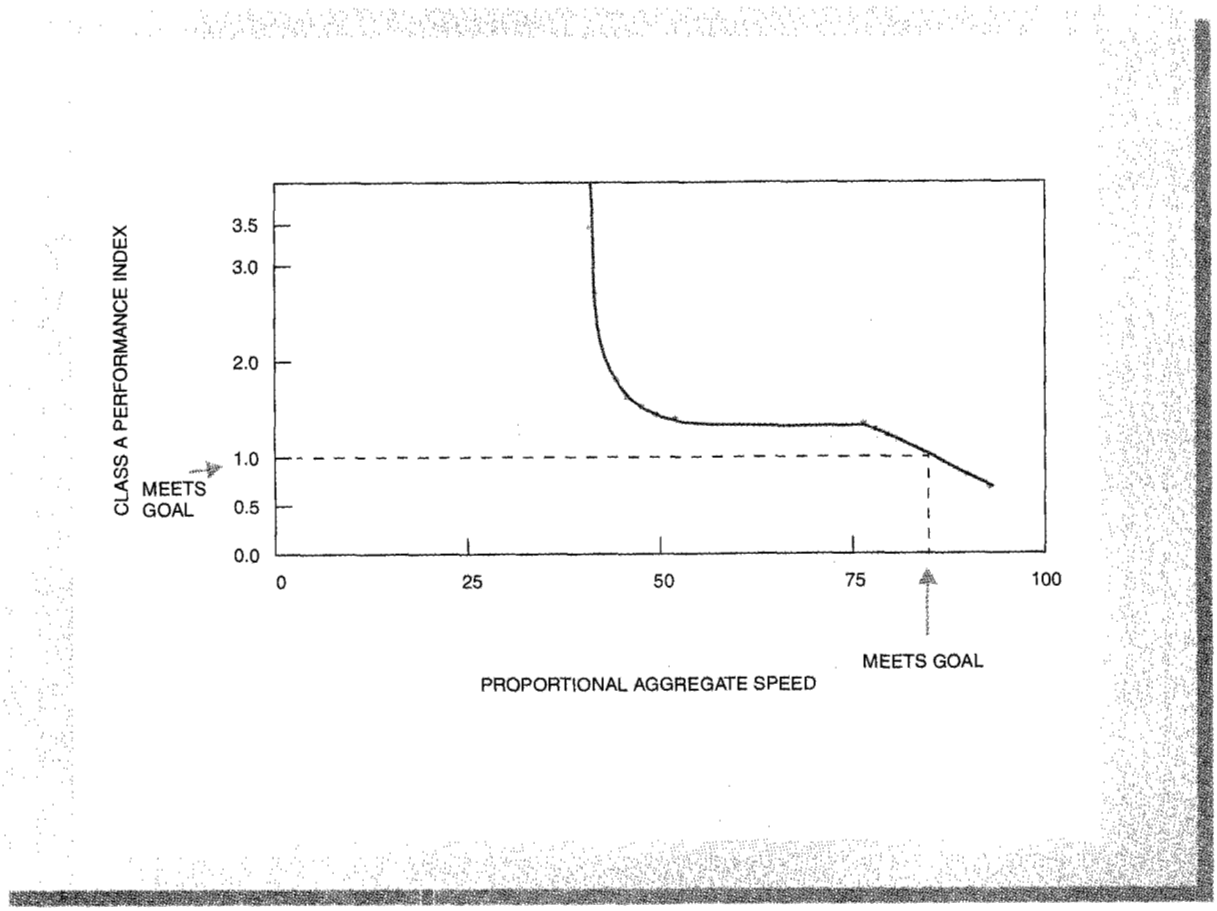
If the work units in the class were never delayed, the speed of the class would be 100.

The proportional aggregate speed of a client service class is the apportioned speed of all the internally defined server service classes serving it. The proportional aggregate speed for each client service class is determined by allocating all of the client's server's state samples to the client service class in proportion to the portion of time that each server service class was observed serving each client service class. The portion of time is determined from the client/server topology. The proportional aggregate speed of a client service class is calculated by dividing the total processor using samples apportioned to the client service class from all server service classes, divided by the total processor using samples plus all delay samples apportioned to the client service class from all server service classes. The calculation follows:

$$\frac{\sum_{\text{servers}} \text{processor using samples apportioned to class A}}{\sum_{\text{servers}} \text{using and delay samples apportioned to class A}}$$

For each client service class, the client's performance index is plotted versus the proportional aggregate speed of the client class. This plot, shown in Figure 3, is then used to determine the effect, i.e., the performance index delta, on the client of changing the

Figure 3 Class A proportional aggregate speed plot



allocation of system resources to server address spaces.

Performance index delta. Just as the performance index is the measure of how well a class is doing with respect to its goals, the performance index delta is the common unit of measure for the relative value of making resource reallocations. The performance index delta is always calculated from delay sample deltas. Each individual resource fix algorithm uses algorithms unique to the resource to determine the delay sample delta that will result from a resource reallocation. Then the delay sample deltas are used to calculate the performance index deltas that are used to assess the relative value of the resource reallocation.

For nonserved classes, performance index deltas are calculated as shown below. The calculation is a three-step process. First, the projected response time delta is calculated. It is the actual response time multiplied by the proportion of the total nonidle samples represented by the sample delta. If the total samples were 100, and the delay samples projected to be eliminated were 20, the response time would be projected to be reduced by 20 percent. Then the delta to the local performance index is calculated from the projected response time delta. Finally, the sysplex performance index delta is calculated from the fraction of total observations in which the class was observed on the local system. Note that these equations apply to both receivers and donors. For a receiver, the delay sample delta is negative, so the performance in-

dex is projected to be lower, which is an improvement. For a donor, the delay sample delta is positive, reflecting additional delay and an increased performance index.

$$\text{proj\_response\_time\_delta} = \frac{\text{delay\_sample\_delta}}{\text{nonidle\_samples}} \times \text{actual\_response\_time} \quad (1)$$

$$\text{local\_proj\_performance\_index\_delta} = \frac{\text{proj\_response\_time\_delta}}{\text{goal}} \quad (2)$$

$$\text{sysplex\_proj\_performance\_index\_delta} = \frac{\text{local\_observations}}{\text{sysplex\_observations}} \times \text{local\_proj\_performance\_index\_delta} \quad (3)$$

For client/server classes, the performance index delta is determined from the client's proportional aggregate speed plot. To read the projected performance index from the plot, a projected proportional aggregate speed must be calculated. The calculation starts with delay sample deltas calculated by the individual fix algorithms. Projected delay sample deltas are calculated for each server that serves the client class. Then the sample deltas are apportioned to the client class based on the server topology. The server topology represents the client/server relationships and the proportion of time each server is serving each client class. After the sample deltas of the server are apportioned to the client, the projected proportional aggregate speed is calculated for the client class. Then, the projected performance index is read from the client's proportional aggregate speed plot. The performance index delta of the client class is the difference between the projected performance index of the client class and the current actual performance index of the client class. Proportional aggregate speed plots contain sysplex data, so no local-to-sysplex performance index delta conversion is required.

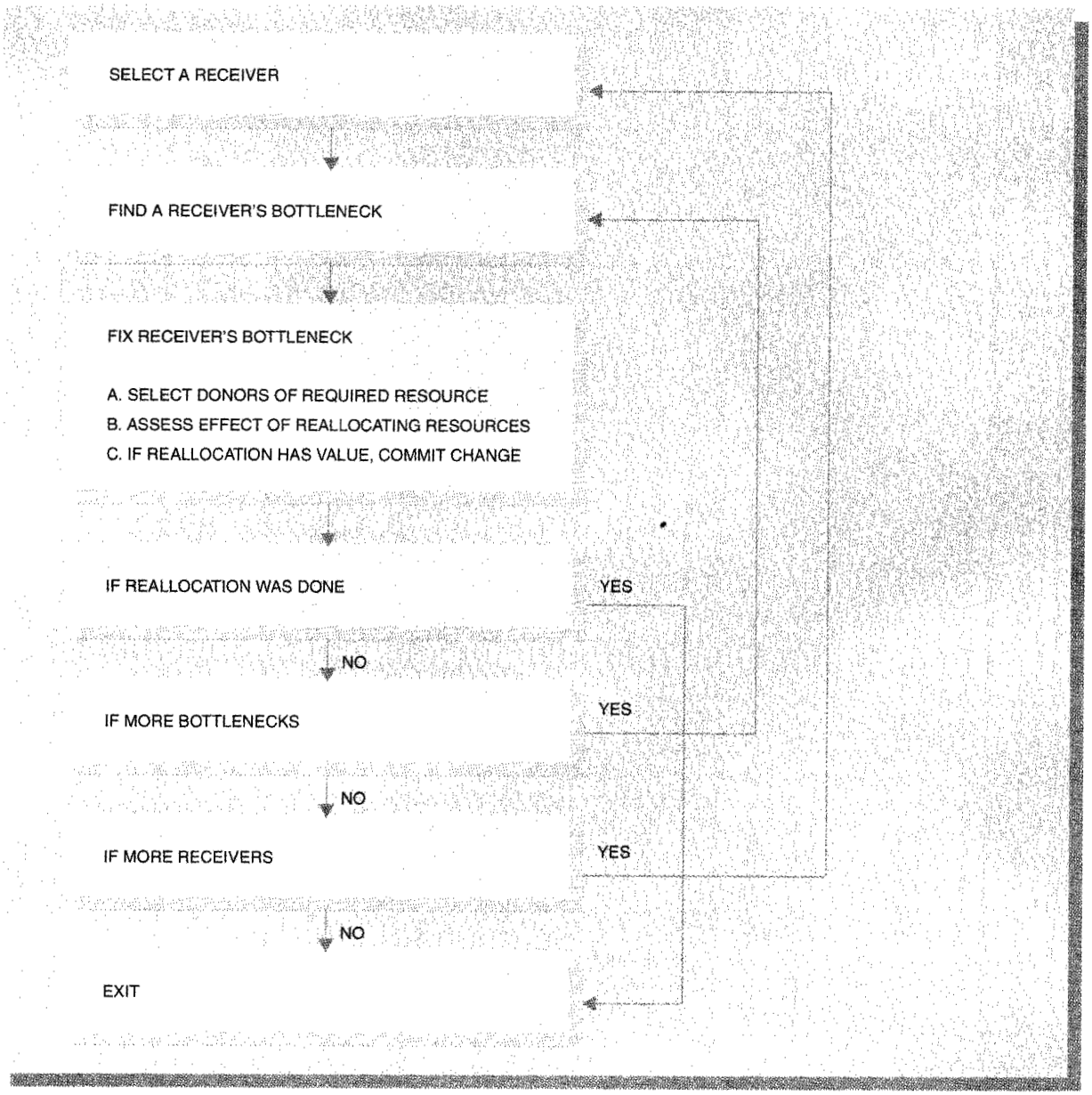
Policy adjustment framework. The policy adjustment algorithm is invoked periodically to assess reallocating system resources to better meet performance goals. The policy adjustment algorithm is invoked every ten seconds. Ten seconds was chosen as a value sufficiently small to be responsive to changing system conditions and external user perceptions, but sufficiently large to allow enough samples to be acquired on which to base new resource allocations. This period of ten seconds is referred to as a policy interval. The effects of the resource reallocations made during one policy interval are observed in subsequent policy intervals and function as a feedback loop for continuous adaptive policy adjustment.

The resource readjustment actions taken are incremental, having the advantage of leaving resource allocations alone except when changes are needed to meet performance goals. Since the MGDPC is invoked every ten seconds, there are ample opportunities for it to make sufficient changes to address any problems and to obtain feedback before making further changes. Some of the most "human" behavior observed in the MGDPC is its inclination to jump in immediately to help whenever it can but also to recognize when its help is not needed.

The MGDPC helps by searching for the one set of actions most beneficial to the service class most in need of help. The *select receiver* algorithm is used to select the receiver service class most in need of help and to select alternative receivers if needed. The *find bottleneck* algorithm is used to find the resources causing the receiver delay. The *select donor* algorithm selects potential donor service classes to donate bottleneck resources to the receiver. The *net value* algorithm determines whether there is net value to the donation. The *receiver value* algorithm determines whether there is sufficient value to the receiver to make the donation worth doing. The *fix delay* algorithms are unique for each resource and are used to assess changes and calculate the value of changes in common value units (performance index deltas) to be used by net value and receiver value algorithms. These algorithms are invoked in a loop, referred to as the *policy adjustment loop*, illustrated in Figure 4, until one receiver service class is helped or all service classes have been assessed, and there is no way or no need to help. All of these algorithms are discussed further in the following subsections.

The policy adjustment loop selects a class to help (select receiver), determines the resource causing the class the largest delay (find bottleneck), assesses reallocating resources from one or more donor classes to the receiver (fix delay, select donor, net value, and receiver value), and makes the changes if there is value to the aggregate attainment of goals. If there is insufficient net value or receiver value with one set of donors, other sets of donors will be assessed.

Figure 4 Policy adjustment algorithm loop



If there is insufficient net value or receiver value with any combination of donors for a given resource and receiver, the resource causing the receiver the next largest delay will be determined and donors of that resource assessed. If there is no combination of donors of any resource for a given receiver, the next most deserving receiver will be selected, and all resources and donors assessed for that receiver until all possible receivers have been assessed or until a receiver is helped. When a receiver has been helped, the MGDPC exits to await feedback on the changes during the next policy interval.

The policy adjustment loop and the select receiver, find bottleneck, select donor, net value, and receiver value algorithms are all common for all the resources. The fix delay algorithm is unique for each resource. This loop is a very powerful framework for performance management. A fix algorithm for any resource can fit into this framework. The only requirements are that a delay that indicated a lack of the resource can be sampled, a control variable controlling access to the resource can be defined, and a relationship can be found between the control variable and the resulting delay samples. These concepts, as they apply to dispatch priority, I/O priority, storage allocation, and MPL[13] slots, are described in later subsections.

Assemble performance data. At the beginning of each policy interval, performance data that have been collected asynchronously by state sampling and other processes are assembled into efficiently accessible data structures to prepare for running the adjustment algorithms. Performance indexes are calculated, data received from other systems are assembled into histories, points are added to plots, sample sets are built, and the server topology is updated. It is similar to what a system programmer would do in preparation for tuning a system. The difference is that the MGDPC does data assembly at machine speed.

