Automated System Design for Availability

G. (John) Janakiraman, Jose Renato Santos, Yoshio Turner
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto

HPL-2004-54
March 31, 2004*

E-mail: {john.janakiraman,joserenato.santos,yoshio.turner}@hp.com

availability, automated design, availability modeling, utility computing, failures, faults

Large-scale systems experience frequent failures which can result in unacceptably high service downtime or application execution time. To meet performance and availability requirements, the user must perform a complex design task including the selection and configuration of hardware and software components and mechanisms for handling failures. We believe users should be relieved of this burden by automating the design process in order to generate cost-effective solutions from high-level application requirements. In this paper, we present Aved, a proof of concept design automation engine which is a first step toward this goal. We describe how infrastructure choices, application models, and user requirements are represented with Aved to automate design space search and reason about design alternatives. We additionally present examples to illustrate how Aved can generate a complete picture of the cost-availability and cost-performance tradeoffs for the infrastructure design.

Automated System Design for Availability

G. (John) Janakiraman Jose Renato Santos Yoshio Turner
Hewlett-Packard Laboratories
{john.janakiraman,joserenato.santos,yoshio.turner}@hp.com

Abstract

Large-scale systems experience frequent failures which can result in unacceptably high service downtime or application execution time. To meet performance and availability requirements, the user must perform a complex design task including the selection and configuration of hardware and software components and mechanisms for handling failures. We believe users should be relieved of this burden by automating the design process in order to generate cost-effective solutions from high-level application requirements. In this paper, we present Aved, a proof of concept design automation engine which is a first step toward this goal. We describe how infrastructure choices, application models, and user requirements are represented with Aved to automate design space search and reason about design alternatives. We additionally present examples to illustrate how Aved can generate a complete picture of the cost-availability and cost-performance tradeoffs for the infrastructure design.

1. Introduction

The computing infrastructure which supports a service or application must be designed carefully to meet requirements for service performance and availability. In large-scale systems, these requirements must be met despite the occurence of frequent failures of hardware and software components. For enterprise services, such failures can result in unacceptably high levels of service unavailability. For applications, component failures can result in lost computation which must be re-executed, potentially leading to unacceptably long application execution time.

To meet service availability and performance requirements, a complex design task must be performed to judiciously select and configure hardware and software components, and also mechanisms for handling failures. The design space that must be explored can be large and consist of multiple dimensions such as choice of hardware components, software configurations such as state checkpointing interval, use of redundancy through active, standby, or cold sparing, redundancy in network paths, and the use of software rejuvenation techniques. Each choice presents a different tradeoff among availability, performance, and cost of ownership. The problem is to find a solution from this multi-dimensional design space that provides the best cost-benefit tradeoff to the user. This tradeoff can be modeled with a utility function of cost, performance, and availability. In a simple case, the problem can be reduced to finding a minimum cost solution that meets the user's availability and performance goals specified as simple thresholds.

In current practice, users explore the design space by drawing on their expertise and experience to manually generate system design alternatives. To evaluate the availability of the generated design alternatives, users invoke an availability modeling tool [4][2][12], and databases of component failure rates and repair times, in order to predict service downtime.

We believe that users should be relieved of the burden of low-level infrastructure design and configuration. Instead, the design process should be automated, freeing users to focus on service and application logic. An automated system design engine should require users to specify only high-level application requirements, and the engine should automatically generate cost-effective solutions that satisfy the requirements. An automated design process may generate optimal designs more rapidly than a human user. In addition, it may improve solution quality by covering a wider range of design alternatives than is usually feasible with a manual approach. Moreover, automated design can quickly generate a complete characterization of the tradeoffs of cost, availability, and performance for the infrastructure design.

In this paper, we present Aved, a proof of concept automated system design engine which is a first step toward this goal. Aved currently targets the automated design of clustered systems to support multi-tier enterprise services (e.g., an Internet service with web, application, and database tiers) and to support parallel compute-intensive applications (e.g., scientific applications). Aved automatically generates designs and evaluates them inside an execution loop that iterates to find a design that meets availability and performance requirements at minimum cost. To evaluate generated designs, Aved interfaces to an external availability modeling tool, which is used to predict service uptime. The predictions are predicated on the use of best practice IT management [11] and thus provide an upper limit on the availability that can be achieved. For applications that must recover lost work after component failures, Aved additionally invokes a model to predict application execution time, taking into account re-execution of lost computations.

Aved can improve the design process compared to traditional manual approaches. In addition, Aved is useful for emerging *utility computing* environments [7][13][8][15][3]. In these environments, a user who wishes to deploy an application or a service can issue a request to a computing utility, which in response automatically allocates and configures appropriate resources from pools of compute, storage, and networking resources to create a secure, virtualized computing environment that realizes the service. A more ambitious goal for utility computing is to automatically manage the provisioned service throughout its lifetime by dynamically tuning the design and deployment of the service's computing infrastructure in response to changes in service workload, component failures, etc. In this vision, the computing utility would continuously manage the service infrastructure to ensure that service availability and performance are at levels that are adequate for the user. A critical component of such a self-managing computing utility is an automated design engine, as exemplified by Aved, which would design a service's computing infrastructure and dynamically re-design it whenever necessary throughout the service's lifetime.

The key challenge in creating a design engine such as Aved is to devise practical techniques for modeling and searching the design space. The design space model used by Aved must describe application characteristics and the various infrastructure choices. The model must represent the impact of different infrastructure options on system attributes that can affect service performance or availability. The infrastructure options include the components that can be used in a design and how they can be configured, composed, and interconnected to form a complete system. The infrastructure options also include the various combinations and configurations of availability mechanisms that can be used to reduce downtime or lost computation. For Aved to be practical, the model must not be so complex that it is infeasible to specify. However, the system model must also be powerful enough to represent infrastructure choices for a large and important class of services and applications.

