QUALITY-DRIVEN SYSTEMS

Kai-Chih Chen, Wei-Chung Lin, and Cheng-Seen Ho
Department of Electronic Engineering, National Taiwan University of Science and Technology, 43 Keelung Road, Section 4, Taipei 106, Taiwan

Abstract

Most traditional systems produce a fixed quality of output under a fixed execution environment given identical input, even when the user does not need that quality of solution. In this paper, we propose a new software architecture that can do quality-driven operations. The key concept of the architecture is that it contains multiple configurations so that it can adapt itself to fit various user requirements and constraints. The adaptation involves the tradeoff of a set of quality of service factors in order to provide the best feasible performance. A simple application is presented to demonstrate the power of the architecture. This quality-driving concept can be introduced to many domains; in fact, any resource-constrained or pay-per-use applications can be improved by the concept to provide a range of quality of services.

Keywords: Quality-Driven Systems, Quality of Service, Adaptive Software.

1. Introduction

The following features are associated with most traditional systems. (1) They produce fixed output under a fixed execution environment given identical input. (2) They produce fixed quality of output even when the user does not need that quality of solution. In a convoluted problem domain, however, finding an optimal solution may take an unacceptable long time. The wide variation in time constraints and user requirement, especially in a dynamic environment, often makes it infeasible to find the optimal solution. A suboptimal solution, instead, may be obtained quickly and found useful. In this case, responsiveness in time is sometimes more useful than accuracy in result with a very long computation time.

In this paper, we propose a novel concept — "Quality-Driven Systems (QDS)," to cope with the problem. A QDS is a system that can adapt itself to satisfy the user's requirement at runtime by accomplishing a given task in such a way that it produces an output with some quality level designated by the user. A system with this feature is much more feasible, efficient, and friendly than traditional ones. For example, a user usually expects a search program to find the optimal solution. In some time-critical environment, however, the user would desire a useful solution before the deadline rather than wait for the optimal solution that comes out after the deadline. In this case, a QDS can reconfigure itself so that it can satisfy all the constraints specified by the user.

In general, we want a QDS to adapt itself whenever it detects that the current system configuration can not satisfy all the constraints. Of course, it is impossible for a QDS to satisfy all constraint requirements all the times. In this case, a QDS may degrade the system performance, e.g. output quality, or even relax some non-critical constraints, in order to provide a good enough (acceptable) quality of service for the user. Consequently, a QDS not only has the ability of reconfiguration but also has meta-control knowledge for directing reconfiguration in order to reach advertised quality of service. As a matter of fact, a QDS needs to do quality evaluation on system-provided service, resource management, and quality management.

There have been lots of research on tradeoff between computation quality and computation cost, including anytime algorithms [2][17], progressive processing [12][13][19], self-adaptive systems [8][14], resource-bounded reasoning [18], and fault-tolerance systems [2][9][16]. Most of these research works focus on guaranteeing network flow [1], or creating a stable, robust, efficient, and fault-tolerant system. Development of a QDS may benefit a lot from them, although the goal of QDS is focusing more on creating a customizable, flexible, and humanized system.

Section 2 will propose an architecture for quality-driven systems. Section 3 illustrates an application of the architecture — "a quality-driven 8-puzzle solver". Related works are reported in Section 4 with Section 5 concluding the work.

2. A quality-driven system architecture

The most important feature of a QDS is that it can adapt itself to provide a proper degree of *quality of service* (QoS) according to the user requirement. But, what is QoS in this sense? We define the QoS of a system to be a set of QoS factors that are related to the interactions between the user and the system. Example QoS factors include response time,

0-7695-0933-9/00 $10.00 © 2000 IEEE

memory requirement, etc. Most of them are related to resource. This is because, from the viewpoint of the user, the effect of resource consumption on his operations is quite important in using a system. Formally,

$$QoS = \{Q_1, Q_2, ..., Q_i, ..., Q_n\}, \quad (1)$$

where Q_i: i_{th} QoS factor.

Thus, before we start to construct a quality-driven system, we need to define a set of QoS factors for the system. It is, however, very difficult and impractical trying to define a universal set of QoS factors for all domains. For example, the QoS factors for a quality-driven multimedia system may contain network delay, frame size, and color depth. The QoS factors for a quality-driven speech recognizer usually does not contain frame size or color depth. Instead, it may contain process speed and accuracy.

The QoS factors in Equ.(1) are at the system level, and are usually not comprehensible by the user. We need a set of user-level QoS factors, too. Equ. (2) defines the user-level QoS.

$$QoS_u = \{q_1, q_2, ..., q_i, ..., q_n\}, \quad (2)$$

where q_i: i_{th} user-level QoS factor.

Now we need a set of mapping functions, $\{f_i\}$, such that we can relate QoS to QoS_u. Formally, we define $q_i = f_i(\{Q_i\})$, meaning that f_i knows how to create a solution that meets the user-desired q_i from the system-level $\{Q_i\}$.