Select receiver. The first decision the policy adjustment algorithm must make is to decide which class to help. The MGDPC makes incremental improvements every ten seconds. It attempts to find one receiver to help each policy interval and looks for the most deserving receiver each time. Making incremental changes ensures that there is a solid base of feedback data to use in the algorithms during each policy interval. Potential receivers are selected based on importance, sysplex and local performance index, and the likelihood of the MGDPC being able to help the receiver. Classes that are missing sysplex goals are selected before classes that are meeting sysplex goals but missing goals on the local system. Classes missing goals are selected in order of importance. Classes meeting goals are selected in sysplex performance index order and then in local performance index order. Because the worst-off classes are selected first, it is more likely that a resource reallocation with significant value will be found.

The policy adjustment algorithm also remembers whether it has tried unsuccessfully to help a receiver in a recent interval. If it did, the select receiver algorithm skips over assessing the receiver. This is an optimization which saves the cycles that would be used to again come to the conclusion that the receiver could not be helped. Select receiver also knows when to leave resource allocations alone. It only selects classes that have a current performance index above 0.9. Classes that are meeting goals but have a performance index above 0.9 are close enough to going over 1.0 to merit some attention if their performance can be improved without harming other work. However, classes with a current performance index of 0.9 or lower are easily meeting their goals and do not need help. The select receiver algorithm has the intelligence to know when to quit.

Find bottleneck. Once the receiver class has been selected, the next step is to select which resource delays to address. For nonserved classes, the selection of the next bottleneck to address is made by selecting the delay type with the largest number of delay samples that has not already been selected for this receiver during the current policy interval. If fixing that delay does not provide sufficient receiver or net value, the next largest delay is assessed and so on until all delays have been considered.

In the client/server case, both a bottleneck resource and the associated bottleneck server must be selected. The selection of which bottleneck to address is made by selecting the server-delay combination with the largest number of apportioned delay samples that has not already been selected during the policy interval. The server samples are apportioned to each client class on the basis of the server topology described previously. The delay type having the largest number of samples apportioned to the receiver class is selected as the resource bottleneck delay type to be addressed on behalf of the receiver class. The server that experienced the bottleneck delay is selected as the bottleneck server.

In either the nonserver case or the client/server case, on each invocation, the delay with the next largest number of delay samples is selected to be assessed. No minimum number of samples is required for a delay to be assessed for fixing. Any defined minimum would by its nature be arbitrary and might eliminate a valuable change from consideration. The MGDPC handles the problem of making insignificant changes by requiring sufficient receiver value for a change. If too few samples would be eliminated to make a significant improvement, the change for that delay would fail the receiver value algorithms. But at that

point the decision would have been well thought out, not arbitrary.

Generic delay fix. There is a specific fix algorithm for each delay addressed by the MGDPC. The function of each fix algorithm is to improve the performance of the receiver class or determine that there is not sufficient value to make a change. Improving performance is done by changing a control variable specific to the delay being addressed. To determine value, the fix algorithm must be able to project the performance index delta that results from changing the control variable. A fix algorithm specific to the delay to be addressed is invoked when that delay is selected by the find bottleneck algorithm.

Each fix algorithm is responsible for selecting potential donors of the resource, projecting the effect on attainment of performance goals if the donor or donors donated to the receiver, accepting or rejecting changes, selecting alternate donors, and reallocating the resources if any reallocation is found that has net value. In all cases, the individual resource fix algorithm projects delay sample deltas and uses them to project performance index deltas for the receiver and donor or donors. The projected performance index deltas are then used to determine whether the resource reallocation has net value. The details of these calculations are specific to individual resources and are described later.

Select donor. The purpose of the select donor algorithm is to choose the most eligible class that will donate the required resource to the receiver from the set of classes owning that resource. Donors are selected in an order that is generally the reverse of the order used to select receivers. However, the donor order is dynamic even within a policy interval. Multiple donors may be needed to provide enough of a resource donation to reach sufficient receiver value. As each tentative donation is evaluated and accumulated, the resulting performance index changes are calculated and factored back into the donor order. The dynamically changing list feature is important, especially when finding storage donors, where donation can take many forms.

Additional constraints on the select donor algorithm require that the donor own the resource needed by the receiver. For example, a dispatch priority donor must be running at a dispatch priority that is at least equal to the dispatch priority of the receiver. In the case of storage, the donor can hold the resource in any form. For example, if the receiver needs MPL slots, the donor does not have to donate MPL slots. What the receiver actually needs is storage for MPL slots. The donor's storage can be in the form of a protective processor storage target or in the form of MPL slots. If the resource required to help the receiver is increased I/O priority, the I/O donor must be in the same I/O cluster[14] as the I/O receiver. This requirement must be met for the donation to have any effect on the receiver.

In addition, the select donor algorithm will not select a class as a donor of a resource if it was selected as a receiver for the same resource in the same policy interval. This is an example of including the experience of a system performance analyst in the code.

Net value. The net value algorithm keeps the MGDPC from making bad resource reallocations. The performance index value for a class is the measure of how well that class is meeting its specified goal. The measure of the value of a contemplated resource reallocation to the receiver is the projected change in the performance index of the receiver that occurs as a result of the contemplated resource reallocation. Similarly, the measure of the net value of a contemplated resource reallocation is the improvement in the performance index of the receiver relative to the degradation of the performance index of the donor.

The net value algorithm uses the projected performance index deltas for the receiver and donor to calculate whether there is net value to the contemplated donation from the donor to the receiver. Net value takes the sysplex and local performance indexes into consideration as well as the importance of the receiver and donor or donors. All donors are checked. A receiver will only be improved by reallocating resource from a specific donor if a net positive value to the resource reallocation is projected. If using a donor to improve a receiver is projected to result in more harm to the donor than improvement to the receiver relative to the goals and importance, the resource reallocation is not done. If the result will yield more improvement for the receiver than harm to the donor relative to the goals, the resource reallocation is done.

Receiver value. The receiver value algorithm is a key feature that keeps the MGDPC from making resource reallocations that are either too small or too drastic. A receiver will only be helped when sufficient receiver value is projected. The receiver value criteria are a minimum performance index improvement or the

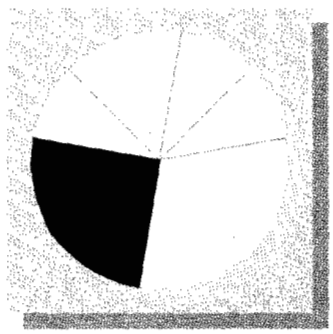
elimination of a minimum number of delay samples. These criteria are designed to reject very small improvements. The reason for rejecting actions having too little receiver value is to avoid making changes that yield only marginal improvements. Marginal changes are not made, and the MGDPC goes on to select and assess another bottleneck for the current receiver or to select a new receiver.

The receiver value criteria also perform the function of indicating to the "individual resource delay fix algorithm" at what point it has given the receiver enough help. These criteria keep one system in a sysplex from trying to solve all of the performance problems of a class when the class is running on more than one system. The criteria also keep multiple systems in the sysplex from trying to solve all parts of the problem simultaneously and running the risk of making too much of a correction. None of the systems require explicit communication or coordination to know how much of the problem is theirs to fix.

Send data. At the end of each policy interval on each system, the MGDPC sends data to all the other systems in the set of independent cooperating systems being managed. Performance data and control data are sent. This action is of key importance to the distributed intelligence of the MGDPC.

The MGDPC on each system maintains a history for each type of performance data received. The histories cover enough intervals of time such that late or out-of-order data do not require special handling or error processing. The late data just roll into the history whenever the data arrive. If a system fails, and its data stop arriving, it simply stops being included in the history, and stops being considered in decisions. The data from the failing system will gracefully age out of the history without the other systems having to be specifically notified that a system went down. It eliminates the need for special-case and error-handling mechanisms and abrupt changes in resource allocation policies on individual systems. The use of histories to manage the remote performance data allows the systems being managed to operate independently.

Control data are also sent to remote systems at the end of each policy interval. An example is sending the fact that an I/O priority change was made. I/O priority changes require a relatively longer time to provide feedback than other changes such as dispatch priority. Since these changes take longer to provide feedback, they are made less frequently. To accomplish this longer interval between changes, each system must know whenever another system made such a change.



Processor delay fix. This subsection describes how performance is improved by reducing the delay the receiver experiences waiting to run on the processor. The controlled variable in this case is the dispatch priority.

Theory. The processor delay experienced by the receiver is a function of the processor time available to the receiver. Processor time available to the receiver is a function of the processor demand from work running at higher dispatch priorities than the receiver and the processor demand from work running at the same dispatch priority as the receiver. Processor delay is also a function of both the receiver's mean-time-to-wait and the receiver's mean-time-to-wait compared with the mean-time-to-wait of the other work at the same dispatch priority as the receiver.

For the processor delay fix algorithm to fit with the resource adjustment framework discussed previously, the processor delay fix algorithm has to be able to project the processor delay sample deltas that would result from dispatch priority changes. Multiple steps and relationships are required to do these projections. In working backward from sample deltas, projected processor sample deltas are a function of the actual processor delay samples of an individual class and the actual wait-to-using ratio and projected wait-to-using ratio. The projected wait-to-using ratio of an individual class is a function of both the actual mean-time-to-wait of the class and the actual mean-time-to-wait of the class compared to the actual mean-time-to-wait of the other work at the same dispatch priority. The projected wait-to-using ratio at a priority is a function of the processor demand of work running at higher dispatch priorities and the processor demand of work running at the same dispatch priority.

Actual delay samples, actual wait-to-using ratios, and actual mean-time-to-wait values are measurable. That leaves the problem of defining algorithms to project processor demand, wait-to-using ratios, and delay samples.

Maximum processor demand. The first problem with projecting the effects of dispatch priority changes is that the inherent processor demand of the work units in a class cannot be measured directly. If a class consumes x amount of processor service when it runs at dispatch priority a, it cannot be assumed that it will still consume the same amount of service when it runs at a higher or lower priority or with a more or less competing demand. The MGDPC required an algorithm to project the processor consumption of a class at any dispatch priority. The solution was to define the concept of maximum processor demand.

Maximum demand is defined as the theoretical maximum percentage of total processor time that work units in a class can consume if the demand has no processor delay. Its calculation follows:

$$\text{maximum\_demand\_percentage} = \frac{\text{number\_of\_work\_units} \times \text{processor\_using\_samples} \times 100}{\text{total\_samples} - \text{processor\_delay\_samples}}$$

Maximum demand is calculated for each class and accumulated for all the classes at each priority.

Wait-to-using ratio. The next step in projecting processor sample deltas is to project the wait-to-using ratio that will be experienced by the classes at each priority given that one or more classes have tentatively changed priority. The aggregate projected wait-to-using ratio at a priority is a function of the processor demand of work running at higher dispatch priorities and the processor demand of work running at the same dispatch priority. The data used in the algorithm are the maximum demand of all the work running at each priority and the processor-using and delay samples accumulated by the classes at each priority. The current values of aggregate wait-to-using and aggregate maximum demand at each priority are used to determine the current functions relating wait-to-using to maximum demand. For each policy interval, these functions are derived dynamically to fit the current environment. Then the dynamically derived functions are used to project the aggregate wait-to-using ratios expected to be experienced by the work at each priority after one or more classes and their demands are moved from one priority to another.

Individual wait-to-using ratio. Next the individual wait-to-using ratio for each class is calculated as shown below. The aggregate projected wait-to-using ratio at a priority was calculated above. The individual mean-time-to-wait was measured. Individual mean-time-to-wait is a function of the work units in the class and does not vary with priority. Service-weighted mean-time-to-wait is the sum of the products of individual mean-time-to-wait and individual processor service consumption with the sum divided by the total processor service consumption at the priority.

$$\text{proj\_ind\_wait\_to\_using\_ratio} = \frac{\text{service\_weighted\_average\_mean\_time\_to\_wait}}{\text{individual\_mean\_time\_to\_wait}} \times \text{proj\_wait\_to\_using}$$

Processor delay sample delta. Finally, projected processor delay sample deltas are calculated as shown below. The projected individual wait-to-using ratio was calculated above. The actual wait-to-using ratio was measured, and the actual processor delay sample value was measured.