The next section presents an overview of Aved. Section 3 describes the design space model used by Aved. Section 4 discusses availability evaluation of designs and how Aved searches the design space. Section 5 presents examples to illustrate how Aved can be used to generate a complete picture of the cost-availability and cost-performance tradeoffs for the infrastructure design. Finally, Section 6 summarizes related work.

2. Overview of Aved

The overall architecture of Aved is shown in Fig. 1. Aved takes as input *service requirements* which specify the desired performance and availability levels for the service or appli-

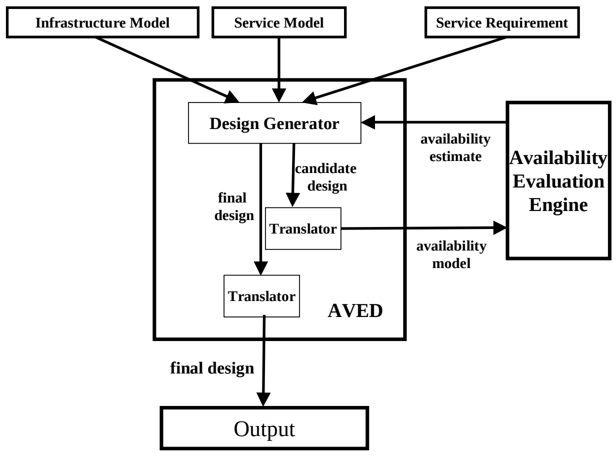

Figure 1. Aved architecture

cation, and a *design space model* which describes the entire space of choices about the computing infrastructure for the service/application. The design space model consists of an *infrastructure model* that describes the building blocks that can be selected and configured as part of a design, and a *service model* that describes how the service or application would behave on the building blocks. Whereas a user would provide a different service model for each new service or application to be deployed, the infrastructure model could be maintained in a repository and be used for all services and applications.

Aved searches the design space by generating and evaluating a series of individual designs. Each design is obtained by resolving all the design choices that are defined in the design space model (e.g., selecting the type of resource, the number of resources, the configuration of availability mechanisms, etc.). Aved evaluates each design that it generates by feeding an availability model of the design to an external availability evaluation engine. After searching the design space, Aved outputs the minimum cost design that satisfies the service requirements.

The *service requirements* inputs to Aved are specified differently depending on the type of application. Aved currently targets the design of cluster systems to support two types of applications: multi-tier enterprise services and parallel scientific applications. For enterprise services that are intended to service client requests indefinitely, the service requirements include a performance metric and an availability metric. The performance metric is specified as the desired minimum throughput in service-specific units such as requests per second for the expected type of request, and the availability metric is specified as the maximum *annual downtime* allowed. We use the term *annual downtime* or simply *downtime* to indicate the expected time a service will be unavailable in a year. We consider a service to be unavailable whenever the number of active resources is not sufficient to meet the service performance requirement.

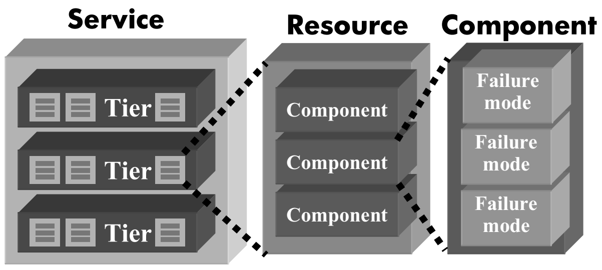

Figure 2. Design model

For jobs, such as scientific applications, that execute a finite computation, the critical metric for users is the expected time to complete the job. Therefore, the service requirements specified to Aved for these applications include only a performance requirement, expressed as job execution time. Although Aved considers the effects of availability metrics such as downtime on the time it takes to complete a job successfully, the values of these availability metrics are irrelevant to the end user as long as the job completion time requirement is satisfied.

3. Design Space Model

A high level representation of a design model is shown in Fig. 2. A service or application is composed of one or more tiers. A service is considered *up* if and only if all tiers are *up*; otherwise the service is considered *down*. Each tier is composed of a set of resources, which must be selected by Aved. A resource is the basic unit of allocation of infrastructure elements to a service or application. Currently, we assume all resources within a tier are identical, but different tiers can use different types of resources. During operation, a resource can be in one of two operational modes: *active* or *spare* mode. A resource is *active* if it is being used to support the service workload, and is in *spare* mode if it is set aside to replace an active resource upon its failure. A resource consists of a set of components. A component is an element that can fail and thus is the basic unit of fault management. A resource fails if any of its components fails. Components can correspond to hardware elements (e.g., an IA-32 compute node) or software elements (e.g., an operating system, or an application software). Each component can have multiple failure modes which indicate different ways in which a component can fail. For example, a hardware element can experience a permanent failure that requires component repair or replacement, or it can suffer a temporary failure that can be fixed by simply resetting the component.

The design space model used by Aved must represent the different choices to select designs. Sections 3.1 and 3.2 explain how these choices are represented in the infrastructure and service models used by Aved.

3.1. Infrastructure Model

The infrastructure model describes availability and cost attributes of the possible building blocks of designs. It describes various types of components, their failure modes, availability mechanisms and their configuration options and impacts, and the composition of components into resources. The infrastructure model is specified as a structured list of attribute-value pairs. An example infrastructure model specification is shown in Fig. 3.

