Scheduling Under Uncertainty: Planning for the Ubiquitous Grid

Neal Sample Pedram Keyani Gio Wiederhold

Computer Science Department
Stanford University, Stanford CA 94305
{nsample, pkeyani, gio}@cs.stanford.edu

Abstract. Computational Grid projects are ushering in an environment where clients make use of resources and services that are far too expensive for single clients to manage or maintain. Clients compose a megaprogram with services offered by outside organizations. However, the benefits of this paradigm come with a loss of control over job execution with added uncertainty about job completion. Current techniques for scheduling distributed services do not simultaneously account for autonomous service providers whose performance, reliability, and cost are not controlled by the service user. We propose an approach to scheduling that compensates for this uncertainty. Our approach builds initial schedules based on cost estimates from service providers and during program execution monitors job progress to determine if future deadlines will be met. This approach enables early hazard detection and facilitates schedule repairs to compensate for delays.

1 Introduction

Advances in the speed and reliability of computer networks in combination with distribution protocols (such as CORBA and Java RMI) allow clients to abstract away heterogeneities in the network, platform, language, etc., and make use of distributed services and resources that were previously unavailable. Remote services and resources can be utilized as if they were locally available. There are still complications that arise from geographic distance, security concerns, service autonomy, and compensation. In order to complete the abstraction to transparently use remote services and resources, it is necessary to have a mechanism to deal with the uncertainties introduced by scheduling services not under local control.

The development of Grid computing has enabled a model where organizations can develop services and charge a fee for their use to clients. Fee-based computing models are gaining success in both cooperative and commercial computing environments [3,5,7,8,11]. In these grids, service providers charge fees or trade resources for the use of their service. The value of the service offered is a combination of the software itself and the execution of the software. This is an attractive opportunity for service providers because they can amortize their development and maintenance costs and share these expenses with clients, while protecting their proprietary interests in the

service. Also, under the CHAIMS model, service updates can be performed in a central location and not need to be propagated to end-users.

Clients also gain from this model in that they can access services that they don't have to develop or maintain [12]. Many clients do not have the resources to develop sophisticated software or purchase the high-end machinery necessary to accomplish their tasks. For example, suppose a small university's genomic research lab had a digitized DNA sequence from which they wanted to isolate a certain gene. Instead of developing the necessary software "in house" they hire a service provider that has the computational power and appropriate genomic software to analyze their data and give them the result they seek. Contracting for the service has the same result as purchasing the computational hardware and proprietary software, but at a fraction of the cost. In an open market, valuable software services may have multiple service providers competing for the same pool of customers. A natural pricing structure would evolve based on the time to completion and the surety of the service providers. (Surety is the probability that a job will finish execution within a deadline window forecast by the service provider.) Quick executing services with a high surety would of course be more valuable than the same services that have longer running times or a low surety. A customer's choice of service providers would depend on what value they place on time, cost, and surety, simultaneously. Until now, schedulers have treated the remote service problem as a multivariate optimization involving only two variables: cost and time [2,4,13]. We extend the worldview by accounting for the uncertainty introduced when services are not under the client's control.

Access to an array of services provides many opportunities for service composition. The ability to compose services is an especially powerful tool for multi-disciplinary projects where no single client has expertise in all sub-problem areas [12]. By allowing for composition of existing modules, researchers can devote less effort to software development and more time to central research questions. But there is a pitfall: distributed services are not under the control of the client. This means that estimates for job completion time may be inaccurate and clients cannot control resource allocation to recover from hazards. An inaccurate estimate in the completion time of single service is undesirable; in a program comprised of multiple services this can quickly become untenable.

The main research problem we address in this paper is the decreased level of scheduling surety that comes from composing a program from multiple distributed services [12,14]. By making programs composed of distributed services more reliable, these compositions become an increasingly viable solution for a wide range of problems, and become an appropriate solution for a larger class of clients.

At a finer resolution, the goal of this project is to take a program composed of multiple services and complete it within a soft deadline and cost budget, while guaranteeing a client-specified minimum level of surety. The scheduling process begins with the selection of an initial schedule based on service provider estimates for completion time and fee [35]. The initial schedule is driven by dependencies (data, control, etc.) between service invocations and estimates from service providers. At runtime, job monitoring detects misbehaving services that can jeopardize the completion of the entire program. During the monitoring phase, surety is recalculated whenever progress is made (or not made, but time has advanced). If surety drops

below the minimum threshold determined by the client, the scheduler takes action to recover from the delay and increase surety to an acceptable level. Any measures taken require finding alternative schedules for the remainder of the program that restore surety without exceeding the program's budget. Monitoring coupled with reactive rescheduling is key to providing clients with the surety of distributed job completion, as they would expect from a program running solely on local resources.