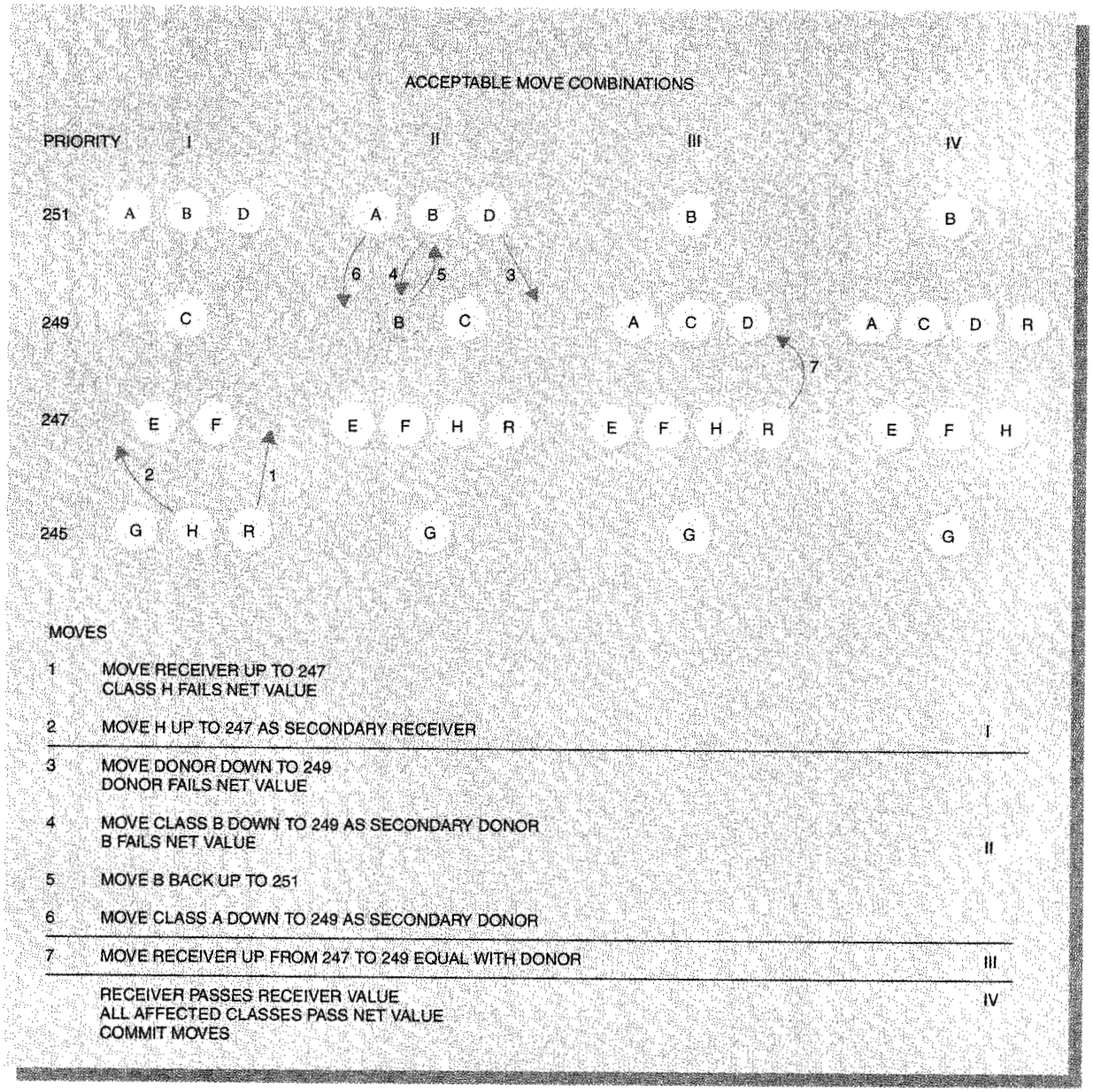
$$\text{proj\_delay\_samples} = \frac{\text{proj\_ind\_wait\_to\_using\_ratio}}{\text{actual\_ind\_wait\_to\_using\_ratio}} \times \text{actual\_processor\_delay\_samples}$$

The projected processor delay samples are equal to the actual observed processor delay samples, multiplied by the projected wait-to-using ratio, divided by the actual wait-to-using ratio. The delay sample delta is equal to the projected delay samples, minus the actual samples.

Operation. A state machine was developed to select and examine combinations of receivers and donors in order to identify and assess combinations of dispatching priority changes. The state machine is the mechanism used to determine whether the next priority move should be to move the receiver up, to move the donor down, to checkpoint interim changes, to commit final changes, or to select another donor. Figure 5 shows an example of a state machine.

The initial donor is selected by the general select donor algorithm. Using that donor as a starting point, the processor fix algorithm alternately assesses the effect of increasing the dispatching priority of the receiver (moving the receiver up) and decreasing the dispatching priority of the donor (moving the donor

Figure 5 State machine example



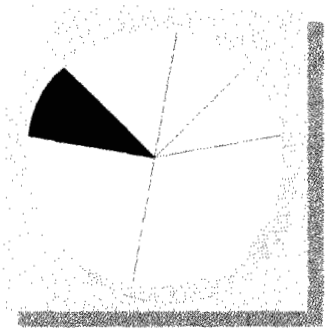
down) until the combination of moves produces sufficient receiver value or insufficient net value. After each tentative priority change, net value is checked for all classes affected by the change. If all affected classes pass net value, the set of interim moves is checkpointed, and receiver value is checked. If there is insufficient receiver value, the state machine proceeds to select another tentative move for the re-

ceiver or donor. If the net value check fails after any tentative move, secondary donors and receivers are selected to be moved up with the receiver or down with the donor to determine whether that combination of moves will pass the net value check.

If at any point a priority change has a projected detrimental affect on another class, the affected class may become a secondary receiver and be moved up with the primary receiver. Multiple combinations of secondary receivers moved up with the primary receiver, and secondary donors moved down with the primary donor, will be considered to the extent necessary to find a combination of priority changes that will improve the receiver without causing relative harm to other workloads. The state machine handles all combinations of primary and secondary receivers and donors.

If moving secondary donors and receivers is still not sufficient to pass net value, the secondary donors and receivers are moved back to the most recently acceptable set of checkpointed priorities that had shown acceptable net value. Then if it was the primary receiver moving up that failed net value, the moves continue with the donor moving down. Conversely, if it was the primary donor moving down that failed net value, the moves continue with the receiver moving up. In both cases, secondary donors and receivers are selected after every move if required to pass net value and to allow the assessment to continue. If even with moving secondary receivers and donors, neither the priority of the receiver nor the priority of the donor can change with acceptable net value, the whole set of tentative and checkpointed moves is abandoned and another donor is selected by the select donor algorithm. Then the whole process starts over with the new donor. The purpose of the state machine is to produce a comprehensive set of move combinations to evaluate, i.e., to leave no stone unturned in a search for changes to allow work to meet goals. However, in reality, the state machine tends to find valuable moves quickly because of the intelligence used by the select receiver and select donor algorithms when selecting initial candidates.

If a combination of priority changes with sufficient receiver value and net value is found, all the tentative priority changes are committed. The processor delay fix algorithm then exits and the MGDPC awaits feedback on the effect of its actions.

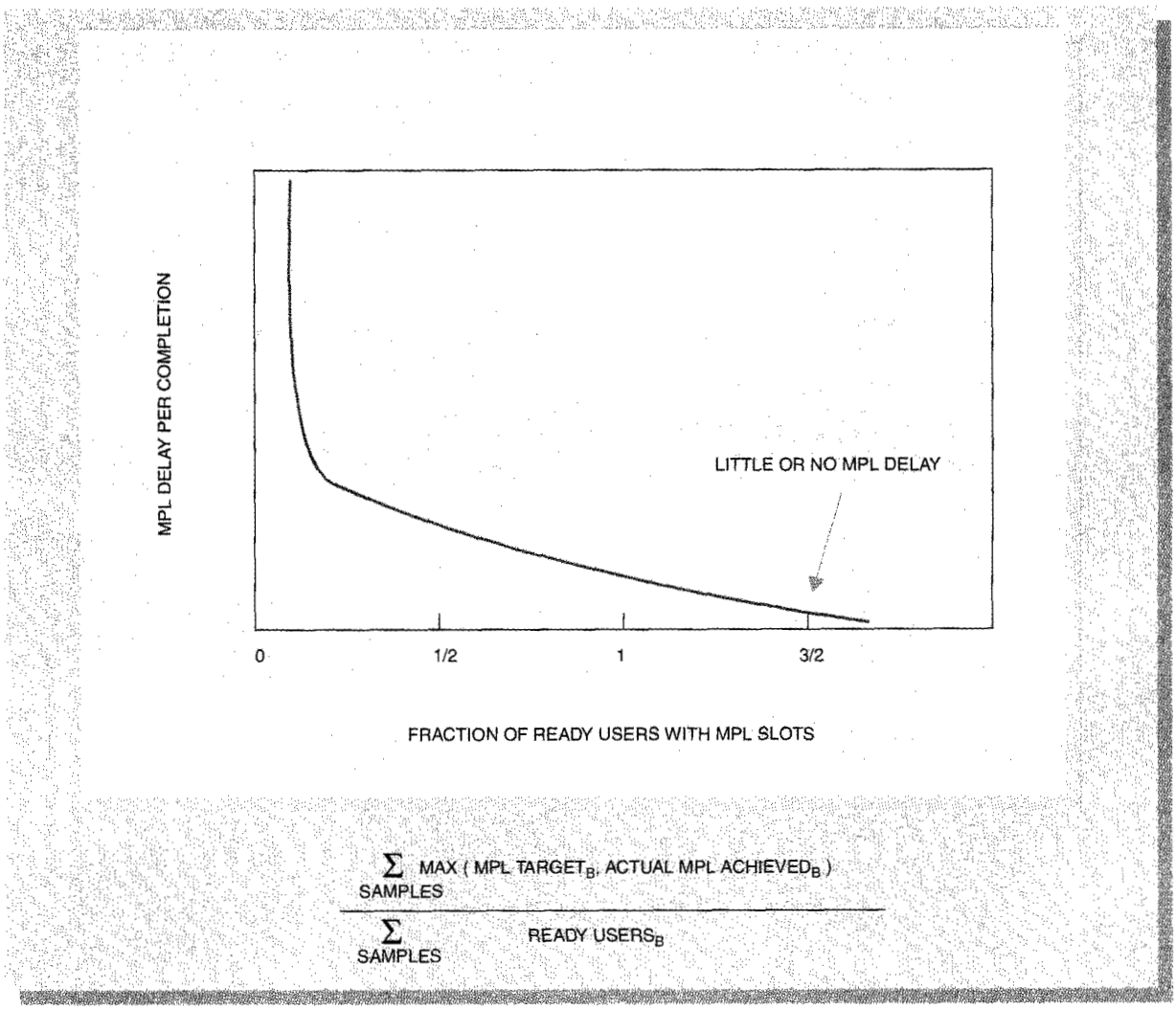


Multiprogramming level delay fix. This subsection describes how performance is improved by reducing the delay experienced by the receiver while it is waiting to be admitted to the multiprogramming set. An address space must be admitted to the multiprogramming set before it can be swapped in and execute. The controlled variable in this case is the number of MPL slots allocated to the class. One MPL slot represents one address space.

Theory. The MPL delay experienced by the receiver is a function of the fraction of ready users in the class that have MPL slots available to them. A ready user is an address space that is ready to execute. If there are fewer MPL slots allocated to the class than the class has ready users, some users will experience MPL delays. The class will not experience MPL delay if the number of MPL slots always equals or exceeds the number of ready users. The MPL delay fix algorithm uses an MPL delay plot to predict the effects on MPL delay of increasing or decreasing the MPL slots allocated to a class. At every policy interval, for each class, the MGDPC plots the most recent value of MPL delay per completion as a function of the fraction of ready users that have MPL slots available to them. Figure 6 depicts an MPL delay plot.

A complication arises when using the MPL delay plot because the number of ready users, required to read off the plot, is a function of the number of MPL slots. If there are too few slots, users will back up at any workload level. As slots are added, the number of ready users decreases. So the number of ready users is a function of MPL slots. The MPL delay fix algorithm uses another plot, the ready user average plot, to deal with this complication. The ready user average plot (Figure 7) records the relationship between the number of ready users in a class and the number of MPL slots available to them. The ready user average plot is used to predict the number of ready users when assessing an MPL target change. The plot can show the point at which work units will start backing up. The number of ready users read off the ready user average plot is used to determine which point to read from the MPL delay plot.

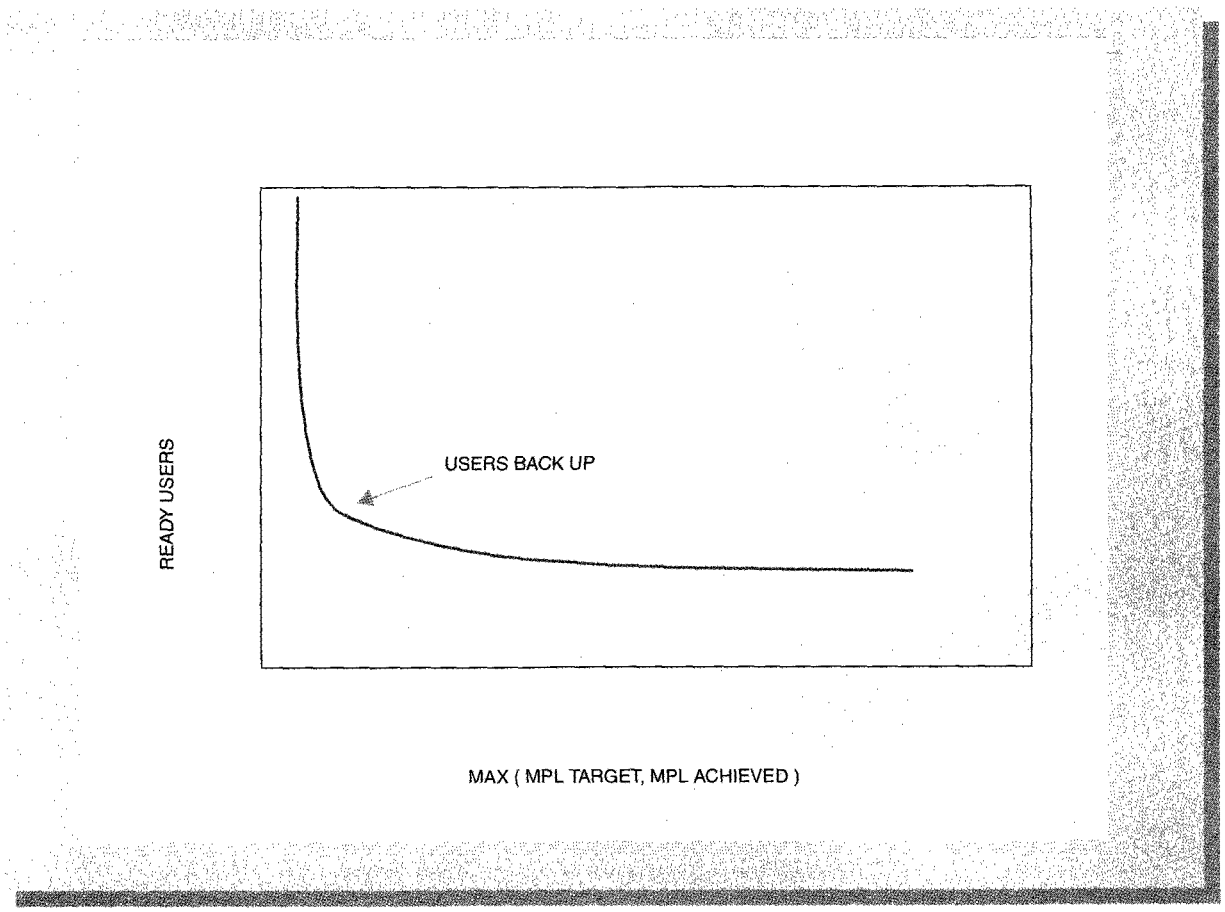
Figure 6 Class B MPL delay plot



Operation. In the operation of this algorithm, first the MPL slot increase necessary to satisfy receiver value for the receiver class is found. This is done by adding one to the current MPL slot allocation of the class, using the new number of MPL slots to read the new number of ready users off the ready user average plot, using the new number of ready users to read projected MPL delay off the MPL delay plot, converting the new MPL delay to an MPL delay sample delta, using the new delay sample delta to project a performance index delta, and using the projected performance index delta to determine receiver value. If there is not sufficient receiver value, another MPL slot is tentatively added, and all the calculations are repeated.