3.1.1. Components To properly balance modeling precision against specification simplicity, we limit the granularity of component description. For example, a computer's hardware is modeled as a single component, but low-level details such as the number of redundant fans in the computer are not modeled explicitly. The specification of a component type describes its failure modes, the component's cost, and optionally the maximum number of instances of the component type that can be used in a design. Each failure mode is described by its Mean Time Between Failures (MTBF), the time required to detect the failure of this type, and the Mean Time To Repair (MTTR) the component after the failure is detected. Application software component types optionally have an attribute *loss window* that represents the maximum amount of computation that may have to be re-executed to regenerate lost results or application state upon each failure event. For applications of finite duration, the loss window size critically affects the expected job completion time. In the worst case, the *loss window* equals the total job, but it may also be a fraction of the job, if the job has well defined points at which it saves intermediate results[1]. A component has a cost attribute which includes the component's annual operational cost (energy, annual software license, etc.) and the initial (capital) cost of the component annualized by dividing by its useful lifetime in years. A component instance in a design can be configured in one of two *operational modes*: *inactive* (powered-off) or *active*. All components of an *active* resource must be in *active mode*, but spare resources can have some or all resources in *inactive mode*. The model allows defining a component's cost as a function of its operational mode to model cases where, for example, electricity costs are incurred only when a hardware component is powered on, or where inactive mode software components are free of license costs. By using inactive components in spare resources, the infrastructure cost can be reduced, but the failover time is usually increased due to the time required to activate the components.

3.1.2. Availability Mechanisms Availability mechanisms are modeled as configurable operators that specify or modify the values of other attributes of the design

1 *loss window* can be defined either in units of application work or in units of time. If necessary, Aved will convert from one unit to the other using the performance model of the application.

```
\\ Units - s:seconds, m:minutes, h:hours, d:days
\\ COMPONENTS DESCRIPTION
component=machineA cost([inactive,active])=[2400 2640]
  failure=hard mtbf=650d mttr=<maintenanceA> detect_time=2m
  failure=soft mtbf=75d mttr=0 detect_time=0
component=machineB cost([inactive,active])=[85000 93500]
  failure=hard mtbf=1300d mttr=<maintenanceB> detect_time=2m
  failure=soft mtbf=150d mttr=0 detect_time=0
component=linux cost=0
  failure=soft mtbf=60d mttr=0 detect_time=0
component=unix cost([inactive,active])=[0 200]
  failure=soft mtbf=60d mttr=0 detect_time=0
component=webserver cost=0
  failure=soft mtbf=60d mttr=0 detect_time=0
component=appserverA cost([inactive,active])=[0 1700]
  failure=soft mtbf=60d mttr=0 detect_time=0
component=appserverB cost([inactive,active])=[0 2000]
  failure=soft mtbf=60d mttr=0 detect_time=0
component=database cost([inactive,active])=[0 20000]
  failure=soft mtbf=60d mttr=0 detect_time=0
component=mpi cost=0 loss_window=<checkpoint>
  failure=soft mtbf=60d mttr=0 detect_time=0

\\AVAILABILITY MECHANISMS
mechanism=maintenanceA
  param=level range=[bronze,silver,gold,platinum]
  cost(level)= [380 580 760 1500]
  mttr(level)=[38h 15h 8h 6h]
mechanism=maintenanceB
  param=level range=[bronze,silver,gold,platinum]
  cost(level)=[10100 12600 15800 25300]
  mttr(level)=[38h 15h 8h 6h]
mechanism=checkpoint
  param=storage_location range=[central,peer]
  param=checkpoint_interval range=[1m-24h;*1.05]
  cost=0
  loss_window=checkpoint_interval
```

```
\\ RESOURCES DESCRIPTION
resource=rA reconfig_time=0
  component=machineA depend=null startup=30s
  component=linux depend=machineA startup=2m
  component=webserver depend=linux startup=30s
resource=rB reconfig_time=0
  component=machineB depend=null startup=60s
  component=unix depend=machineA startup=4m
  component=webserver depend=linux startup=30s
resource=rC reconfig_time=0
  component=machineA depend=null startup=30s
  component=linux depend=machineA startup=2m
  component=appserverA depend=linux startup=2m
resource=rD reconfig_time=0
  component=machineA depend=null startup=30s
  component=linux depend=machineA startup=2m
  component=appserverB depend=linux startup=30s
resource=rE reconfig_time=0
  component=machineB depend=null startup=60s
  component=unix depend=machineB startup=4m
  component=appserverA depend=unix startup=2m
resource=rF reconfig_time=0
  component=machineB depend=null startup=60s
  component=unix depend=machineA startup=4m
  component=appserverA depend=unix startup=30s
resource=rG reconfig_time=0
  component=machineB depend=null startup=60s
  component=unix depend=machineA startup=4m
  component=database depend=unix startup=30s
resource=rH reconfig_time=0
  component=machineA depend=null startup=30s
  component=linux depend=machineA startup=2m
  component=mpi depend=linux startup=2s
resource=rI reconfig_time=0
  component=machineB depend=null startup=60s
  component=unix depend=machineB startup=4m
  component=mpi depend=unix startup=2s
```

Figure 3. Infrastructure specification example

(e.g. MTBF, MTTR, etc.), thus changing the availability characteristics of the system. An example of an availability mechanism is a maintenance contract with a configurable level which determines the response time of repair staff. Another example is a checkpoint-restart mechanism that periodically saves the state of an application to stable storage to limit the loss window. A mechanism is specified by defining a list of configuration parameters, each with a defined range of possible settings, and a list of attributes whose values are affected by the mechanism and its configuration parameter settings. For example, in the selection of a maintenance contract, the level of the maintenance service (e.g. bronze, silver, gold, platinum, etc.) can be defined as a parameter whose value is used to determine the repair times (mttr) of components. Each availability mechanism also has a cost attribute which indicates the annual cost of using the mechanism. The cost can be a function of the mechanism's parameter settings (e.g., cost can increase with the level of maintenance contract).