With the concept of QoS in mind, we now present an architecture for QDS as shown in Fig. 1. The QoS Manager (QoSMan) controls the operation of the whole system. It is responsible for deciding a proper system configuration to satisfy the user-designated operation quality based on the QoS profile information from the QoS information agent and resource management information from the resource information agent. The modeling information agent is responsible for constructing an execution model corresponding to the system configuration decided by QoSMan. The executor solves the problem based on the execution model. We explain the major components of this architecture below.

2.1 QoS profiles

A QoS profile specifies the co-relationships among the QoS factors. A simplest example of QoS profile is shown in Fig. 2. It is constructed by diagramming the relationships between the output quality and execution time. It can be formally specified as Equ. (3).

$$Q_o = f_{QoS}(Q_t), \quad (3)$$

where $Q_o \in QoS$ is the output quality factor, $Q_t \in QoS$ is the time quality factor, and f_{QoS} is the tradeoff function. The QoS profile can be used to predict the system behavior and to make the best configuration to meet the user requirement. For example, if the system has 4 seconds to process the task, the output quality is predicted to be around 0.8 in the system-level. On the other hand, if the user-designated quality is 0.8, the system can find a configuration to get that quality of solution in 4 seconds. A simple QoS profile like this makes it fairly similar to the performance profile of an anytime algorithm [17].

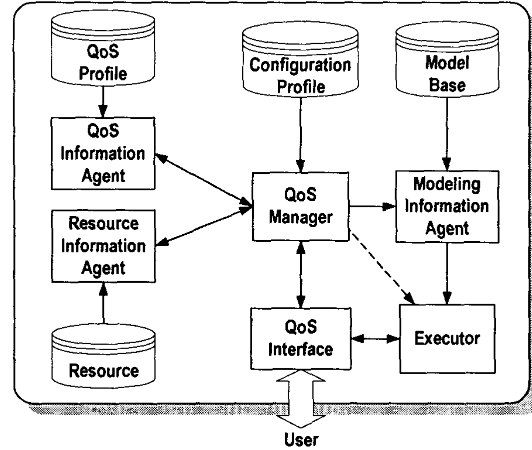

Fig. 1 Quality-Driven System Architecture

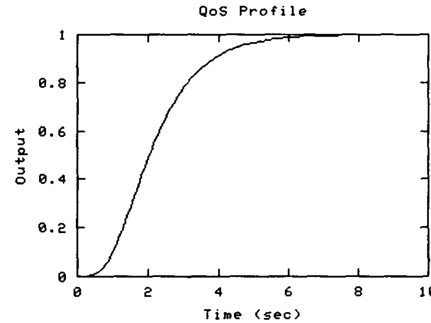

Fig. 2 Two-dimensional QoS Profile

Fig. 3 shows a QoS profile that involves multiple QoS factors, including execution time, output quality, and memory requirement. This QoS profile not only illustrates the behavior of a system, but also tells us that the minimal memory requirement for working is 1.5 KB.

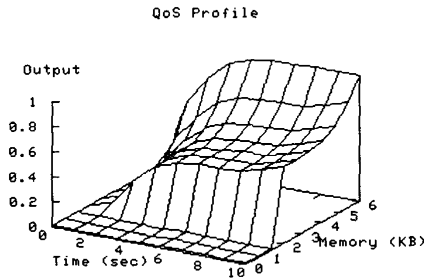

Fig. 3 Three-dimensional QoS Profile

2.2. QoS interface

The QoS interface solicits two types of information from the user, namely, problem specification and quality specification. The problem specification contains the information related to problem solving. The quality specification contains desired solution quality and/or constrained resource. To simplify the interaction with the user, the fuzzy technique is introduced to specify what fuzzy quality level solution, e.g. better or medium, is requested. Constrained resources are similarly defined. The major task of the QoS interface is thus to perform the inverse f_i function to derive the corresponding system-level QoS from the user-given quality specification.

2.3. Information agents

The three information agents assist QoSMan to accomplish quality-driven operations by supporting relevant information. First, the resource information agent monitors the relationships among a variety of resources. Thus, in a quality-driven videoconference application if QoSMan wants a higher resolution, the resource information agent will check for coherent resources. If it finds that the bandwidth is not enough, it will notify QoSMan of resource constraint violation. The QoSMan may turn to degrade the fame rate or change the compression policy to cope with the problem.

The QoS information agent supports the retrieval of a QoS profile that meets the retrieval condition set by QoSMan. The responsibility of the modeling information agent is to construct an execution model based on the system configuration concluded by QoSMan.