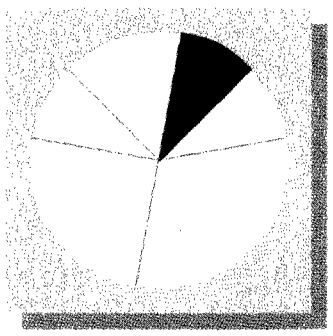
When a number of MPL slots with sufficient receiver value is found, it is necessary to find storage to accommodate the additional swapped-in address spaces. Otherwise, simply adding address spaces could cause storage contention and other problems. Storage donors are identified using the find donor algorithm. The projected delay sample deltas are projected by the algorithm specific to the resource

Figure 7 Ready user average plot



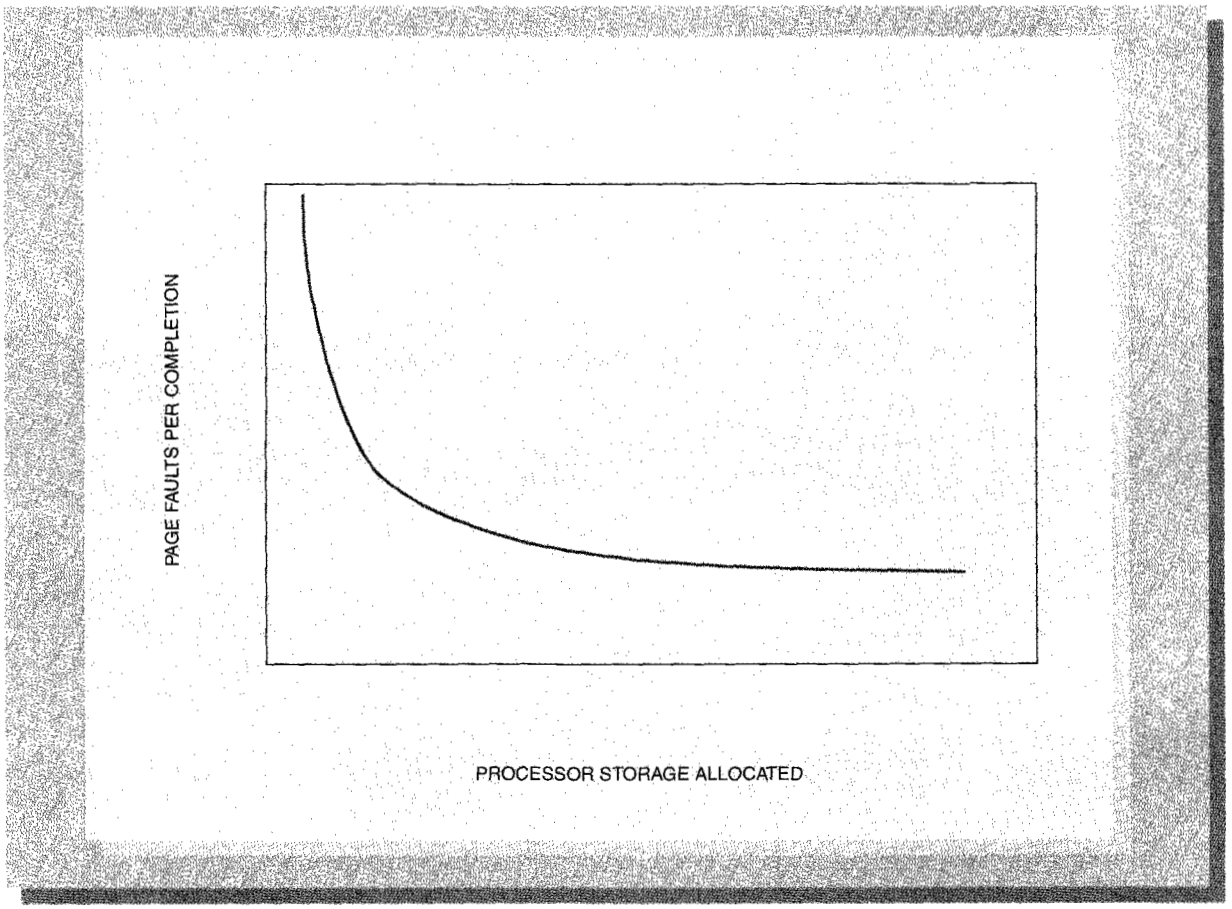
being taken (MPL slots or storage); the performance index delta is calculated as described previously; and the net value algorithm is used to determine whether the donation has value. If necessary, additional donors are identified and evaluated until an acceptable set of donors, able to donate the required storage, is found or all donors have been evaluated and all donations have been found to have insufficient net value.

If a change with sufficient receiver value and net value is found, the additional MPL slots are allocated to the receiver, and all the storage donations are committed. During the next policy interval, the receiver will use the new MPL slots, and the donors will be allowed to use less storage. The MPL delay fix algorithm then exits, and the MGDPC awaits feedback on the actual effect of its actions.



Disk paging delay fix. This subsection describes how performance is improved by reducing the disk paging delay experienced by the receiver. The controlled variable in this case is the number of processor storage frames protected for an address space. The protected number of frames is referred to as the protected processor storage target. The operating system will not steal frames from the address space below the protected processor storage target of the address space.

Figure 8 Page fault rate plot



Theory. The disk paging delay experienced by the receiver is a function of both the number of page faults taken by work units in the class and the time required to satisfy the page faults. The number of page faults taken is a function of the processor storage allocated. The time per page fault is not a function of the processor storage allocated. It is a function of the demand put on the paging subsystem by all of the work units in any classes taking page faults. Both the number of page faults taken by the class and the time per page fault must be used in combination to accurately predict paging delay changes.

The disk paging delay fix algorithm combines two techniques to predict disk paging delay changes. The first technique is the page fault rate plot, shown in Figure 8. This is a plot of page faults per completion as a function of processor storage allocated. A point is plotted on the class page fault rate plot after every few transactions in the class complete. The plot always reflects the latest condition but also remembers the page fault rate for the class when the class was allocated a larger or smaller number of frames. This plot is used to predict a new number of page faults per completion given a contemplated change to processor storage allocation.

After a new page fault rate has been read off the plot, disk paging delay samples are used to predict the new time that will be experienced because of disk paging delay. This prediction is arrived at by taking the ratio of the change in page fault rate and multiplying it by the disk paging delay samples experienced by the class. The calculation follows:

$$\text{projected\_samples\_delta} = \frac{(\text{new\_page\_fault\_rate} - \text{old\_page\_fault\_rate}) \times \text{disk\_delay\_samples}}{\text{old\_page\_fault\_rate}}$$

If a page fault is taking a long time because of other demands on the paging subsystem, this situation is reflected in the number of delay samples experienced by the class. Introducing disk samples into the algorithm serves the function of introducing time per page fault into the algorithm. The number of page faults and the time per page fault used in combination are accurate predictors of the disk paging delay that will be experienced by a class after a processor storage allocation change. The performance index deltas are calculated from the delay sample delta as described previously.

Operation. In the operation of the algorithm, first the storage allocation increase necessary to satisfy receiver value for the receiver class is found. This is done by reading the page fault rate corresponding to increasingly larger numbers of processor storage frames off the paging rate plot. Delay sample deltas and performance index deltas are calculated as described previously, and the receiver value algorithm is applied until a storage increase with sufficient receiver value is found. The required storage increase per address space multiplied by the number of swapped-in address spaces yields the total number of frames required. The storage required is found by the find donor algorithm, and the value of the storage reallocation is evaluated using the net value algorithm as described previously.

The paging rate plot captures the nonlinear relationship between the amount of storage allocated to work and the value of the storage to that work as measured by the page fault rate of the work. For work on the right side of this plot, additional storage will be of little benefit, whereas the same amount of storage could provide a very large benefit to work on the left side of the plot. All other things being equal, work on the left side of its paging rate plot will tend to be a receiver of processor storage, and work on the right side of its paging rate plot will tend to make a good donor of processor storage.

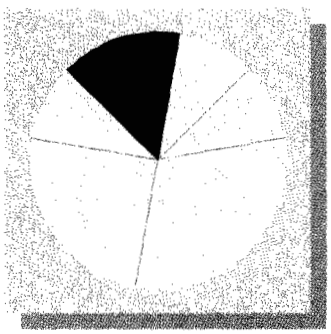
Long-running transactions. The preceding discussion assumed that each address space in the class would have similar storage requirements and benefit similarly from the same amount of storage. This assumption is true for classes where each transaction is relatively short. In these cases, each transaction in the class is given the same allocation of processor storage as soon as it arrives in the system. This technique allows short transactions to receive the benefit of the storage before the algorithms would have had time to learn the particular paging characteristics of each transaction. In cases where the transactions are longer, the transactions in a class, and their storage requirements, are more likely to be different from one another. Also in this case, the algorithms can afford the time to learn about each transaction individually. To learn, the algorithms build a paging rate plot for each individual transaction that experiences significant paging delay. This plot is used similarly to the class paging rate plot. To project the effect of giving storage to a receiver, a new paging rate is read off the transaction paging rate plot. Then a projected number of samples is calculated for the transaction in the same way as a new number of paging delay samples was calculated for a class. The final step is to project the delay sample delta for the class by multiplying the delay sample delta for the transaction by the proportion of the paging delta of the class attributable to the transaction.

Client/server considerations. To improve the performance of a client class by reducing the paging delay seen by a server class, the delay sample delta is calculated as described above for the nonserved case. Then the projected samples are apportioned back to the client class, and a new proportional aggregate speed is calculated for the client class. The proportional aggregate speed plot is read to obtain the projected client class performance index and calculate the projected performance index delta as described previously.

Feedback. In all cases, if a change with sufficient receiver value and net value is found, the additional storage is allocated to the receiver by increasing the protected processor storage target of the receiver, and all the storage donations are committed. The storage donations may be in the form of reducing MPL slots or reducing protective processor storage targets. The disk paging delay fix algorithm then exits, and the MGDPC awaits the next policy interval to obtain feedback on the effect of its actions.

Anticipatory resource allocation. The previous topics on fixing storage-related delays all discussed how

the MGDPC responded to situations where a class was experiencing delay waiting for MPL slots or paging. However, a good system programmer would not just wait for a problem with these resources to occur. The MGDPC does not just wait for problems either. The MGDPC anticipates what storage is needed by classes and allocates MPL slots and protective processor storage targets in advance to prevent problems. The MGDPC does these anticipatory allocations to the extent that the storage resource is not needed to solve problems that another class is experiencing. The MGDPC reconsiders these anticipatory allocations every policy interval, providing another mechanism for the MGDPC to respond to changing situations. The anticipatory allocations require no input from the customer. The MGDPC determines these allocations by observation.



I/O delay fix. This subsection describes how performance is improved by reducing the I/O delay experienced by the receiver. The controlled variable in this case is the I/O priority.

Theory. Managing access to I/O devices has many parallels with managing access to the processor. Both have using time and wait time, which suggested a wait-to-using algorithm. The concept of maximum demand, used successfully in processor management algorithms, is also applicable. This concept led to I/O priority assessment algorithms that in many ways paralleled the dispatching priority algorithms. I/O maximum demand and I/O wait-to-using measures are used, and the underlying concepts in the I/O projection algorithms are very similar to the concepts in the processor projection algorithms. A state machine is used to make a comprehensive search for I/O priority increase and decrease moves and secondary donors and receivers. The operation of this state machine is similar to the operation of the processor state machine described previously.

Resources subsets. There is a complication with I/O devices in that they, unlike processor and storage, are not a common resource. All work does not use all devices. If the MGDPC was going to affect performance by changing the I/O priorities of receivers with respect to donors, it had to know that the donor actually affected the receiver. If the receiver uses devices a, b, and c, and the donor uses devices x, y, and z, changing the I/O priority of the receiver with respect to the donor will have no effect. The MGDPC solved this problem by determining disjoint subsets (clusters) of I/O devices such that it knew, for example, that service classes a and b use the devices in cluster 1, and classes c, d, and e use the devices in cluster 2, and so on. The MGDPC dynamically builds these cluster and class relationships every ten minutes to reflect changes in how the work in the classes is using the devices.

Multisystem shared resources. Another problem involved with management of I/O priorities is that I/O devices, again unlike processor and storage, can be shared among systems. Managing I/O priorities on one system would be an incomplete solution in a sysplex. It led to the more general problem of being able to manage resources shared by multiple systems to meet performance goals.

When processor priorities are changed, the changes need only be done on one system because only work on one system is affected. However, when shared resources are involved, the changes must be propagated across all systems that share the resource. For example, if class a is running with an I/O priority of 253 on one system, it must run with an I/O priority of 253 on all systems to maintain its priority relative to other classes. If changes were not propagated across all the systems so work used consistent priorities, the changes on any one system would have an unpredictable effect. The MGDPC solved this problem by coordinating the I/O priority changes.

A fundamental and very valuable attribute of the MGDPC algorithms is that the systems in the sysplex are independent as well as cooperating. There is no master system. Each system sees the same data and can make changes to any resource that the MGDPC manages. A complication arises with shared resources where any system can make I/O priority changes that it expects all systems to implement. The MGDPC solved this problem by continuing the philosophy that any system can make changes, but the MGDPC added coordination such that only one set of changes is propagated to all the systems at any one time. To reduce the instances of frequent competing I/O priority changes and to encourage the instances of the MGDPC to work on other problems such as storage or processor delays, the MGDPC added the requirement that the MGDPC on each sys-

tem had to wait "n" number of intervals since the last I/O priority change by any system to make more I/O priority changes. The "n" used is six, so each system knows it has to wait six intervals before considering more changes. Maintaining the independence of the systems is very important because it allows each system to work on its local performance problems if all the sysplex goals are being met and eliminates many problems of a master-slave operation.