Availability mechanisms can affect the attributes of more than one failure mode or component type (e.g., an application-transparent checkpointing mechanism could be applied to both scientific applications and databases). Therefore, we chose to separate the availability mechanism description from the component specification and allow mechanisms to be applied and configured independently for each component at design time.

3.1.3. Resources A resource type is defined as a combination of components that can be allocated as a unit to a service. Attributes of a resource include the startup times of each of its components, and the dependencies among components. Component dependencies indicate both the order in which components of a resource must be started, as well as which components are affected by a failure of another component. For example, an operating system can only be started after the hardware is started, and a hardware failure causes the operating system to fail as well. Finally, a resource has an attribute giving the reconfiguration time incurred upon failover to a spare resource of this type. Reconfiguration time includes, for example, time to reconfigure load balancers and to transfer needed data to the spare resource instance.

3.2. Service Model

The service model describes a service as a set of tiers, where each tier is a cluster of resource instances. Example service model descriptions are shown in Figs. 4 and 5. For each tier, the service model defines a list of possible resource types that can be used to support that tier. For each of these resource options, the service model must capture the resulting model of parallelism for the tier for analysis of the performance and availability behavior of the tier and the overall service. Thus for each resource option for each tier, the service model defines the freedom in setting and varying the number of active resources, the impact of the failure of a single resource instance, and the performance characteristics of the tier if the resource is selected for a design. The attribute *nActive* specifies the set of possible number of active resources. This allows specifying, for example, that a scientific application requires the number of resources to be a power of 2, or that a non-parallel application requires exactly one resource. The attribute *sizing* determines whether during the lifetime of a service the number of resource instances can be changed (*dynamic*) or not (*static*). An example of static sizing is a scientific application that partitions the data across the nodes at initialization time and cannot support subsequent data redistribution. An example of dynamic sizing is a web tier where the number of web servers can change arbitrarily. A *failure scope* attribute defines whether failure of a single resource instance affects the operation of the whole *tier* or whether the impact is limited to the failed *resource* instance. Finally, the performance associated with a resource option is specified in service-specific *units of work* per units of time and is typically defined as a function of the number of active resources.

Although availability mechanisms are part of the infrastructure model, their performance impact depends on characteristics of the specific service. Therefore, we currently define performance impact of availability mechanisms as service attributes associated with each resource alternative of the tier affected by the mechanism[2]. In the future, we plan to specify parameterized performance functions as part of the infrastructure model rather than the service model, in order to simplify the task of specifying services. Services would only have to specify parameters for the performance functions.

Finally, for applications of finite duration for which the service requirement is the expected time to complete the job, the service model has an attribute which specifies the job size in application-specific units (e.g., the number of frames in a rendering application).

```
application=ecommerce
  tier=web
    resource=rA sizing=dynamic failurescope=resource
      nActive=[1-1000,+1] performance(nActive)=perfA.dat
    resource=rB sizing=dynamic failurescope=resource
      nActive=[1-1000,+1] performance(nActive)=perfB.dat
  tier=application
    resource=rC sizing=dynamic failurescope=resource
      nActive=[1-1000,+1] performance(nActive)=perfC.dat
    resource=rD sizing=dynamic failurescope=resource
      nActive=[1-1000,+1] performance(nActive)=perfD.dat
    resource=rE sizing=dynamic failurescope=resource
      nActive=[1-1000,+1] performance(nActive)=perfE.dat
    resource=rF sizing=dynamic failurescope=resource
      nActive=[1-1000,+1] performance(nActive)=perfF.dat
  tier=database
    resource=rG sizing=static failurescope=resource
      nActive=[1] performance=10000
```

Figure 4. Service model example: e-commerce

```
application=scientific jobsize=10000
  tier=computation
    resource=rH sizing=static failurescope=tier
      nActive=[1-1000,+1] performance(nActive)=perfH.dat
      mechanism=checkpoint mperformance(storage_location,
        checkpoint_interval,nActive)=mperfH.dat
    resource=rI sizing=static failurescope=tier
      nActive=[1-1000,+1] performance(nActive)=perfI.dat
      mechanism=checkpoint mperformance(storage_location,
        checkpoint_interval,nActive)=mperfI.dat
```

Figure 5. Service model example: scientific

4. Searching the Design Space

Aved searches the design space specified by the infrastructure and service model to identify the optimal design that meets service requirements. This optimal design specifies, for each tier, values for: **1)** the type of resource selected for the tier, **2)** the number of active resources, **3)** the number of spare resources, **4)** the operational mode of each component in the spare resources, and **5)** each parameter of availability mechanisms. We describe the algorithm used to search the design space first in Section 4.1 and then describe how each design generated in the design space search is evaluated in Section 4.2.

4.1. Search Algorithm

The design space search algorithm first examines each tier in isolation and determines the optimal design for each tier assuming the other tiers do not experience failures. If the overall design combining these individually optimal designs satisfies the high level requirements, this is the optimal design. If not, the search continues by refining the design of each tier until the optimal multi-tier design is identified. To refine the design of a tier, Aved computes the best "next" design by making the requirements for that tier incrementally more aggressive.