Systems such as CHAIMS (Compiling High-level Access Interfaces for Multi-site Software) allow clients to abstract away heterogeneity and service autonomy while simultaneously compensating for pitfalls associated with both [15]. We focus on scheduling with CHAIMS because its preferred development language (CLAM – Composition Language for Autonomous Megamodules) provides key language primitives that enable dynamic scheduling with the possibility of recovery from hazards. CLAM contains a primitive to get estimates of the job completion time and cost from a service provider (ESTIMATE), and a primitive to examine job progress from a service provider (EXAMINE)[16]. These two capabilities used in concert allow for scheduling a program with more confidence in execution time. Other coordination languages such as MANIFOLD are appropriate for this type of composition, but lack the EXAMINE and ESTIMATE primitives found in CLAM [17,18].

The current supported runtime system for CHAIMS is known CPAM (CHAIMS Protocol for Autonomous Megamodules) [19]. The protocol removes the barriers imposed by different programming languages and distribution protocols, while providing support for the scheduling primitives in CLAM. Programs written in CLAM are known as megaprograms, though the class of programs is known by myriad names in other scheduling literature (ensembles, compositions, grid programs, workflows, etc.) [5,19,22,23]. Within CHAIMS, megaprograms are composed from megamodules. Megamodules are what we have referred to simply as services; they are assumed to possibly come from multiple programming languages, distinct hosts, and have different native distribution protocols [16].

An initial objective of CHAIMS was to simply develop a language and runtime support for the programs composed from distributed modules. The focus of this work is to add a dynamic scheduler to the system that can deal with the issues that arise in an unreliable environment. We build on their prior efforts because the language and runtime support overlap well with the requirements of runtime testing and surety monitoring. However, our work is broadly applicable to any system where estimates may be gathered a priori and where clients may monitor runtime progress.

Current distributed service scheduling research has not presented a complete solution that incorporates uncertainty. Most distributed computation schedules assume a cooperative environment where delays are rare, and that initial estimates come from oracles. The foundation of this work is that distributed systems (in practice) are rife with uncertainty that affects the reliability of schedules generated a priori. Section 2 discusses the characteristics of autonomous service providers and the attributes central to the scheduling task. Section 3 gives a brief description of CHAIMS and how it enables composition and coordination of distributed services. Section 4 explains the scheduling techniques and job monitoring that we advocate. Section 5 covers related work and explains how our techniques may be leveraged in distributed architectures

other than CHAIMS. This research has opened further questions, detailed in section 6.

2 Autonomous Service Providers

The Internet has made distributed services a reality and opened up a completely new scheduling problem area to explore [1,4,21,24,25]. Without careful consideration, as computations are moved farther and farther from client control, it is increasingly likely that hazards will slow or halt progress. These hazards may arise from hardware or software failures, to power outages, to resource mismanagement by a service provider. Additionally, in competitive markets, service providers try to maximize profits. A greedy service provider could mistakenly take on more jobs than it can handle and delay the finishing time of all jobs. Alternately, an unscrupulous service provider might stop the execution of a low-paying job, however unfair it may seem, for more lucrative jobs that arise. Running into delays for a single service can be costly, but when programs are composed from multiple distributed services, delays in one service can have an undesirable cascade effect that destroys scheduling commitments for the entire client program. One aid to avoid such problems comes in the form of contracts [11, 34]. In the simplest contracts, clients use initial estimates of job completion time to bind service providers. This still does not guarantee that service providers will be able to meet the deadline of their contract. As such, it is also necessary to monitor job progress during execution to determine if the contract will be met (and to react swiftly and appropriately to recover if it is not).

In this uncertain environment it is necessary to leverage contracts to motivate clients and service providers to meet their mutual obligations. At its core, the contract enables two parties who do not trust each other to enter into a mutually beneficial agreement. While contracts are a tool to promote accountability, they do not enforce it. In this paper we do not discuss contracts negotiation or enforcement, as they are implementation details that each distribution model must decide on. However, more information on contracts and negotiations within distributed systems can be found in [2,20,23]. We expect contracts to consider:

- *Cost* – what a service provider will charge for the service.
- *Completion time* - the estimated length of time to complete the job.
- *Variance* - the amount of time before or after the completion time that the job may finish. (It is assumed no service provider can be completely accurate in job estimation.) Variance may be presented symmetrically or asymmetrically. In this paper, we assume symmetric variance in the examples, but provide equations to handle asymmetric variance.
- *Late fee* – a credit the service provider returns to the client per unit of time that the job is not finished after completion time plus variance.
- *Cancellation fee* - a set amount that the service provider will return to the client if the client breaks the contract. This value may be zero, but is of utility to both client and service provider.
- *Reservation* – an amount the client pays after negotiating the contract to hold resources until it exercises the contract and invokes the service. This

reservation fee guarantees that the client will be given access to the service provider's resources, and confidence that the client can have some control over job start time. Reservations are a complicated issue themselves, and are further discussed in [11,23]. We assume "American options", which allow a client to start a service at any point until expiry.