Multisystem goal-driven performance controller in action. This subsection describes an experiment that was run to show how WLM can manage a large commercial workload. The experiment had two phases. In the first phase, an on-line transaction processing and interactive workload was run. We discuss how WLM sets dispatch priorities for this work based on the goals and importance of the work. In the second phase, a large batch job stream was added to this mix. We next discuss how WLM adjusted to this change in the workload to continue to meet the goals of the WLM policy. It should be noted that in order to show the robustness of the WLM adjustment algorithms, the second phase of this experiment overloaded the processor capacity of the system in a way that a commercial environment with important on-line work would be unlikely to do.

The on-line transaction work consisted of two transaction-processing subsystems: Customer Information Control System Version 4.1 (CICS V4.1) and Information Management System Version 5.1 (IMS* V5.1). Both CICS and IMS are considered servers by WLM (see subsection on server topology). The interactive work was made up of 350 simulated users of the OS/390 Time Sharing Option (TSO). The batch work consisted of 10 large jobs designed to simulate commercial batch operations. This work was divided into two service classes, BatchHi and BatchLow. The workload was run on an IBM ES/9000*/9021 with two CPUs.

Table 1 summarizes the WLM policy used for the experiment.

A significant observation about this policy is that the most important work in the system is made up of the IMS transactions. The least important work consists of the two batch service classes. The CICS transaction and TSO users are of medium importance.

There were two interesting phases to this experiment. First, the nonbatch workloads were started and stabilized. During this interval the system was about 80 percent utilized, and there were no storage constraints. The order of dispatching priorities that WLM chose for the work was:

1. CICS address spaces
2. TSO Period 1
3. TSO Period 2 and TSO Period 3
4. IMS address spaces

This order might be considered a surprising result given that the IMS transactions are the most important work in the system. The explanation is that although the IMS transactions are the most important, they are also very easily meeting their goal as shown by an average performance index of 0.12 over this interval. Table 2 shows the average performance indexes of each service class and the percentage of the CPU that each service class was using during the first phase of the experiment.

Notice that all the other service class periods have significantly higher performance indexes than the IMS transaction class (IMSTRX). If the IMS address spaces were given a higher dispatch priority, it would increase the difference between the performance indexes of the IMS transaction class and the other service class periods.

Table 1 Goals for mixed workload with IMS more important

Service Class Period	Type of Goal	Goal	Importance
CICSTRX	Response time	0.090 sec	Medium
IMSTRX	Response time	1.000 sec	High
TSO Period 1	Response time	0.100 sec	Medium
TSO Period 2	Response time	1.000 sec	Medium
TSO Period 3	Response time	3.000 sec	Low
BatchHi	Velocity	7%	Lowest
BatchLow	Velocity	1%	Lowest

Table 2 Average performance index and CPU percentage

Service Class Period	Performance Index	CPU (%)
CICSTRX	0.70	24
IMSTRX	0.12	23
TSO Period 1	0.52	11
TSO Period 2	0.34	4
TSO Period 3	0.31	8
All work	N/A	70

Table 3 Performance index and CPU percentage after batch started

Service Class Period	Performance Index	CPU (%)
CICSTRX	0.65	24
IMSTRX	0.07	22
TSO Period 1	0.51	12
TSO Period 2	0.45	4
TSO Period 3	0.66	8
BatchHi	0.82	25
BatchLow	0.60	4
All work	0.60	99

Now consider how a system programmer might go about setting the dispatch priority for this workload. Given that the IMS transactions are the most important work for this installation, the system programmer would probably not give the IMS address spaces the lowest dispatch priority. If IMS address spaces were given a higher dispatch priority, response time for at least TSO Periods 2 and 3 would be unnecessarily elongated, whereas the IMS transactions would beat their goal by even a larger amount.

If the system programmer did set the above dispatch priority order, the response time of the IMS transactions would have to be constantly monitored to look for a change in the workload that would cause the IMS transactions to miss their goals. If such a change did occur, the system programmer would have to detect it and decide how to change the priorities on the fly before too much damage was done to the IMS transactions.

The second part of the experiment was to start the batch work. With the batch work running, the overall processor demand of the total workload was significantly more than the system could deliver. There still was no significant storage contention. Figure 9 shows how processor service was consumed by each of the different types of work during the overall run.

The batch work was started at about 15:47. Notice that the batch processor service immediately jumps to a peak of about 23 000 with a corresponding drop in the processor service for the IMS address spaces and TSO work. Figures 10 through 12 show plots of how the performance index for the work changed during the run. Figure 10 is for IMS and BatchHi and Figure 11 is for the TSO periods. Figure 12 shows the same data on one plot. Note that the performance indexes for the IMS transaction class and the TSO service class periods shoot up as their corresponding processor service goes down. Because the IMS transaction class is the most important, WLM first addresses its problem by increasing the dispatch priority of the IMS address spaces relative to batch. The result of this action shows clearly in Figure 9 as the IMS processor service recovers as quickly as it dropped off. The processor service of TSO recovers after that of IMS since WLM addresses its processor delay problem as the next most important work missing its goal. After WLM went through several steps of incrementally improving the performance of TSO by adjusting TSO Period 2 and Period 3 dispatch priority versus the BatchHi class, the final dispatch priority order that WLM sets is:

1. CICS
2. TSO Period 1
3. IMS
4. TSO Period 2, TSO Period 3, and BatchHi
5. BatchLow

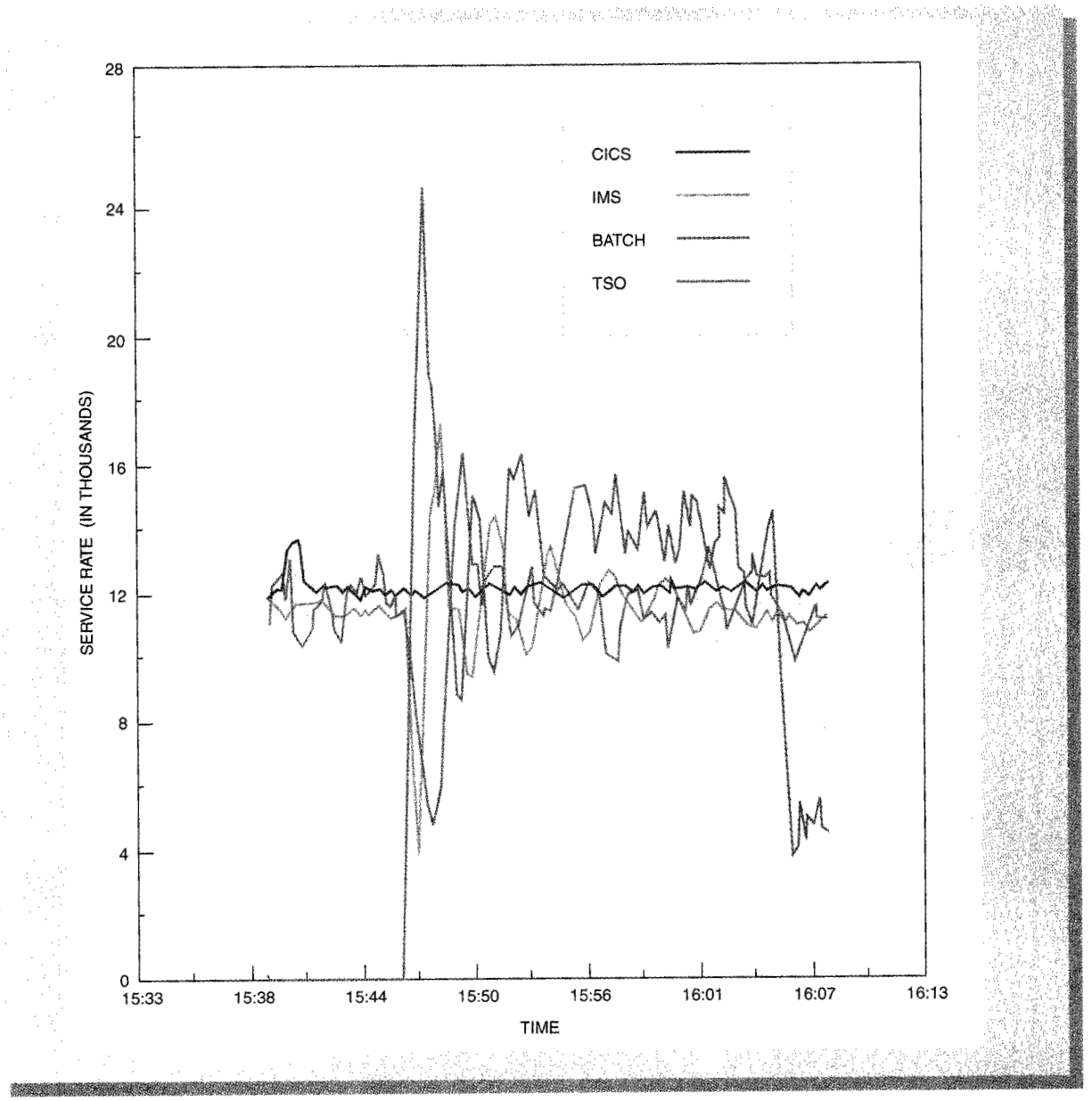
Examining the graph of performance indexes shows this dispatch priority order allows all the nonbatch work to meet its goals almost all of the time. Table 3 shows the average performance index and the percentage of the CPU each service class was using after the batch started and WLM had a chance to readjust dispatch priorities.

Notice that the work that is most affected by the addition of batch operations is TSO Period 2 and Period 3. Before the batch work started, the average performance indexes for these periods were 0.34 and 0.31. After the workload has stabilized again following start of the batch, the average performance indexes for these periods increase to 0.45 and 0.66. Even the BatchHi service class is able to meet its goal on average, though it has the highest average performance index which is reasonable, since it is the least important work.

Given that overall this workload requires more processor power than is available, some work is not going to run. Since BatchLow is the least important work with the easiest goal, it is the work that is sacrificed. Figure 13 shows the service rate of BatchHi versus BatchLow. Other than a small burst of service before WLM readjusted the priorities for the new work, BatchLow does not run until the jobs in BatchHi begin to finish at about 16:01.

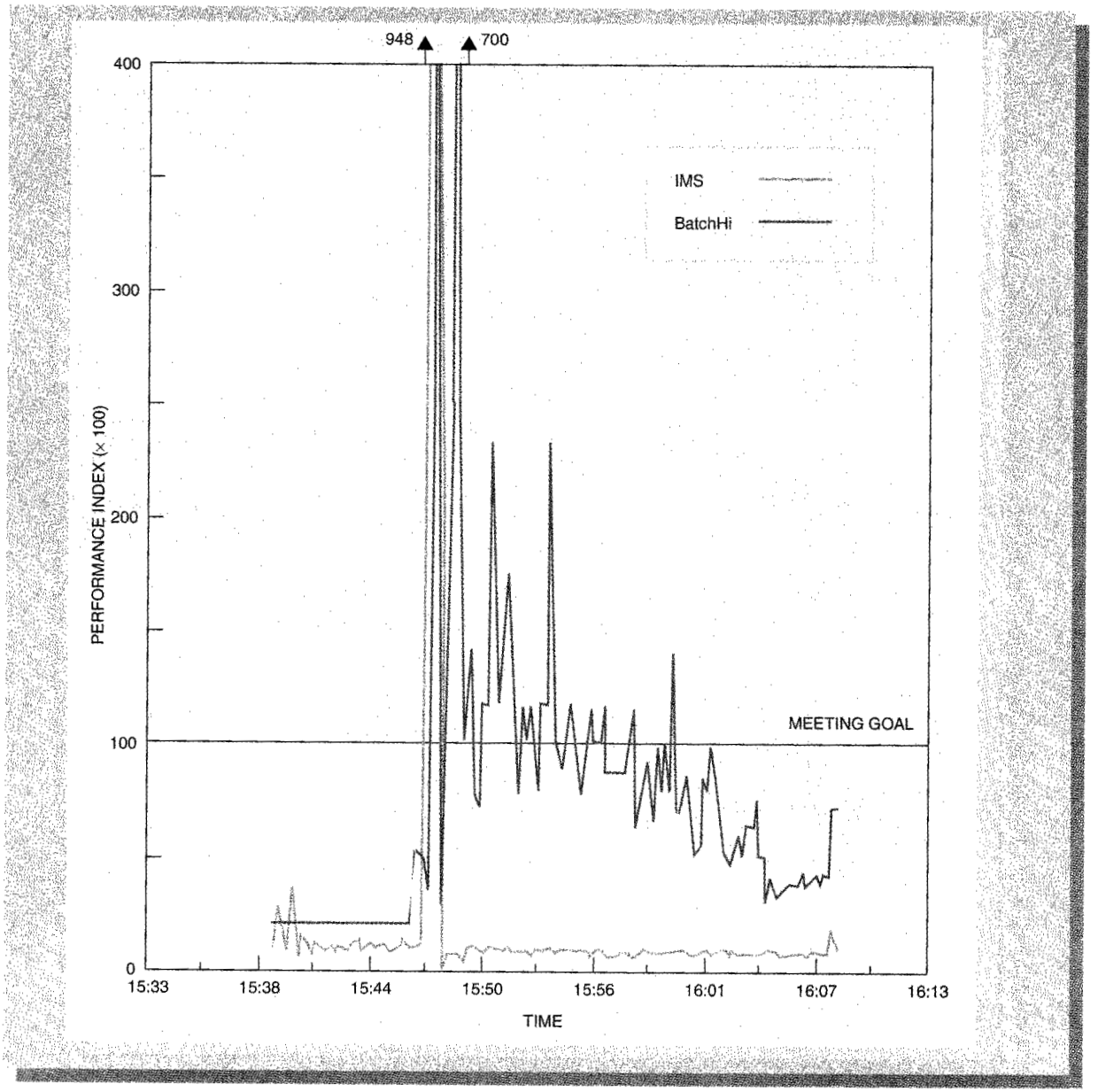
In summary, this example shows how the MGDPC function of WLM is able to allocate system resource

Figure 9 Service rate by work type



to a diverse workload to meet the performance goals of the installation. Because the MGDPC is continually monitoring how the work in the system is performing, WLM can be more aggressive than a system programmer in reallocation of resources to the work in the system having the biggest problem meeting its goal even if it is not the most important. The order in which the MGDPC chooses receivers and do-