The design space search for each tier considers all possible choices of resource types for that tier to find the best solution for each possible resource type. For each resource type, Aved first evaluates designs using the minimum number of resources required to meet the performance requirement in the absence of any failures. The number of resources is successively incremented in subsequent search steps. For each selected number of resources, all possible combinations of the number of active resources, the number of spare resources, the operation mode of spare resources, and values for the parameters of availability mechanisms are considered and evaluated. For each design, Aved evaluates both its cost and availability until a design that satisfies the service requirements is

2 Currently, mechanisms affecting more than one tier cannot be modeled.

found. Once such a solution is found, subsequent designs are evaluated for cost first (which is simple and fast) and higher cost designs are rejected without evaluating their availability. If a feasible solution is not found and the availability metric starts to degrade when the number of resources is increased, no feasible solution exists and the search stops. The search ends successfully when increasing the number of resources generates only designs with costs higher than the current solution.

4.2. Design Evaluation

The evaluation of designs generated by the design search process has two parts: evaluation of cost and evaluation of availability. The cost of a design is simply calculated as the sum of the cost of all components at their selected operational mode (*active* or *inactive*) and the cost of the availability mechanisms for the selected values of their parameters.

To evaluate availability, Aved generates an availability model for the design. Multiple tiers in a design are modeled as an association in series, where the whole design is considered up only when each tier is up. The availability model for each tier specifies the following parameters:

1. *n*, the number of active resources.
2. *m*, the minimum number of active resources required for the tier to be considered up. This is equal to *n* when *sizing* is declared as *static* or *failure scope* is declared as *tier* for the resource used in this tier. Otherwise, this value is computed using the performance requirement combined with the performance model of the service associated with the resource used in this tier.
3. *s*, the number of spare resources. Spare resources can be used to replace failed resources if the number of active resources drops below the minimum *m*, but incurring downtime during the failover.
4. $MTBF_i$, the MTBF of each failure mode *i* that is possible for each component that is included in the resource used in this tier. The MTBFs are the same as defined in the design space model.
5. $MTTR_i$, the MTTR for the repair of each failure mode *i* that is possible for each component that is included in the resource used in this tier. The MTTR is computed as the sum of the failure detection time, the component repair time for that failure mode, and the startup times of the components affected by the failure.
6. $FailoverTime_i$, the failover time for each failure mode *i* that is possible for each component that is included in the resource used in this tier. The failover time is computed as the sum of the failure detection time, the reconfiguration time, and the startup latencies of components that are in *inactive* operational mode in the spare resource.

When generating an availability model, Aved only considers failover for failure modes in which MTTR is greater than the failover time. While this is the common case for permanent hardware failure, it may not be the case for software and hardware glitches that can be fixed by just restarting the component with a corresponding repair time equal to zero.

This availability model can be evaluated using a traditional availability evaluation engine, such as Avanto [4], Mobius[2], and Sharpe[12] to compute the expected annual downtime of the design. Aved currently generates representations of this availability model that can be used with Avanto [4] and our own simplified Markov Model (this can be easily translated to work with other engines). These availability evaluations assume failures are independent but their distributions depend on the assumptions of the specific availability evaluation engine (e.g., exponential interarrival for the Markov model).

For applications with a specific job size (e.g., scientific applications), Aved must also estimate the expected job completion time. The job completion time is derived analytically using the average annual downtime estimate from the availability engine. For this analysis, we assume failure modes are independent with the time between failures and the time to repair components being exponentially distributed. If the application has a *loss window lw* (which is the maximum amount of work that can be lost due to a single failure event), we define T_{lw} ($T_{lw} \geq lw$) as the mean computation time necessary to execute an lw amount of useful work. Based on the assumption of exponential distributions, T_{lw} can be computed as:

$$T_{lw} = mtbf\frac{P_f}{(1-P_f)} \quad , \quad P_f = 1 - e^{-lw/mtbf} \qquad (1)$$

This formula can be obtained using a formal analytical derivation, but we provide a simple interpretation here. P_f is the probability that there is at least one failure during an interval of duration lw. The ratio $(1-P_f)/P_f$ is the average ratio of the number of intervals of duration lw that do not experience failures to the number of intervals that do experience failures. In an interval of duration MTBF, there is only one failure on average. Thus, there are approximately $(1-P_f)/P_f$ intervals of duration lw that do not experience failures for each MTBF interval (ignoring intervals with multiple failures). Since T_{lw} is defined as the mean time required to execute lw time units of useful work, there should be $(1-P_f)/P_f$ intervals of duration T_{lw} in an MTBF interval as indicated by the equation above.

With this equation we can compute the fraction of the computation time that is used for useful work as lw/T_{lw}. Given the uptime T_{up} computed by the availability model engine, we can compute the effective uptime that the system is executing useful work as $T_{ef} = T_{up} * (lw/T_{lw})$. We can then easily compute the expected job execution time using the performance model of the application and the job size.

tier, resource	attribute	function
application, rC	performance(n)	200*n
application, rD	performance(n)	200*n
application, rE	performance(n)	1600*n
application, rF	performance(n)	1600*n
computation, rH	performance(n)	(10*n)/(1+0.004*n)
computation, rI	performance(n)	(100*n)/(1+0.004*n)

tier, resource	attribute	function
computation, rH	mperformance(*central*,cpi,n)	max(10/cpi,100%) ($n < 30$)
		max(n/(3*cpi),100%) ($n \geq 30$)
	mperformance(*peer*,cpi,n)	max(20/cpi, 100%)
computation, rI	mperformance(*central*,cpi,n)	max(5/cpi,100%) ($n < 30$)
		max(n/(6*cpi),100%) ($n \geq 30$)
	mperformance(*peer*,cpi,n)	max(100/cpi, 100%)

(cpi is checkpoint interval expressed in minutes)

Table 1. Performance functions for examples

5. Examples

We illustrate the value of Aved using two simple example scenarios. In the first we consider the design of the computing environment for the application tier of an e-commerce service, whose model is described in Fig. 4. In the second example we consider the design of the computing environment for a compute intensive scientific application whose model is described in Fig. 5. For both examples we assume infrastructure model and attribute values defined in Fig. 3 and Table 1. The examples shown in this section are hypothetical but based on realistic parameters expected for these types of application. The goal of this section is not to describe the detailed characteristics of a specific real scenario, but to illustrate how Aved could be useful for designing the computing infrastructures for different classes of application.