This distributed computational model for accessing services closely parallels traditional economical models. Significant detail on how grid computing relates to (and can leverage) various economic mechanisms can also be found in [2,4,11,23]. We do not make assumptions about the trading or cash economies that will lubricate the grid; we simply examine the general considerations that economic factors place on scheduling under uncertainty.

3 CHAIMS

Technological advancements and social change have made grid computing a feasible option for computation. We have chosen the CHAIMS platform as a test bed for our investigations for the reasons mentioned in section 1. Specifically, we leverage its compositional programming language (CLAM) and the runtime support system (CPAM) to enable composition and coordination of distributed services. In the future we expect to plug our scheduler into other compositional and grid platforms. In this section, we give more detail about CHAIMS.

3.1 CLAM

CLAM is a declarative language intended for large-scale module composition that lacks constructs for complicated computation [16]. Motivating the absence of computational primitives in CLAM was a desire to create a simple language for non-programmer domain experts to accomplish their desired tasks [19]. The premise rests on the idea that there is a collection of service providers who will offer some services that may be used with other distributed services, or perhaps with modules locally controlled by domain experts.

CLAM is designed for a large-scale environment where parallelism is important. Unlike many languages for distributed or parallel computing (e.g., HPF (High Performance Fortran) [26]), CLAM does not specify what resources a service invocation uses. This permits service instances to be scheduled identically, regardless of the capabilities and resources of the service's backing system. Breaking free from resource specification enables freedom in choosing service providers at runtime, rather than compile time, thus selecting providers based on costs at runtime. This late binding time is similar to the tactic introduced in [11,13].

While CLAM is simply a declarative compositional language, it makes heavy use of the coordination capabilities supported by CPAM to achieve good performance. CPAM enables asynchronous service invocation and parallel execution. The key facilities that CPAM offers are the ability to get estimates on job completion and to examine job progress during service execution. These two CPAM capabilities are mirrored in the ESTIMATE and EXAMINE primitives of CLAM. Further discussion

in Section 4 will show how ESTIMATE and EXAMINE provide the capabilities for building an intelligent scheduler. Note that enabling these primitives in any distributed framework provides a generic framework for scheduling under uncertainty.

3.2 CPAM

CPAM is a generic high-level protocol for remote service invocation. CPAM compensates for language and platform differences by representing data as ASN.1 structures [19]. Data is transmitted from clients to services (and between services) opaquely, with no alteration possible during transit. Services must be wrapped to unpack data type locally. CPAM has been implemented above many transport protocols such as DCOM, Java RMI and CORBA. There is also a "native" transport mechanism for CPAM.

CPAM breaks service invocation into a multi-step process to enable asynchrony and parallelism. The steps of invocation are SETUP, SETPARAM (parameter setting), ESTIMATE (garner cost estimates), INVOKE, EXAMINE (monitor progress), EXTRACT and TERMINATE [16]. An extended discussion of each is available in [16]. These functions are exposed to programmers in CLAM. However, with an automated scheduler, ESTIMATE and EXAMINE are not required in the language, just as part of the runtime. ESTIMATE and EXAMINE are necessary for scheduling because they allow the construction of an initial schedule and allow for monitoring during runtime. Calling ESTIMATE from a service provider returns an estimate of job completion time and a variance related to that value. Because the runtime of a service is often dependent on its input, SETPARAM is used to give the service provider enough information to make an informed estimate. Once a service is executing, EXAMINE can be called to view the progress of the job. The ultimate goal is to deprecate the EXAMINE and EXTRACT primitive in CLAM by leveraging them in a scheduler built directly on top of CPAM. Runtime support provided by mature grid implementations such as Legion are also appropriate [27,28].

4 Scheduling in an Uncertain Environment

In order to deal with an uncertain environment it is necessary to consider surety when scheduling. A surety threshold is set before a program is started, and is monitored throughout the execution lifetime of the program. When progress information indicates that the surety threshold has been breached, dynamic repair and rescheduling operations are triggered. As mentioned previously, the scheduler will make use of the ESTIMATE and EXAMINE capabilities to build initial schedules and to monitor progress.

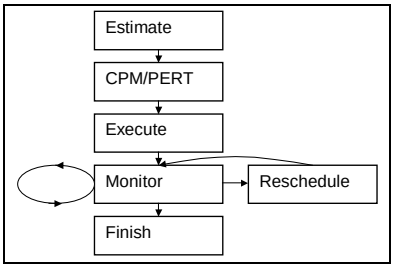

Fig. 1. The scheduling process

Before scheduling can start, the client determines a budget for the program. A budget is made up of (1) the deadline that the program must be finished by, (2) the amount of consideration (money, credits, bartered resources, etc. depending on the economic model) that can be spent on the program execution, and (3) the minimum level of surety that must be met by the scheduler. Individual clients determine the amount of time and consideration available for a specific program's execution. The surety is a limit on the risk the client will tolerate in meeting their budget. Once the constraints of the budget are determined, the client can select to optimize or balance these three budgetary concerns (time, cost, surety). Figure 1 shows a simplified view of the scheduling process; steps not central to this work are omitted. We will discuss each of these steps in detail. In this section, we present an overview of the process steps.