Figure 10 IMS and BatchHi performance index

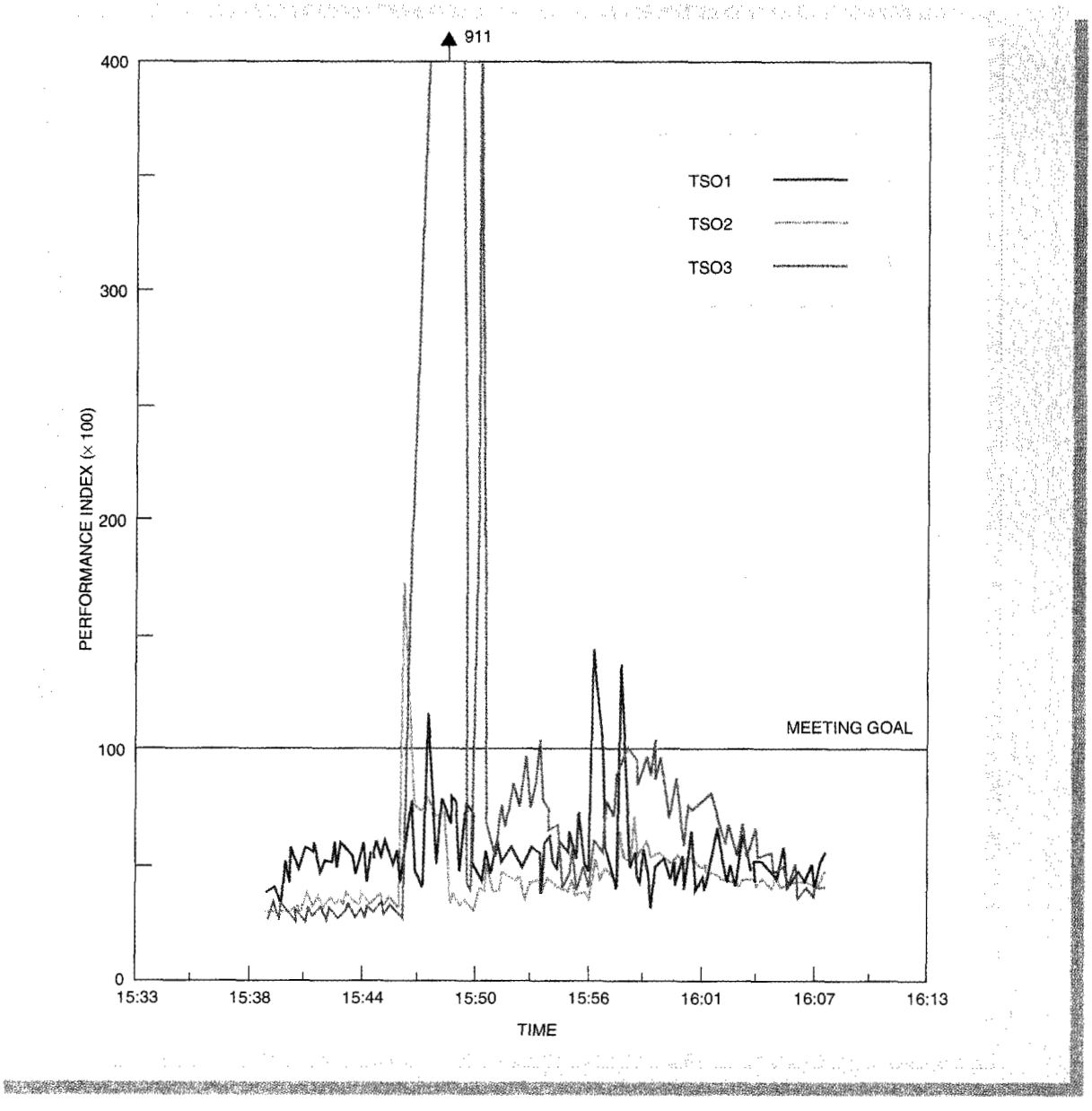


nors and its net value check ensures that such a reallocation does not hurt more important work. The first phase of this experiment shows the results of such actions. The second phase of the experiment shows that WLM can quickly react to major workload changes in reallocated resources as necessary to best meet the performance goals of the installation.

Balancing work across a parallel environment

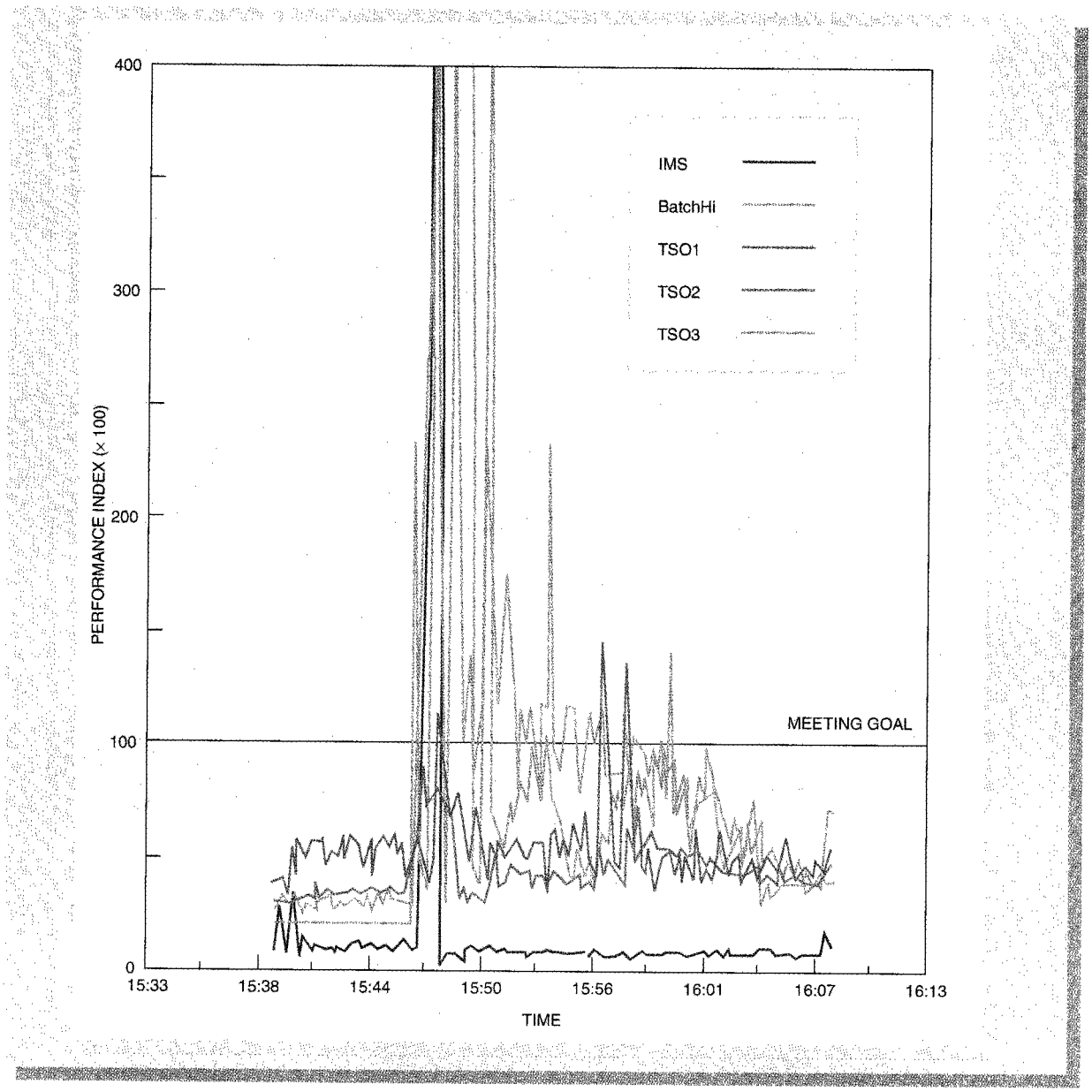
A number of problems arise with the existence of multiple images that share work and resources. This section describes how WLM addresses these problems while remaining focused on the goals specified by the business policy.

Figure 11 TSO performance index



In many interactive or client/server environments, a single user, while under some application suite, will remain "connected" to a particular instance of an application server running in the parallel environment for a protracted period of time. This time frame may extend for minutes or hours or days, depending on the nature of the application and the activities of the end user. Furthermore, the intended duration of this "connection" at the outset is unknown to the operating system or even the application itself in general. A further complication is that it is not known at the outset what will be the resource requirements of the end user, nor is it known what business goals and importance will be associated with this work.

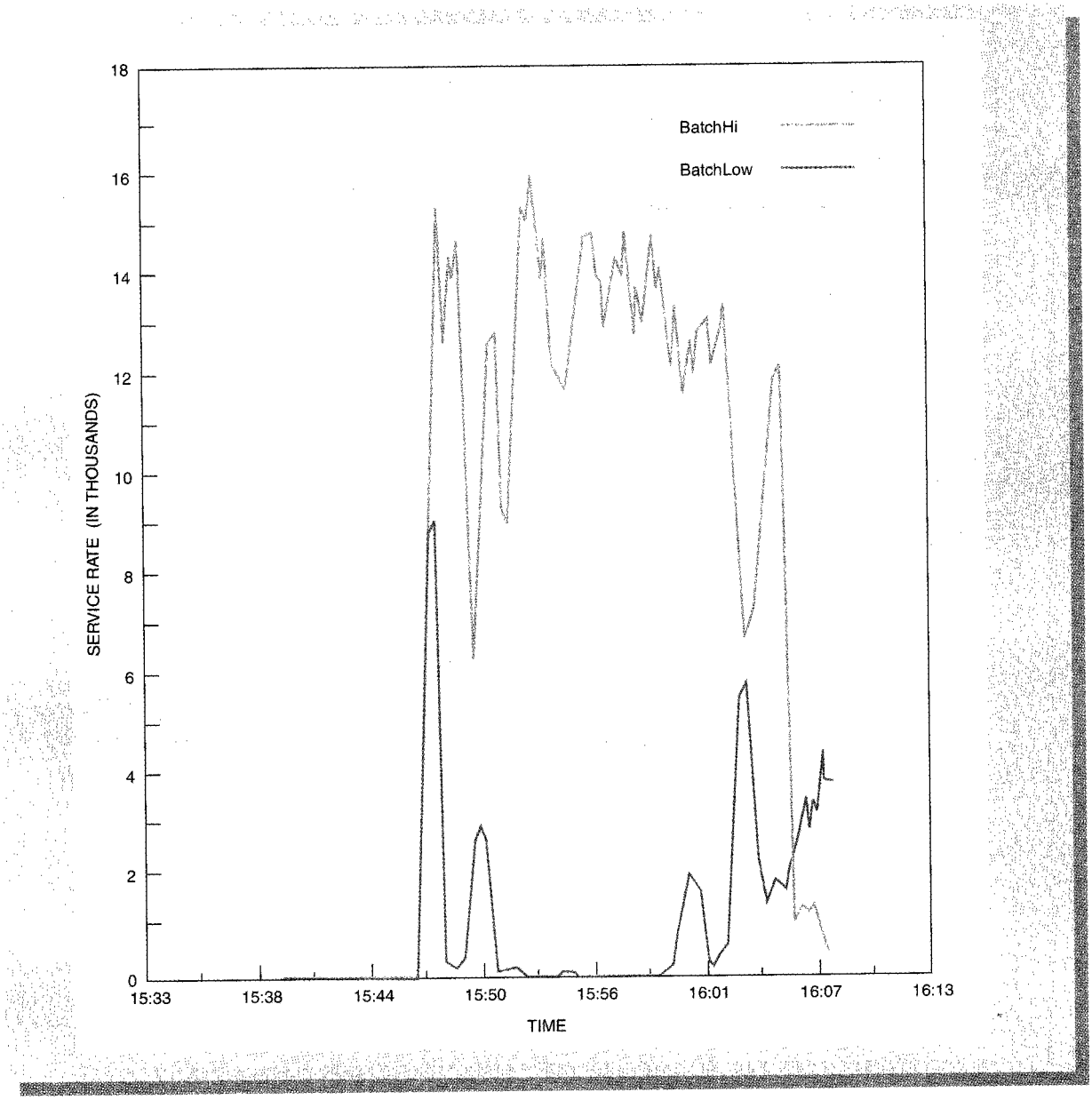
Figure 12 Performance indexes



Since workload conditions in the parallel environment may change while the "connection" exists, it is impossible to guarantee that a decision to "connect" the end user to a particular server instance on one image will remain optimal for the entire duration. For a variety of reasons, including network protocols, the existence of transient data, and recovery schemes, an end user cannot be arbitrarily reconnected to another image or application server instance by the operating system in order to rebalance work when conditions change.

Prior to WLM, the techniques to solve this problem in an OS/390 environment involved planning the network connections so as to spread sessions across as many images as necessary to balance the workload.

Figure 13 Batch service rate



End users needed to be aware of which server or image they would be connected to in order to use their application. Some improvement was offered with Virtual Telecommunications Access Method (VTAM*) support of generic resource, wherein sessions (connections in a Systems Network Architecture, or SNA, world) were balanced across eligible logical units (an LU is an SNA session endpoint in this context). However, this support did not incorporate any knowledge of utilization or machine size. Techniques such as "round-robin" have a similar deficiency, and furthermore do not incorporate knowledge when "connections" are no longer active.

The approach taken by WLM in the face of the above limitations and uncertainties is to "connect" the end

user to a server instance on an image that is meeting its goals and that has a threshold amount of "displaceable capacity" at the lowest importance level among eligible servers. Displaceable capacity at the lowest importance level refers to processor capacity that is either unused or that is consumed by recently observed work that is as low in business importance (as given by the business policy) as possible to achieve the threshold amount. The value for the threshold is dependent on the workload to which the user belongs and may be calculated based on samples that are taken as the workload runs, or may be based on historical values for the workload or on default values.

The principle behind this approach is that work requests created by the user will either have access to unused capacity or will compete with work that is deemed least significant by the installation business policy. In the latter case, the WLM algorithms will adjust resources, as needed, to ensure that the most important work achieves its business goals. This adjustment is the meaning of the phrase "giving work requests the 'best chance' to meet their goals."