5.1. Application Tier Example

In this example, the following design dimensions are explored by Aved: i) resource type, ii) number of active machines, iii) number of spare machines[3], iv) selection of maintenance contract. We assume the service can be supported on two different types of machine: a dual processor machine (*machineA*) which can run Linux, and a more powerful 16-way machine (*machineB*) which runs a proprietary version of UNIX. In addition, one of two different types of J2EE application server software, *appserverA* and *appserverB*, can be installed on either hardware platform. By combining the two hardware options with the two software options, Aved can explore four different resource options. We obtained failure rates or MTBF values for hardware components from the manufacturer historical database. We selected costs and response times for service maintenance contracts based on typical contracts offered by the hardware vendors. Software and hardware costs were obtained from vendors' published prices. However, software failures rates were estimated based on the authors' intuition, since this data was not readily available.

3 For simplicity, in this example we restricted components of spare resources to be inactive.

We have used Aved to identify the optimal designs for this example scenario over a range of service performance and availability requirements. Fig. 6 shows these optimal designs as a function of the performance requirement (application units of load) and the availability requirement (annual downtime). In this Figure, designs are grouped in families, represented by tuples of the form (*resource, contract, n_extra, n_spare*), where *resource* indicates the selected type of resource; *contract* indicates the selected service maintenance contract; *n_extra* indicates the number of extra active machines used for availability (i.e. the extra machines in addition to the minimum number of machines needed for performance in case of no failure); and *n_spare* indicates the number of inactive spares Each design in a family can use a different number of machines depending on the load, i.e., a design family with *m* spares has a fixed number of redundant machines, in addition to the number of primary machines which can vary as a function of a load. To facilitate the discussion in the rest of the paper we also refer to the design families using numbers as identified in Fig. 6. The load range shown on the *x axis* varies from 400 to 5000 *load units*. The *y axis* shows the range of practical annual downtime values, from a fraction of a minute to 10,000 minutes, i.e., approximately one week. In the two-dimensional space of requirements mapped by the performance requirement and the availability requirement, each curve corresponds to a particular design family that is cost optimal for all requirement points above the curve (and points on the curve) and beneath the immediately higher curve. Furthermore, the curve plots the downtime estimate for each design of this family at various load levels where it is the optimal solution. Therefore, for requirement points above the curve, the downtime estimated with this design family is less (i.e., better) than the requirement. For example, for a requirement ($load = 1000$, $downtime = 100$) in Fig. 6, the curve immediately below this point corresponds to the optimal design family (number 9), which has downtime of approximately 50 minutes.

The results in Fig. 6 show that despite the small size of our example design space, the number of optimal solutions distributed across the requirements space is large and would be tedious to evaluate manually. The results also show that Aved filters out suboptimal solutions. For example, design family 3 is not selected for loads above 1400 *units*. For loads above that, design family 6, which provides lower downtime, is selected instead of family 3. For low loads the extra cost of the *gold* maintenance contract is lower than the cost of an additional resource and designs of family 3 are preferred. As the load increases, the extra cost of the *gold* contract becomes higher than an extra resource, since the cost of a maintenance contract is proportional to the number of machines it covers. Thus for higher loads it is better to use an extra resource than

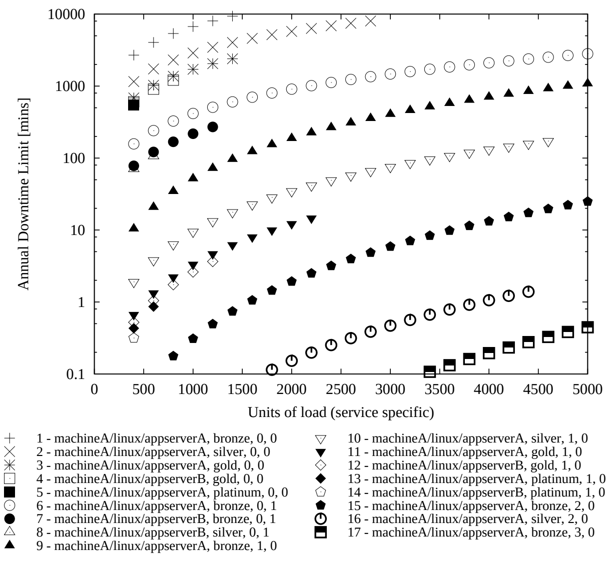

Figure 6. Optimal solution for a range of service requirements: load and annual downtime limit.

to pay for a higher maintenance contract.

As shown in Fig. 6, the more powerful *machineB* is never selected. This is expected since we assumed linear scalability for the application (Table 1), and the low end machines have a better cost-performance ratio (i.e., lower cost per unit of load). However, the situation may be different if the application performance scales sublinearly with the number of resources as in the case of the scientific application described in section 5.2.

We observe in Fig. 6 that the downtime estimated for a particular design family increases with load. This is expected since higher load levels require a larger number of resources which results in a higher failure rate (because if any of these resource fail, the service cannot meet its minimum performance requirement and the service is considered down). Therefore, the optimal design family may change as the load level fluctuates. In a utility computing environment, where the infrastructure can be easily reconfigured, an automated design engine such as Aved could dynamically re-evaluate and change designs as conditions change.