First, estimates are collected for each service from potentially many service providers and used in the program to build possible schedules. In our current naïve implementation, we exhaustively enumerate all schedules and select one from the pool of best choices. Before discussing which schedules are "best", we will clarify our underlying schedule evaluation techniques.

Once a candidate schedule is created, the shortest expected running time of that schedule can be determined using CPM (Critical Path Method)[29]. With this information it is trivial to test whether a schedule meets the minimum time and budget criteria, however nothing is known at this point about surety. The longest path (in terms of expected execution time) in the program determines the runtime of the program. This longest path is called the "critical path" because any delay along the critical path will affect the running time of the entire program. To determine surety it is necessary to extend the CPM analysis to a probabilistic PERT (Program Evaluation and Review Techniques) analysis [29,30,31]. PERT extends CPM by accounting for the uncertainty in each estimated service duration to compute the surety of the entire program. We will discuss our use of PERT in significant detail in the next section.

Once a schedule is selected and contracts are finalized, the scheduler may invoke any ready services in the program. As services execute, their progress is monitored to ensure that completion times are met; if the overall surety of program completion drops below the predetermined threshold, the scheduler begins the repair and reschedule phase. In the repair phase there are many options. New service providers may be found to replace the service module that is delaying overall progress. Or other services along the critical path may be substituted for alternative services that have

shorter runtimes (though at an increased cost). Once repair and rescheduling is complete, the scheduler returns to monitoring the execution.

4.1 Simple Planning

The first step in scheduling is program analysis to discover any dependencies among component services and construct a dependency graph for the workflow. The very simple program in Figure 2 shows implicit data dependencies between services. For instance, `service3` takes `A` and `B` as input. `A` and `B` are outputs of service1 and `service2`, respectively. These dependencies are mapped into the workflow of Figure 3 where nodes represent services and dependencies are shown as arcs between nodes with the arrow pointing to the dependent node. These workflows consist of paths that are created by dependencies between nodes. Once the dependencies are mapped, the scheduler requests bids from service providers in order to fill in cost values for the proposed schedule.

The scheduler contacts a repository or directory service that returns a list of service providers that perform a specific service. Based on this list, the scheduler contacts the service providers and requests bids. The *bid request* is based on the service needed, the expected start time for the service, and information about the size and complexity of the input parameters to the service. For some services, the inputs cannot be known at runtime because they are the outputs of other services; in these cases information about the size and complexity of parameters is currently based on heuristics that the Service provider uses to make best estimates. Service providers collate this information and calculate a possible bid. The client receives a collection of bids and will either accept one or more bids for the schedule, or attempt to renegotiate with the service providers [2,20,23]. The deciding factor in which bids are accepted is based on the Pareto optimality [32] of the "best" schedules. For a bid to be Pareto optimal, there can be no other bid with an absolute advantage in terms of price, time and surety. The Pareto curve in this case is weighted by the optimization strategy presented by the client, for instance in soft real-time applications surety will have a high weighting. All decisions are based simultaneously on cost, time, and surety.

Using CPM gives the scheduler the critical path and the expected runtime of the

```
// begin program
A = service1();
B = service2();
C = service3(A,B);
D = service4(C);
E = service5(C);
// end of program
```

Fig. 2. Sample program

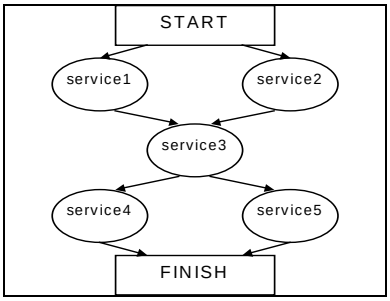

Fig. 3. Dependency graph of sample program

program. This information allows the schedule to select candidate schedules that meet the client's budget and to further optimize and refine the schedule. For instance, CPM indicates positions of "slack" in the schedule, where cheaper, longer running processes may be used because they're not along the critical path [29]. By choosing slower services in these non-critical paths, the scheduler can possibly decrease the overall cost of the program, thus saving resources that may be necessary for repairs at a later time. The total price cost for all services executed plus the cost of any reservations not kept is the total cost of the program.