Initially, this approach should give work requests created by the end user a maximal opportunity to achieve their goals. Over time, workload conditions may change and leave the end user's work less capable of meeting its goals, at least as compared to other server instances or other images. Of course, if these work requests are deemed to be sufficiently important by the business policy, the algorithms should ensure that sufficient resources are available to meet their goals. However, this condition leaves open the question of what to do for work requests that are not sufficiently important when other images may now have unused capacity or less important work.

This problem is addressed using techniques that are workload-dependent. For example, some workload environments have a further layer of "transaction routing," wherein a work request arriving at an application server is then forwarded to another application server instance that is a better choice. CICS V4 and CICSPlex* System Manager (CP SM) cooperate to implement this approach as one example. Another approach is to dynamically change the "connection" based on the server's awareness of when this change can be done transparently. DB2* (DATABASE 2*) V4 is able to perform this change for distributed SQL (structured query language) requests to a remote data-sharing group, as it spreads such requests across those members in the proportions recommended by WLM. Indeed, DB2 will poll WLM on a regular basis to ensure that the distribution pattern matches current conditions. A third possibility is that the work request may be split up, or parallelized, to run on multiple processors (i.e., instruction streams) either within the same image or on different images. For example, DB2 V4 can parallelize queries in this fashion. As with "transaction routing" discussed below, some assessment must be made that using these techniques will overcome their cost.

Note that even when "transaction routing" can be implemented, it imposes an additional "hop" and therefore additional cost, which could reduce throughput and increase response time. It is therefore desirable to choose the target "connection endpoint" carefully, mindful that conditions will change, possibly unpredictably, and that elaborate analysis may be counterproductive.

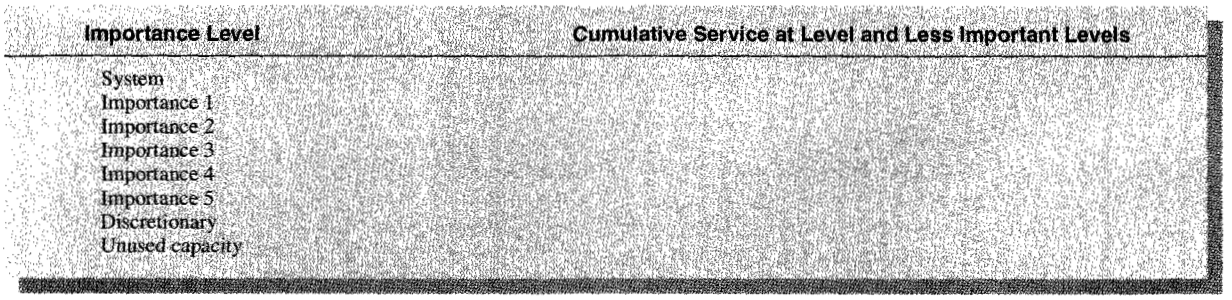
WLM supports a number of environments that exhibit such "long-term" connections to a server through a variety of interfaces, including generic resource, domain name server, and sysplex routing.

The generic resource and domain name server allow a group of equivalent servers to be treated as a single entity by end users when requesting a "connection." Generic resource is used in the SNA world, where a "connection" equates to a session. During initialization, each server identifies itself as belonging to a particular group and gives the (LU) name(s) with which it is associated. Domain name server is used in the TCP/IP (Transmission Control Protocol/Internet Protocol) world.

The OS/390 domain name system (DNS) implementation allows system administrators to set up a common host name for a set of OS/390 systems. When DNS is queried for IP address resolution, WLM services are used to choose the best system or server to place the new work. In this way the DNS/WLM resolutions cause incoming TCP/IP requests to be distributed intelligently across the sysplex.

Sysplex routing allows a group of equivalent servers to be registered and monitored for purposes of routing individual work requests among its members. WLM will provide recommendations on what proportions should be allocated to each server within the group for a narrow window on the order of a few minutes in length. Users of this service are then able

Figure 14 Importance level service summary table



to spread individual work requests across multiple servers in these proportions so as to enhance their chance of meeting their goals.

WLM also provides interfaces to allow a product that coordinates and provides services for a collection of related server address spaces to query WLM for its recommendation on which server space is the best choice for a set of related work requests. Typically a daemon process within the product will interact with WLM to manage such work requests. This interaction moves the scheduling responsibility from the daemon to WLM, where work for the entire parallel environment is monitored and managed.

Each of these interfaces draws on common samples, measurements, and projections of system activity that are described next.

Balancing data. WLM implements its routing decisions and makes recommendations on the basis of five types of information. The first indicator is the presence of resource constraints in the recent past. These constraints would include shortages of processor storage, paging space (secondary storage in a UNIX** environment), etc., or dangerously high levels of paging or swapping. Systems with such problems are automatically given the lowest possible recommendation value to receive new work, since they will likely be unable to start new server instances and be otherwise unlikely to support an increased workload. In fact, such a system would be operating in a mode to shed work since it is seriously overloaded.

Next, WLM maintains a dynamic list of eligible servers, along with the image on which it is located, and any other information needed to uniquely identify each server instance. This list is updated not only with the startup and shutdown of servers themselves, but also with the unexpected failure of images within the parallel environment. This list is necessary so that WLM can recognize which servers exist and make a choice (or choices) among this list.

In addition to tracking the presence of servers, WLM evaluates the ability of each server to meet the goals of work requests that have flowed through the server using an aggregated performance index (PI). This PI takes into account the various importance levels for such work requests and their contribution to the universe of work requests that the server accepted in the recent past, and projects what the PI will be as a result of the policy actions taken. The significance of the PI is that it incorporates the effect of all activity in the system and reflects the real delivered responsiveness measured against the business goals. Delays include contention and constraints for all resources, including storage, locks, queuing effects, etc.

The third type of information used by WLM is the importance level service summary table, which tracks the normalized processor consumption of work at each importance level on each image. To be more precise, the cumulative processor service delivered to all work at the given level and for all less important work is maintained. This table includes unused capacity, discretionary work, and system overhead not directly attributable to any particular work request. The purpose of this table (see Figure 14) is to allow the WLM algorithms to understand where "displaceable capacity" exists on an image-by-image basis, and where work may compete most favorably for processor access.

The fourth type of information is an assessment of the average cost of each work request that flows through the server. For some work environments, this assessment may be based on historical measurements or, alternatively, may represent a default cost associated with a single end user. For other environments, this assessment may reflect the measured average cost of an end user over some recent interval. The purpose of this estimate is to set the threshold value for "displaceable capacity" needed, and also to estimate the latent demand that arises when a new "connection" has been established but before actual demand shows up in new measurements kept in the importance level service summary table and the aggregate PIs.

The fifth type of information used by WLM is the list of recent selections that allows some projection of latent demand of new connections, as discussed in the previous paragraph. This information is aged out fairly quickly as new measurements pick up the actual demand that has been introduced.

The above information is maintained on an ongoing basis during normal system execution. When a WLM interface is invoked to make a recommendation, as described above, WLM will also calculate the target service value for "displaceable capacity" that is needed. This value incorporates both the average cost and an estimate for latent demand that is reflected by the list of recent connections.

Balancing algorithm. With the above data in hand, the general approach in deciding how to place a new "connection" is to go through the list of all servers one at a time and assess whether the current server is a better choice than a server previously chosen. A server on an image that is not resource-constrained will be given preference to a server on a resource-constrained image. Beyond that filter, preference is given to servers that are meeting goals, then to servers that are narrowly missing goals, and finally to servers that are badly missing goals. Within each of these three categories, servers are ranked according to the importance level at which the target "displaceable capacity" is achieved, with preference given to utilizing the lowest-possible business importance.

If the interface allows multiple selections (as for sysplex routing), the algorithm will keep all selections that survive with the same attributes for resource (un)constraint, degree of meeting goals, and target importance level. Weights are set according to the ratio: (target displaceable capacity)/(total capacity of all images at target importance level).

The rationale for the above ordering reflects a number of trade-offs. As has been discussed previously, systems that are recently resource-constrained are immediately shunned since they are overloaded, and the system is actively reducing the workload that is allowed to run by disallowing new spaces from being created, reducing the number of spaces that are swapped in, and so forth.

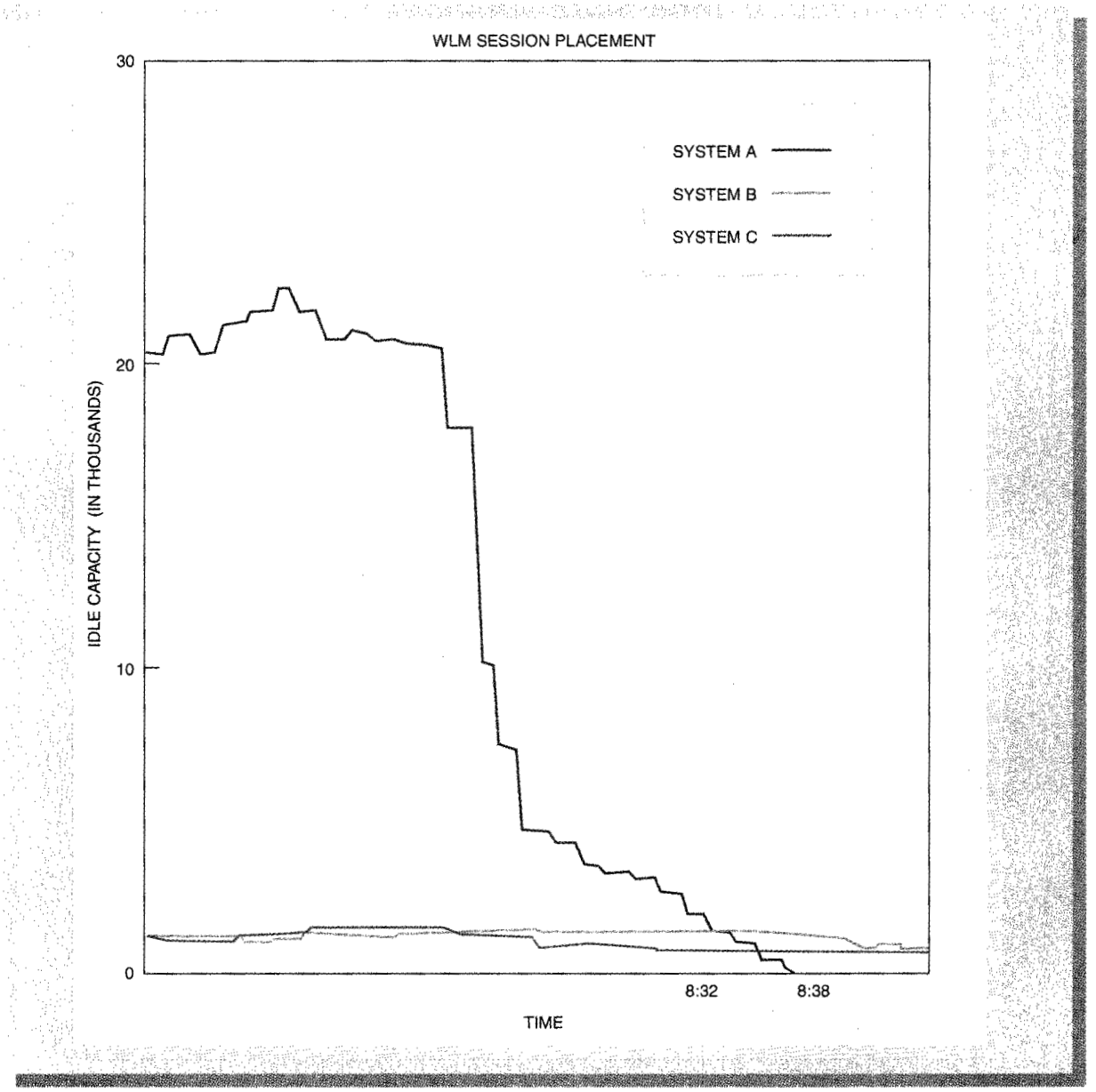
In the absence of a resource constraint (or when all relevant images are similarly constrained), the ideal server choice is one that is meeting goals for its work and that resides on an image with sufficient displaceable processor capacity to accommodate new work. Ideally this would be unused processor capacity, but in any case, there is a preference to compete with work at the lowest possible importance level so that new work will be favored as much as possible.

A server that is narrowly missing its goals but with sufficient unused processor capacity to accommodate new work is almost as good as a server that is meeting its goals, since the resource management algorithms will likely address the problem—which would generally be one of processor storage allocation.

In looking at the above categories used in ranking server choices, it is worthwhile to observe that a strong reliance is placed on the actual performance of servers against the goals of the work they serve, as measured and projected by their aggregate PI. This observation reflects a philosophical bias to use actual observed behavior and to value feedback so as to correct inaccurate assumptions that might be made from other measurements or design points. This concern has been discussed in other papers. Reference 9 discusses sensitivity to inaccuracy in the values of communication costs, locality statistics, etc. Also observe that the number of separate factors in making a decision is a mere handful. This latter observation reflects a second philosophical bias mirrored in Reference 7 relative to overly complex algorithms.