5.2. Scientific Application Example

In a second usage scenario for Aved, we consider the design of a system for supporting a long running scientific application which is described in Fig. 5. For such applications a checkpointing mechanism can significantly reduce re-execution of computation after failures. As described in the infrastructure model shown in Fig. 3, we consider a checkpoint mechanism with two configuration parameters: checkpoint interval and storage location where the application state is saved. The storage location parameter indicates if the the state is saved in a central network attached file system (*central*); or alternatively in both the local disk and in the disk of a peer resource (*peer*). The checkpoint mechanism determines the value of the *loss window* associated with the application, based on the selected checkpoint interval. The performance impact of the mechanism is defined as a function of the selected storage location, checkpoint interval, and number of resources.

To avoid overloading the graphs, we fixed the *maintenance* contract to the bronze level in this example. The job can be supported on *machineA* running *linux* and *mpi* or on the more powerful *machineB* running *unix* and *mpi*. The other design options are the same as used in the first example, but the values of the attributes in the service model descriptions are different since the applications have different characteristics. The only high level service requirement for the job is the expected execution time of the job.

Fig. 7 shows the optimal design choices made by Aved as a function of the job execution time requirement, for a wide range of values. As opposed to the first example in which just one type of resource was selected for all range of requirements, in the second scenario both resource types are selected depending on the specified requirement. When higher job execution time is tolerated, the system selects a lower cost resource based on *machineA* (e.g. a dual processor machine); and when a lower execution time is required the more expensive component *machineB* (e.g. a 16-way machine) is selected. For lower execution times a huge number of low cost machines would be required, but this is not cost effective because of the sublinear scalability of the scientific application. In addition, the large number of resources would increase the failure rate and thus increase the system overhead of re-executing lost computation. Unlike the application tier scenario in which adding resources is always beneficial (since a resource failure does not affect other resources in the tier), in this scientific application scenario we assume a failure causes the whole tier to be down. Thus adding resources can reduce the availability of the system and increase job execution time.

As expected, for the same resource type the number of resources decreases as the user tolerates a longer execution time. The number of resources is determined not only by the performance characteristics of the application, but also by the availability characteristics of the system, for example the time lost to redo computation after a failure. The figure also shows that the number of spare resources increases as the number of total resources increases.

We also observe in Fig. 7 that the checkpoint interval increases as we relax the execution time requirement. A short checkpoint interval reduces re-execution after failures but in-

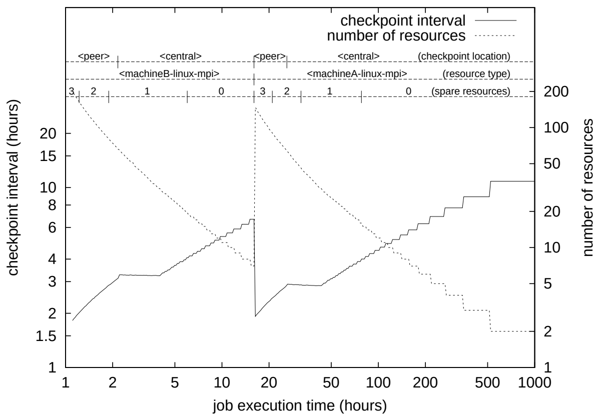

Figure 7. Scientific Application Example. Optimal design as function of execution time

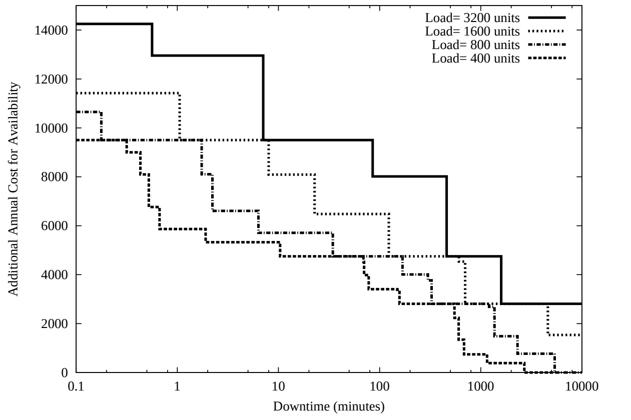

Figure 8. Cost/availability/performance tradeoff for application tier example

creases the overhead in normal operation. Both types of overhead increase job execution time. Thus the optimal configuration is an intermediate value which balances the overhead due to lost work at failures and the overhead during normal operation. As the number of resources increases, the failure rate increases and the overhead due to failures becomes more important, causing the optimal checkpoint interval to be reduced.

Fig. 7 also shows the choice of the storage location used to store the application state at each checkpoint. As expected, for a small number of nodes the best option is to save the state at a *central* storage location, since the other alternative, storage on a *peer* node, has a higher overhead per node. But for a large number of nodes, the central location becomes a bottleneck and storing the state on a *peer* node becomes more effective[4].

5.3. Cost, performance and availability tradeoff

Although the curves shown in Fig. 6 and Fig. 7 enable the selection of the optimal design for a given application requirement, the knowledge of the cost associated with each design can help the user make cost/benefit tradeoffs. Aved can be used to generate plots of the costs of optimal designs at various levels of availability and performance requirements. Fig. 8 shows the cost associated with the optimal designs at various levels of availability and performance requirements, for the application tier example shown in Fig. 6. Each curve shows, for a particular level of load, the additional annual cost as a function of the required downtime. This is the extra annual cost necessary to provide the required availability when compared to a minimum cost design that can support the same load when there is no availability requirement. Fig. 8 reveals the tradeoffs among cost, availability, and performance that must be understood to make a judicious design choice. For example, in some cases a large improvement in downtime can be achieved with a low additional cost. Alternatively, slightly relaxing the downtime requirement can significantly reduce the cost overhead for availability.