The PERT method extends CPM to account for uncertainty of individual service completion times and determines the probability of completing a complete program by an expected time [29]. PERT analysis forms the basis for our rescheduling decisions. To perform the initial analysis, the scheduler uses three time estimates for each service: most likely(m), optimistic(a) and pessimistic(b) completion times. "Optimistic" and "pessimistic" times are derived from the expected time coupled with the variance. With this information, the *expected duration* e_i of a single service can be determined by a weighted combination of the most likely duration m_i and the midpoint of the distribution $\frac{(a_i + b_i)}{2}$ for each service *i*:

$$e_i = \frac{2m_i + \frac{a_i + b_i}{2}}{3} \quad \textbf{(1)}$$

There is a spread of about 6 standard deviations from a_i to b_i thus:

$$\sigma_i = \frac{b_i - a_i}{6} \quad \textbf{(2)}$$

Activity durations are independent; hence the sum of the independent random expected functions e_i is normally distributed. From the expected completion time and standard deviation of each service on the critical path, we construct the expected completion time and standard deviation of the entire program as:

$$\bar{e} = \sum e_i \text{ and } \sigma_{program} = \sqrt{\sum \sigma_i^2} \quad \textbf{(3)}$$

for all services *i* on the critical path. With this we can calculate the probability that the program completion time *T* is less than the deadline of time *t*. This $prob\,(T \le t)$ represents the surety level of the program completing execution by its deadline, *t*. We specify the completion time as:

$$t = \bar{e} + x * \sigma_{program} \quad \text{giving us} \quad x = \frac{t - \bar{e}}{\sigma_{program}} \quad \textbf{(4)}$$

which is used to express the surety of the program as:

$$prob\,(T \le t) = \left[\frac{T - \bar{e}}{\sigma_{program}} \le \frac{t - \bar{e}}{\sigma_{program}} \right] \quad (5)$$

This is the same as the probability of a random variable from N(0, 1) distribution being less than or equal to $\frac{t - \bar{e}}{\sigma_{program}}$.

Using CPM and PERT, the scheduler can evaluate the bids that form the final schedules. What follows now is an example of the complete repair and scheduling activity.

In this example, Table 1 shows a set of bids produced for the sample program in Figure 2. In some cases, multiple service providers bid for a service, giving the scheduler some flexibility. In the case of `service3`, only one service provider has replied with a bid, thus it will have to be used.

Table 1. Bids received for each service

Service	Cost	Time
service1	8	8 +/- 2
	9	10 +/- 1
	11	6 +/- 0
service2	10	5 +/- 2
	12	4 +/- 1
service3	5	6 +/- 0
service4	15	2 +/- 0
	10	4 +/- 2
service5	20	1 +/- 0
	10	2 +/- 2
	5	3 +/- 1

After all bids have been received, the scheduler searches for an optimal schedule,

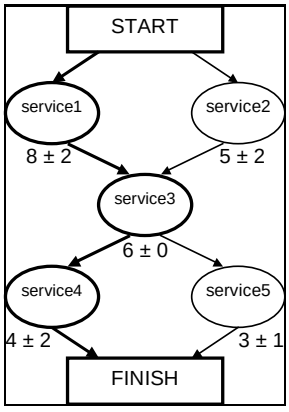

Fig. 4. Sample Schedule

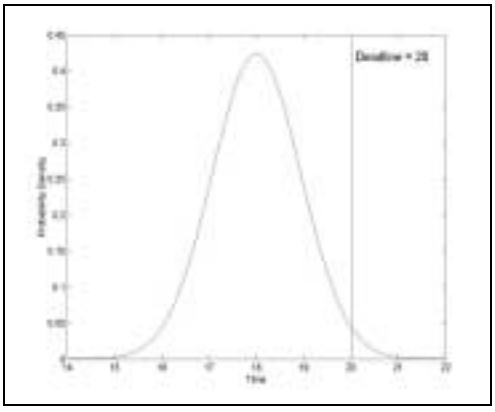

Fig. 5. Probability distribution of original program schedule

based on the client's budget. Assume the client has given a deadline of 20 units of time, a budget of ¥65 and a minimum surety requirement of 90%. (We use the "¥" symbol simply to distinguish the cost numbers from the time numbers, and make no implication about any particular specie or economic model.) The bids from table 1 are used to construct the schedule in Figure 4. In Figure 4, the critical path to consist of nodes `service1`, `service3`, and `service4`. By application of CPM, this schedule has an expected finishing time of 18, an earliest finishing time of 14, and latest finishing time of 22 assuming no hazards. The total budget estimated for this program is ¥38, allowing a reserve of ¥27, which can be used to repair the schedule in case of delay or hazard. Figure 5 shows the probability distribution of this program's completion time. The surety for the program is determined via PERT to be 98.31%, an acceptable level (≥90%). Please note that these values for cost, time, and surety are from an *a priori* analysis based solely on service provider estimates. In the next section, we explain surety based monitoring and repair strategy.

4.2 Monitoring and Repairing Schedules

Once the initial planning stage is complete and contracts have been drawn, execution starts. For our running example, we assign this start time a convenient value of time=0. The schedule for our example (shown in Figure 4) is very "tight" in terms of cost. There are alternative assignments of service instances that would give a lower overall cost. This trades off with overall runtime of the schedule to some extent, but the tradeoff is considered acceptable because the schedule does not fall below the surety threshold.