Balancing algorithms in action. We now describe an experiment designed to show how WLM allocates work across clustered systems. The experiment was run on three IBM 9672 Parallel Enterprise Servers. Each 9672 had six CPUs. At the start of the experiment, CPU-intensive jobs were started on two of the systems. Next, 1000 simulated users were logged on

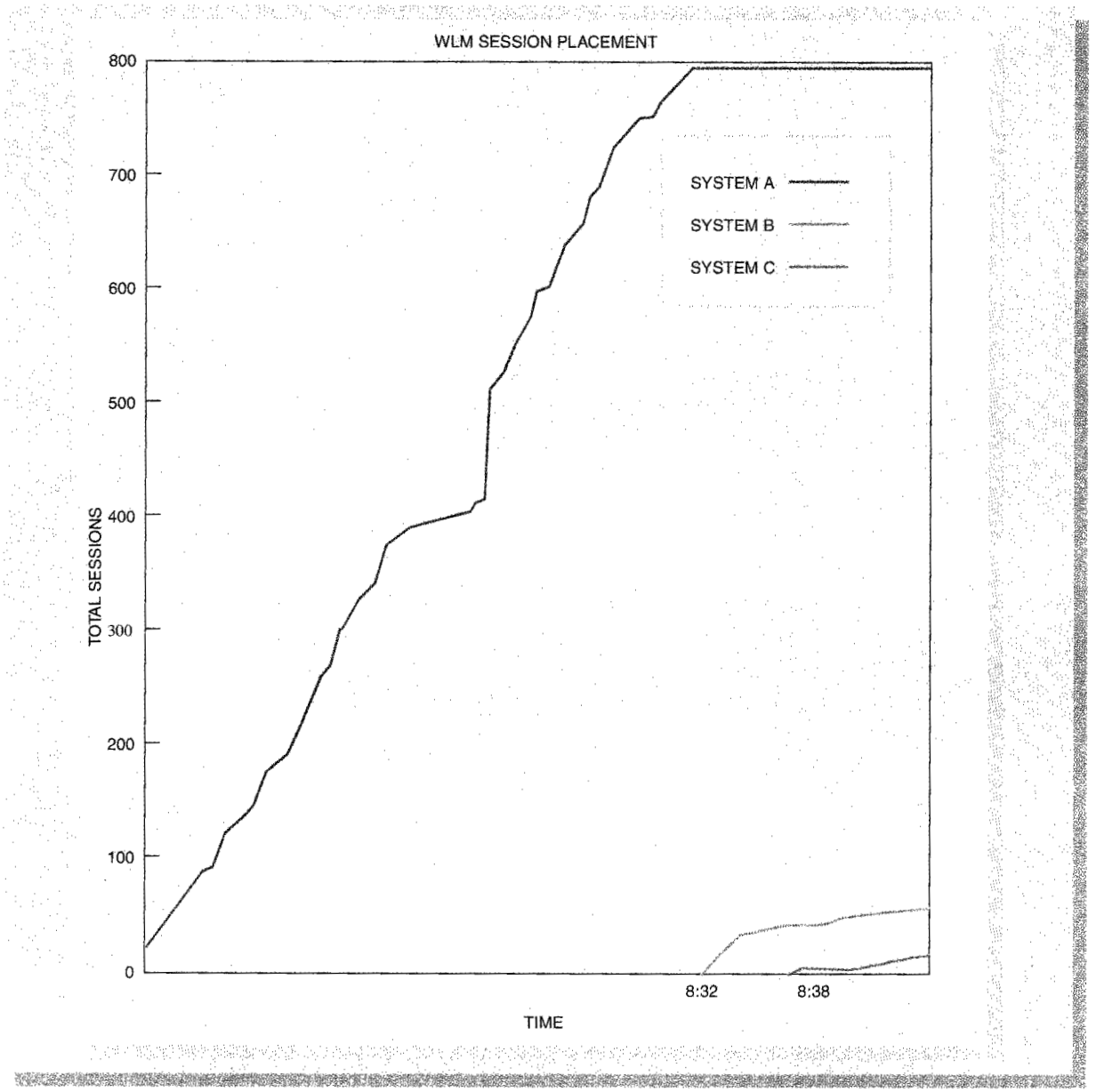
Figure 15 Idle capacity plot



to the parallel system. We discuss how WLM distributes these new users to achieve workload balancing across the three systems.

In Figure 15, Systems A, B, and C are receiving work to establish OS/390 userid log-on sessions. System A has much more idle capacity than System B or System C. The CPU-intensive jobs running on B and C are absorbing the capacities of B and C. Figure 16 shows that as new userids request to log on, WLM places them on System A until the idle capacity of System A falls below that of Systems B and C. WLM then places new log-on sessions on System B because at time 8:32, System B has more idle capacity than

Figure 16 Total sessions placement plot



A or C. At time 8:38, the new users on System B have reduced the idle capacity of B, and the graph in Figure 15 shows an amount of idle capacity equal to System C. WLM now directs new log-on sessions to System C.

This experiment shows the effect of capacity considerations on the balancing algorithms of WLM and the responsiveness of the algorithms. Within microseconds, new log-on sessions are sent to the systems with better capacity. A human operator could not be as responsive or vigilant 24 hours a day, every day.

Cooperating products on WLM environment. MVS/ESA SP5.1.0 provides the initial support to allow work managers and resource managers to cooper-

ate with WLM to surface: completions of business units of work, the associated delays as viewed by the subsystem product, and which address spaces are involved in processing each work request. The following products utilize these new interfaces to make this possible:

1. CICS V4.1
2. CP/SM V1.0
3. IMS V5.1 (transaction manager and database)
4. Resource Measurement Facility (RMF* V5.1)

The following traditional work environments are also supported for goal definition and management in MVS/ESA SP5.1.0:

1. APPC/MVS scheduler (ASCH)—Advanced Program-to-Program Communication initiators for client/server environments
2. Job Entry Subsystem (JES)—batch work
3. OpenEdition* MVS (OMVS)—OpenEdition work utilizing UNIX and other programming semantics
4. Started Task Control (STC)—started tasks
5. TSO—interactive environment

MVS/ESA SP5.2.0 provides the ability to balance sessions via generic resources or balance work requests via sysplex routing. It also supports splitting of work requests either on a single OS/390 image or on multiple images. Other support allows products to communicate the presence of user requests to WLM for individual management within a single server as in the following:

1. CICS V4.1—generic resource
2. DB2 V4.1—distributed DB2, single-CEC (central electronics complex) parallelism, sysplex routing, generic resource
3. DB2 V4.2—multi-CEC parallelism, TCP/IP routing
4. VTAM V4.2, V4.3—generic resource
5. RMF V5.2

OS/390 R3 provides the ability to dynamically control the number of server address spaces on an application environment level based on goal satisfaction of the work requests queuing work to these servers. Related services allow products to dynamically route work requests to the optimal server address space. Additional support was added to allow server address spaces to separately identify multiple business units of work for classification and WLM goal management. The following products utilize these and related services:

1. Local Area Network File Server (LFS)
2. System Object Model (SOM*)
3. Internet Connection Secure Server (ICSS)
4. DB2 V4.2—WLM-managed stored procedures
5. RMF V5.3
6. TSO generic resource

Conclusion

The systems management burden upon installations supporting large-scale enterprise-wide computing is growing at an alarming rate. The cost of configuring system hardware and software, application programs, and network configurations escalates with the introduction of new servers, new networking technologies, and new application program development technologies. The ability to absorb new technologies is vital to those wishing to exploit the possibilities offered—to gain business advantage. This paper described a technology to assist such exploitation. In addition, three significant dimensions are worthy of discussion: a strong supporting philosophy, an initial product implementation, and a dream of what the future may hold.

The foundation for workload management capabilities described in this paper is a crisp definition—to operating system software—of the underlying business rationale for employing a computer system. Knowledge of goals for work and the business importance of that work provides a well-informed basis for operating system software to take direct actions having positive business value to the purchaser of that system. The underlying philosophy is that the system should manage itself, using all available means, toward those business goals, with no additional requirement for human intervention. This workload management philosophy is in sharp contrast with other efforts within the industry, efforts focused on delivery of tools to aid information technology professionals with the optimization of critical resources. Low-level, detailed focus on individual application servers and the resources consumed by the servers enables one to quickly lose sight of what is really important: satisfaction of business needs.

Much information can be found in the literature describing theoretical problems and solutions, evaluations, and comparisons of alternative system structures, including descriptions of areas for valuable future research. The content of this paper has been limited to subjects embodied in software product implementations being used in commercial data pro-

cessing environments. The functional capabilities required by those users are much greater than the topics reviewed in this paper; detailed descriptions of many capabilities were omitted to limit the length of this paper. Some of the additional functions are:

- The ability to dynamically instantiate application server processes based upon trade-offs between demand, the ability to satisfy goals for the work requests that the servers serve, and the availability of system resources. This function eliminates the need for preconfigured application servers while preventing potential problems caused by excessive or inadequate application serving capabilities.
- The ability to manage a given work request as a single unit of work, even though the work request requires the services of more than one application server. This function allows the system to manage more closely to the requesting client view of a response time and eliminates the need to set individual performance goals for each application server.
- The ability to set an overall resource consumption limit for a set of service classes. This function provides a mechanism to guarantee availability of an amount of capacity for one or more critical workloads.
- The ability to temporarily promote the "importance" of an individual work request when that request has acquired a serially reusable resource needed by more important work. This function aids nondisruptive management of resource contention.
- The ability to generate detailed descriptive information showing resource consumption information and resource delay information in the context of the various service class goals in effect. This function explains why the actual results were achieved and provides a basis for capacity planning and system performance modeling tools.

Beyond the capabilities of the current System/390 workload management implementation lie future opportunities for expanding the quality of heuristic decision-making and the scope of resources being controlled. These opportunities represent much more than mere enhancements to the OS/390 operating system—they represent the implementation of a dream where information technology resources adapt themselves to satisfy business objectives without requiring human guidance beyond definition of those business objectives. As long as the need exists for detailed, low-level performance control parameters, the possibility of incorrect or inappropriate definitions will exist. Is the realization of such a dream possible? Maybe. It is easy to envision numerous functional extensions that fit neatly into the framework of the existing OS/390 workload management implementation. Some of these extensions are:

- To dynamically manage the size of memory-resident buffer pools, balancing the availability of memory against the value received from avoiding physical I/O activity and the cost of managing the buffer pools
- To more closely coordinate the scheduling of interrelated "networks" of jobs, managing an entire set of jobs toward a specific time-of-day completion requirement
- To retain longer-term histories of resource utilization patterns, so that repeatable peaks and valleys in workload demands can be anticipated

Although these problems appear to be solvable, other more complex problems remain. Complete goal-oriented management of an enterprise would require end-to-end management of distributed, heterogeneous operating systems and the network interconnection mechanisms that join these systems. Although the infrastructure exists within OS/390, varying amounts of capabilities exist on other operating system platforms. An end-to-end perspective implies the inclusion of:

- End-user workstations having little or no programming capabilities
- Network gateways and intermediate servers that currently have no understanding of the "work" that they process
- Operating system platforms having primitive resource management controls

Focusing on the operating system platforms alone does not address the full set of requirements, since the interconnection mechanisms must also manage the available bandwidth toward the needs of the work requests associated with data being transported.

These longer-term desires represent significant technical challenges.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of NCR Corporation, X/Open Co. Ltd., or Digital Equipment Corporation.

Cited references and notes

1. S. Zhou, J. Wang, X. Zheng, and P. Delisle, *Utopia: A Load Sharing Facility for Large Heterogeneous Distributed Computer Systems*, Technical Report CSRI-257, University of Toronto, Toronto (April 1992).
2. *IBM LoadLeveler Administration Guide, Release 2.1*, SH26-7220-03, IBM Corporation (August 1995); available through IBM branch offices.
3. D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems*, Research Report, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 (October 1994).
4. P. Yu, "On Robust Transaction Routing and Load Sharing," *ACM Transactions on DB Systems* **16,** No. 3, 476–512 (September 1991).
5. G. Goldberg and P. H. Smith III, *The VM/ESA Systems Handbook*, J. Ranade, Editor, IBM Series, McGraw-Hill, Inc., New York (1993).
6. T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, "SP2 System Architecture," *IBM Systems Journal* **34,** No. 2, 152–184 (1995).
7. P. S. Yu, "Performance Analysis of Affinity Clustering on Transaction Processing Coupling Architecture," *IEEE Transactions on Knowledge and Data Engineering* **6,** No. 5, 764–786 (October 1994).
8. E. Berkel and P. Enrico, "Effective Use of MVS Workload Manager Controls," *Computer Measurement Group Winter Transactions* (1995), pp. 21–38.
9. D. Eager, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering* **SE-12,** No. 5, 662–675 (May 1986).
10. J. M. Nick, J.-Y. Chung, and N. S. Bowen, "Overview of IBM System/390 Parallel Sysplex—A Commercial Parallel Processing System," *10th International Parallel Processing Symposium* (April 1996), pp. 488–495.
11. Service class defining the client's performance goal.
12. Count of observations work classified to this service class using the processor.
13. A multiprogramming level (MPL) slot represents one address space. The multiprogramming level is the number of address spaces resident in central storage at any given time.
14. An I/O cluster is a unique set of devices that are shared by address spaces in the same set of period(s). Changing the I/O priority of one period in an I/O cluster could affect the I/O priority of any of the periods that are part of the same I/O cluster.

Accepted for publication January 27, 1997.

Jeffrey Aman *IBM S/390 Division, 522 South Road, Poughkeepsie, New York 12601 (electronic mail: jdaman@vnet.ibm.com).* Mr. Aman is a Senior Technical Staff Member. He has been in the MVS software design and development area since 1981, involved in many programming projects as a developer, tester, development team leader, and designer. For the last six years he has been the technical leader of the MVS workload manager project, working with design and development of the MVS base control program components and design of exploitation by the IMS, CICS, DB2, VTAM, and RMF products.

Catherine K. Eilert *IBM S/390 Division, 522 South Road, Poughkeepsie, New York 12601 (electronic mail: eilert@vnet.ibm.com).* Mrs. Eilert is a senior software engineer in the S/390 Performance Design department. She was the team leader for the WLM algorithms project and received an IBM Outstanding Technical Achievement Award for that work. Mrs. Eilert holds four issued patents related to performance management algorithms. She received a B.S. degree in industrial engineering and an M.S. degree in computer science, both from the Illinois Institute of Technology in Chicago. Her current interests are performance management algorithms and designing software for performance and scale.

David Emmes *IBM S/390 Division, 522 South Road, Poughkeepsie, New York 12601 (electronic mail: demmes@vnet.ibm.com).* Mr. Emmes joined IBM in 1978 and has worked in MVS development since then. During that time, he worked for seven years on processor storage management for MVS/XA™ and MVS/ESA, five years on multisystem coupled communication for MVS/SP™ V4, and six years in workload management for MVS/SP V5 and OS/390 R3.

Peter Yocom *IBM S/390 Division, 522 South Road, Poughkeepsie, New York 12601 (electronic mail: yocom@vnet.ibm.com).* Mr. Yocom is currently in workload manager design and development and worked for 10 years in MVS development.

Donna Dillenberger *IBM Research Division, Thomas J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, New York 10532 (electronic mail: engd@watson.ibm.com.)* Ms. Dillenberger joined the former IBM Data Systems Division in 1988, and the Research Division in 1994. She has worked on future hardware simulations, workload manager, objects, and the Web.

Reprint Order No. G321-5643.