4 We assumed that the central storage is based on a highly reliable system, such as a high end RAID system, and does not fail.

6. Related Work

The idea of automating the design and configuration of systems to meet user-specified availability requirements is relatively recent. We are only aware of a few examples, each of which is focused on a limited domain. The Oracle database implements a function that automatically determines when to flush data and logs to persistent storage such that the recovery time after a failure is likely to meet a user-specified bound [10]. Automated design has been proposed for storage systems to meet user requirements for data dependability, which encompasses both data availability and data loss [9]. Their approach is complementary and could be combined with ours to obtain a comprehensive solution that designs systems based on performance, availability, and data protection requirements.

Most other work on system automation for managing availability has been limited to automated monitoring and automated response to failure events and other such triggers. For example, cluster failover products such as HP MC/Serviceguard [5] SunCluster [14] and Trucluster [6] detect nodes that fail, automatically failover application components to surviving nodes, and reintegrate failed nodes into active service when they recover from failure. IBM Director [1] detects resource exhaustion in its software components and automates the rejuvenation of these components at appropriate intervals. Various utility computing efforts underway will also automatically detect failed components and automatically replace them with equivalent components from a free pool [7][13][8].

7. Conclusions

In this paper, we have presented Aved, an automated system design engine which determines the minimum cost design and configuration of computing infrastructure that satisfies service level performance and availability requirements. Automating the design and configuration of computing infrastructure improves the design process compared to a traditional manual approach, and will be particularly useful in emerging self-managing utility computing environments.

To enable automatic exploration of the design space, we proposed a model which can represent various options for designing and configuring the infrastructure. A primary challenge in developing our design space model and its specification approach was to satisfy competing goals of generality and practicality. That is, the model should enable representation of a wide variety of system infrastructure choices and service types. At the same time, the practicality of the model should be preserved by enabling it to be specified at a conceptual level that is close to the actual, real-life properties of services and system building blocks. Our solution approach is to define a structured model which uses a small number of fundamental constructs (components, failure modes, mechanisms, etc.) which are intuitive to map to real-life entities. We carefully selected the attribute set for each construct, and defined clear dependencies between attributes in different constructs in the model which are related. This allows the constructs to be interconnected or composed in well defined ways to construct models describing a wide variety of complex systems. To demonstrate the applicability of this approach, we presented examples that show the model can represent infrastructure design options for two completely different environments, an E-Commerce service and a scientific application.

Our examples additionally illustrate the need for automated design. Although the examples include only a small number of configuration parameters, they result in a large set of possible designs. Moreover, the examples show that the optimal design can be different as the requirements change, for example as the service's throughput requirement is increased as a result of an increase in client demand. Therefore, in self-managing environments, an engine such as Aved is needed to automatically reevaluate and reconfigure designs in response to changes in such parameters.

Although Aved is a good first step towards automated system design, there are several remaining issues that need to be addressed as future research. To address overall service availability, the design engine must examine the impact of the network and storage subsystems. Thus we plan to extend Aved to factor LAN topologies and network failures. We also plan to integrate Aved with an automatic process for storage system design and management for data dependability[9]. We plan to integrate Aved with online mechanisms to continuously monitor service performance and other infrastructure attributes to dynamically refine Aved's models and to generate design changes in response to environment changes. This would eliminate the need for precise initial performance models that may be difficult to obtain and specify. Finally, we plan to conduct case studies of real environments to identify additional design choices encountered in practice and to evaluate what extensions would be needed in Aved to support them.

References

[1] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. Proactive management of software aging. *IBM Journal of Research and Development*, 45(2):311–332, March 2001.

[2] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. The Möbius modeling tool. In *9th Int'l Workshop on Petri Nets and Performance Models*, pages 241–250, Sep 2001.

[3] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for distributed system integration. *Computer*, 35(6), 2002.

[4] Hewlett Packard Company. *Availability advantage*. `h18005.www1.hp.com/services/advantage/aa_avanto.html`, January 2003.

[5] Hewlett Packard Company. *HP MC/ServiceGuard*. `www.hp.com/products1/unix/highavailability/ar/mcserviceguard/index.html`, January 2003.

[6] Hewlett Packard Company. *TruCluster software*. `www.tru64unix.compaq.com/cluster/`, January 2003.

[7] Hewlett Packard Company. *Utility computing*. `devresource.hp.com/topics/utility_comp.html`, January 2003.

[8] International Business Machines, Inc. *Autonomic computing*. `www.ibm.com/autonomic/index.shtml`, January 2003.

[9] K. Keeton and J. Wilkes. Automating data dependability. In *10th ACM-SIGOPS European Workshop*, Sep 2002.

[10] T. Lahiri, A. Ganesh, R. Weiss, and A. Joshi. Fast-Start: quick fault recovery in Oracle. In *ACM SIGMOD*, pages 593–598, 2001.

[11] Office of Government Commerce. *ITIL Service Support*. IT Infrastructure Library. The Stationery Office, United Kingdom, June 2000.

[12] R. A. Sahner and K. S. Trivedi. Reliability modeling using SHARPE. *IEEE Transactions on Reliability*, R-36(2):186–193, June 1987.

[13] Sun Microsystems, Inc. *N1: Revolutionary IT architecture for business*. `www.sun.com/software/solutions/n1/index.html`, January 2003.

[14] Sun Microsystems, Inc. *Sun[tm] Cluster*. `www.sun.com/software/cluster/`, January 2003.

[15] vmware. *VirtualCenter white paper*. `www.vmware.com/pdf/vc_wp.pdf`.