Of course delays along the critical path are likely to be the most damaging since they extend the run time of the entire program. Constant monitoring is required to ensure that single delays do not affect the entire program. However, delays not in the critical path can also impact surety. We illustrate this case next.

At time=0, `service1` and `service2` begin execution. Imagine that we monitor progress at each time integer interval. At time=1, time=2, and time=3, the scheduler observes no anomalies. However, at time=4, a hazard is detected.

According to the expected schedule, at time=4, `service2` should be 80% completed. Imagine that the scheduler observes that `service2` is only 50% complete. Based on this information, the scheduler projects that the `service2` will complete at time=11, thus altering the critical path to include `service2` instead of `service1`. This potential delay changes the expected running time of the program, which subsequently lowers the surety of the overall execution to 14.44%, which is an unacceptable level (< 90%).

To counter this delay, the scheduler first contacts all service providers to get new bids. (It is interesting to note that in our model as system conditions change, initial estimates become moot. This is especially true when scheduling long-running services.) The scheduler determines that the most effective strategy is to accept a bid to attempt to finish `service2` at an earlier time. At time=4, a bid for an instance of service2 that will cost ¥10 and complete in 5 units of time (with a variance of 2 (5±2)) is found and accepted. Immediately, this second service provider for service2 begins work in parallel with the delayed instance. This repair strategy has increased the surety

of the complete program, but we now expect the program to finish somewhere between time=15 and time=23 with a mean expectation of time=19. This repair increases the surety to 85.56%, and reduces the reserve budget to ¥17. A surety of 85.56% is below the threshold. This schedule requires further repair.

After the delay is caught and an alternative found, the surety remains below the 90% threshold. To further increase surety, it is necessary to select a node along the critical path and either find alternate service providers that can perform the same service in less time, or contract with multiple service providers to execute the same service in parallel, thus increasing the probability that at least one of them will deliver results in time. The method used to increase surety depends on how much budget is left over, and if alternative service providers can be found with the required performance capabilities. In this example, assume that we discover another service provider that offers `service4` for ¥10 with a completion time of 3, and variance of 2. Using this service provider increases overall surety to 95.83%, which is above the threshold for this execution.

Surety represents the risk of a client program not meeting its deadline that the client will accept. Monitoring job execution at runtime allows our CHAIMS scheduler to compare current surety to the limit established by the client. Falling below the surety threshold triggers the scheduler to repair or reschedule to counteract the effects of hazard. This is achieved primarily by finding alternative or duplicate services increase surety. If surety is set too high (e.g., 100%) or the budget too low, the space of acceptable schedules is radically reduced, and the likelihood of successful schedules decreases as well.

4.3 Initial Results

Initial results are promising but difficult to quantify. The prototype scheduler generates Pareto optimal schedules and selects a schedule based on the client-specified criteria. However, this does not address the central performance question: *how effective is the scheduler at working around delays and hazards?* Our initial evaluation strategy was to simulate network conditions and service providers and allow the scheduler to make its best attempt at scheduling a set of randomly generated programs. Various delays plagued the scheduler during simulation, and we determined a metric for comparison. Using dynamic programming, we simulated all schedules and compared the optimal overall costs and completion times to our scheduler's performance.

This analysis technique does not produce meaningful results. For instance, we tested a program in which its last required service would become permanently unavailable shortly after its execution began. It turns out that the "ideal" schedule is counter-intuitive and would not be selected by any rational scheduler. For instance, if a schedule had only very long running, inexpensive services, then it would fail because it exceeded its time budget long before reaching the final service that was designed to go offline. In those cases, the schedule that had no chance to finish on time wasted less of the client's resources during a futile attempt to solve the problem. This edge case shows the extreme flaw in straightforward quantitative analysis: if a schedule cannot be completed, the worst scheduling policies are rewarded. In this

particular case, a scheduler that constantly returned "failure to find any satisfying schedules" would performance best. We are exploring alternative ways to quantify our results.

5 Related Work

There is significant work in the area of scheduling, though the missing ingredient to move from laboratory conditions to real world systems has been surety. Our approach differs from previous research operating under closed world assumptions where a) *a priori* estimates are provided by infallible oracles, or b) *a priori* estimates of cost will be valid at time=*n*, where *n* is potentially far in the future after the estimate was given. Finally, we see scheduling techniques for time and cost simultaneously, and repair strategies under the oracular estimates assumption, but we have not seen these techniques in conjunction with surety analysis.

5.1 Mariposa

Mariposa is a scheduler for operations over large distributed databases. Entities negotiate with each other for services such as queries, data transmission, and storage [11,13]. Entities act through agents to process their requests. A key assumption is that estimates will be met, without exception. This assumption only holds if a central administrator manages all entities and the administrator ensures that each entity behaves properly. Our scheduler could provide Mariposa the intelligence to handle issues that arise when there is no resource overseer, as expected in a truly distributed environment.

5.2 NOW (Networks of Workstations)

The premise of NoW [33] is that collections of desktops working together have a much better price-performance ratio than mainframes and supercomputers of the same power. Applications considered highly suitable for NoW range from cooperative file caching to parallel computing within a network. Specific projects such as POPCORN [5] seek to take concepts of NoW and extend them to work on the entire Internet. POPCORN is providing programmers with a virtual parallel computer by utilizing processors that participate in this system. POPCORN is based on the notion of a market where buyers and sellers come together and barter for resources.

POPCORN assumes that nodes will fail, and that it is easier to repeat work on backup nodes if a worker misses a deadline. Our scheduling system could account for this uncertainty by monitoring job progress and rapidly migrating computation to alternative nodes if delays are detected.

5.3 Grid Computing

Many computational grid projects are being developed simultaneously (ecogrid, DataGrid, power grid, etc.) [9,10,22]. Contributions to this field are coming from many different projects, each with tailored goals for grid computing. An overarching goal is to allow for resource sharing and services spread over large geographic, political, and economic distances. Projects like the European DataGrid currently focus significant attention on developing a network infrastructure that supports the rapid transport of multi-PetaByte datasets between different locations [1].

Other projects such as Globus provides tools to bridge the gap between heterogeneous grid participants [6]. Globus provides a low level toolkit to handle issues of network communication, authentication and data access. These tools can be used to create high-level services such as intelligent schedulers that can be inserted into a computational grid. Thus, Globus makes it possible to insert our scheduler into a grid infrastructure and provide surety to clients. The resulting increase in schedule dependability will extend the power and use of grid computing.

5.4 ePert and Extensions

This work extends workflow management systems to include time management [30,31]. ePert determines internal deadlines in the workflow and monitors progress at run-time. If deadlines are not met, alternate schedules are chosen. However, these alternative schedules must be known at runtime, and fully available during execution time. ePert extends the PERT method to include with it alternate execution paths a process can take. These extensions allow for some level of pro-active scheduling by detecting time failures and recovering from them. Our scheduling techniques can contribute dynamic components to their work. However, the closed-world assumptions are more likely to hold with workflow schedulers since there is often a strong command and control structure responsible for the workflow (e.g., workflows within a single corporation).

6 Future Work

This project has produced tools that allow us to develop and test advanced scheduling techniques. We are currently working on two parallel tracks. First, we are testing various scheduling heuristics that should improve our repair tactics. Second, we are investigating techniques to quantify performance of our scheduler. We have considered various multivariate analysis evaluation metrics but since they neglect the notion of surety in their cost models, reasonable and rational schedules have demonstrably poor performance compared to the most irrational schedulers in certain cases. We reiterate that the usefulness of our scheduler is not limited to the CHAIMS system, but we are eager to fulfill this claim by inserting our scheduler into a standard grid component.

7 Conclusions

We present a scheduling technique that uses surety to overcome much of the uncertainty naturally present in distributed computing environments. We take a program composed of multiple distributed services and complete it within a client-specified soft deadline by guaranteeing that a minimum level of surety is maintained throughout the program execution. The scheduling tactics demonstrated here make initial schedules, monitor runtime progress, and then repair the schedule if surety drops below a threshold value. This work is broadly applicable to systems whose distributed nature is impacted by uncertainty.

References

1. F. Berman, High Performance Schedulers in Building a Computational Grid, I. Foster and C. Kesselman, editors, Morgan Kaufmann, 1998.
2. R. Buyya, J. Giddy, D. Abramson, "An Evaluation of Economy-based Resource Trading and Scheduling on Computational Power Grids for Parameter Sweep Applications," The Second Workshop on Active Middleware Services (AMS 2000), August 1, 2000.
3. S. Lohr, "I.B.M. Making a Commitment to Next Phase of the Internet" New York Times 2001, http://www.nytimes.com/2001/08/02/technology/02BLUE.html.
4. D. Marinescu, L. Bölöni, R. Hao, and K. Jun, "An Alternative Model for Scheduling on a Computational Grid," Proceedings of ISCIS'98, the Thirteenth International Symposium on Computer and Information Sciences, Antalya, pp. 473-480, IOP Press, 1998.
5. N. Nisan, S. London, O. Regev, and N. Camiel, "Globally distributed computation over the internet-the popcorn project," In Proceedings for the 18th Int'l Conference on Distributed Computing Systems, 1998.
6. I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, SIAM, Lyon, France, Aug. 1996.
7. Juno, "Juno Announces Virtual Supercomputer Project," Juno Press Release, February 1, 2001), http://www.juno.com/corp/news/supercomputer.html.
8. United Devices "Edge Distributed Computing with the MetaProcessor(TM) Platform," White paper, 2001, https://www.ud.com/customers/met.pdf.asp.
9. UDDI, Technical White Paper, September 6, 2000 http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf.
10. C. Kurt, "UDDI Version 2.0 Operator's Specification", UDDI Open Draft Specification 8, June 2001 (Draft K) , http://www.uddi.org/pubs/Operators-V2.00-Open-20010608.pdf
11. M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. "An economic paradigm for query processing and data migration in Mariposa," In Proceedings of the Third International Conference on Parallel and Distributed Information Systems, Austin, TX, September 1994.
12. Carl Bartlett, Neal Sample, and Matt Haines "Pipeline Expansion in Coordinated Applications", 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas, Nevada, June 28 - July 1, 1999.
13. J.Sidell, "Performance of Adaptive Query Processing in the Mariposa Distributed Database Management System," unpublished manuscript, June 1997.
14. W. K. Shih, J. W. S. Liu, and J. Y. Chung. "Algorithms for scheduling imprecise computations with timing constraints," In Proc. IEEE Real-Time Systems Symposium, 1989.
15. G. Wiederhold, P. Wegner, S. Ceri, "Towards Megaprogramming", CACM, Nov.1992.

16. N. Sample, D. Beringer, L. Melloul and G. Wiederhold, "CLAM: Composition Language for Autonomous Megamodules," 3rd Int'l Conference on Coordination Models and Languages, Amsterdam, Apr. 1999.
17. F. Seredynski, P. Bouvry, and F. Arbab, “Parallel and distributed evolutionary computation with Manifold,” In V. Malyshkin, editor, Proceedings of PaCT-97, volume 1277 of Lecture Notes in Computer Science, pages 94--108. Springer-Verlag, September 1997.
18. F. Arbab, “The IWIM Model for Coordination of Concurrent Activities,” First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, April 15-17 1996. (Also appears in LNCS 1061, Springer-Verlag, pp. 3456.)
19. L. Melloul, D. Beringer, N. Sample and G. Wiederhold, “CPAM, A Protocol for Software Composition," CAiSE'99, Heidelberg, Germany, June 1999 (Springer LNCS).
20. A. Garvey, K. Decker, and V. Lesser, “A Negotiation-based Interface Between a Real-time Scheduler and a Decision-Maker,” Tech. Rep. 94-08, U. of Massachusetts Department of Computer Science, March 1994.
21. A. Garvey and V. Lesser. “Design-to-time scheduling with uncertainty,” CS Technical Report 95--03, University of Massachusetts, 1995.
22. F. Berman, “High-performance schedulers,” The Grid: Blueprint for a New Computing Infrastructure, 1999.
23. R. Buyya, J. Giddy, D. Abramson, “A Case for Economy Grid Architecture for Service-Oriented Grid Computing,” 10th IEEE International Heterogeneous Computing Workshop (HCW 2001), In conjunction with IPDPS 2001, San Francisco, CA, April 2001.
24. A. Geppert, M. Kradolfer, and D. Tombros. “Market-Based Workflow Management,” Int'l Journal on Cooperative Information Systems (IJCIS), 7(4):297--314, December 1998.
25. M.J. Atallah et al, "Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations," Journal of Parallel and Distributed Computing, Vol. 16, 1992.
26. High Performance Fortran Forum (HPFF), "HPF Language Specification", Version 2.0, January 31, 1997.
27. A. Grimshaw and W. Wulf. “Legion - a View from 50,000 Feet,” Proc. 5th IEEE Symp. on High Performance Distributed Computing, pp. 89-99, IEEE Press, 1996.
28. N. Sample, C. Bartlett, M. Haines, “Mars: Runtime Support for Coordinated Applications,” Proceedings of the ACM Symposium on Applied Computing, San Antonio, TX, February 28- March 2, 1999.
29. P. Lawrence, editor, Workflow handbook 1997, John Wiley 1997.
30. H. Pozewaunig, J. Eder, and W. Liebhart. “ePERT: Extending PERT for Workflow Management Systems,” In First EastEuropean Symposium on Advances in Database and Information Systems ADBIS ' 97, St. Petersburg, Russia, September 1997.
31. J. Eder, E. Panagos, H. Pezewaunig, and M. Rabinovich, “Time Management in Workflow Systems,” In 3rd Int. Conf. on Business Information Systems, 1999.
32. J. Doyle, “Reasoned assumptions and Pareto optimality,” Proc. Ninth International Joint Conference on Artificial Intelligence, 1985.
33. T. Anderson, D. Culler, and D. Patterson, “A Case for Networks of Workstations: NOW,” IEEE Micro, February 1995.
34. R. G. Smith, “The CONTRACT NET: A formalism for the control of distributed problem solving,” In Proceedings of the 5th Intl. Joint Conference on Artificial Intelligence (IJCAI-77), Cambridge, MA, 1977.
35. R. Balzer and K. Narayanaswamy, “Mechanisms for generic process support,” In Proc. First ACM SIGSOFT Symp. Foundations Software Engineering, pages 21--32. ACM Software Engineering Notes, Vol. 18(5), December 1993