2.4. QoS manager

QoSMan coordinates the whole QDS components. Upon receiving the quality specification from the QoS interface, QoSMan invokes a negotiation session with the QoS information agent and resource information agent. A successful QoS negotiation session implies that at least one of the system configurations can meet all the user quality requirements. QoSMan continues to choose the best system configuration according to the user's specification, and then informs the modeling information agent to construct a proper execution model for the executor to solve this problem instance.

If QoSMan can not find a suitable system configuration that meets all quality constraints, it will rectify the QoS factors according to the negotiation policy, designated by the user, to find an acceptable QoS for the user. Finally, QoSMan informs the user of this modified QoS, asking for confirmation.

Should the negotiation fail, e.g. the user requirement exceeds system capacity, QoSMan will return a failure notification to the QoS interface. The user will be asked to modify his requirement or constraints in order to start a new operation with new quality specification.

3. Applications

We have constructed a prototype quality-driven system — a quality-driven 8-puzzle solver [10]. It contains two distinct configurations to deal with different problem and quality specification [10]. Fig. 4 defines the QoS profile for the 8-puzzle solver. Figs. 5 and 6 show two actual QoS profiles under different system configurations.

$Q_o = f_{QoS}(Q_i, Q_t)$, where
Q_o: solution quality,
Q_i: initial state quality,
Q_t: execution time (resource constraint).

Fig. 4 The QoS profile definition for the application

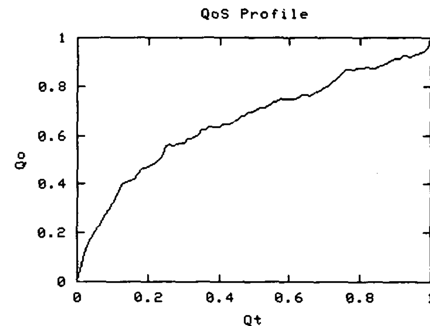

Fig. 5 QoS profile (1)

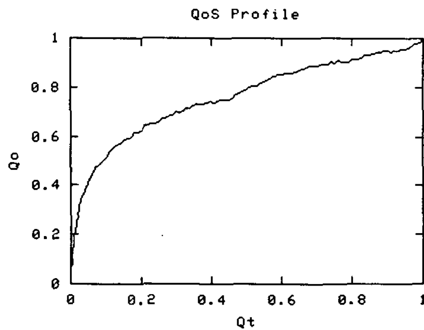

Fig. 6 QoS profile (2)

First of all, the QoS interface interacts with the user through the dialog as shown in Fig. 7. The user enters the initial state along with the desired output quality or execution time constraint. Note that the output quality and time constraint are expressed in fuzzy concepts. Fig. 8 shows the fuzzy concept definition for quality. The QoS interface is responsible for translating these user-level factors for f_{QoS}, which allows the system to estimate the possible interaction among the QoS factors and to use that to negotiate with the user.

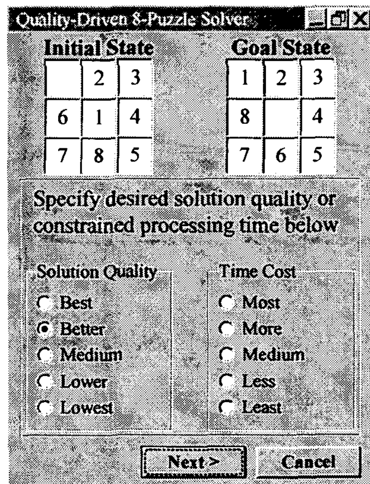

Fig. 7 User interface of 8-puzzle solver

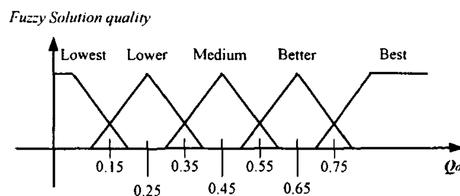

Fig. 8 Fuzzy partition of solution quality

For example, given the input of Fig. 7. The interface will show a dialog box as shown in Fig. 9 to confirm the user input as well as to give a prediction on the required time cost. The prediction is based on the QoS profile that is related to the given problem instance. If the user is not satisfied with this predicted result, he is allowed to choose another output quality level. Otherwise, the system uses this to actually produce an output that meets the user specification as shown in Fig. 10. Note that the QoS profile is used here to determine when to stop the system operation.

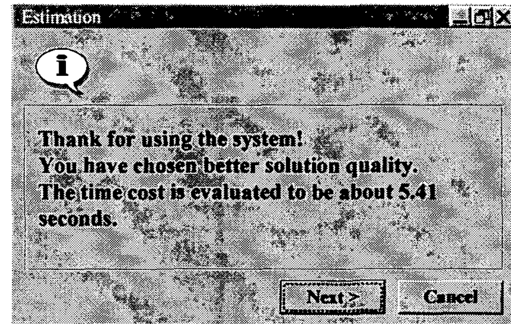

Fig. 9 User specification confirmation

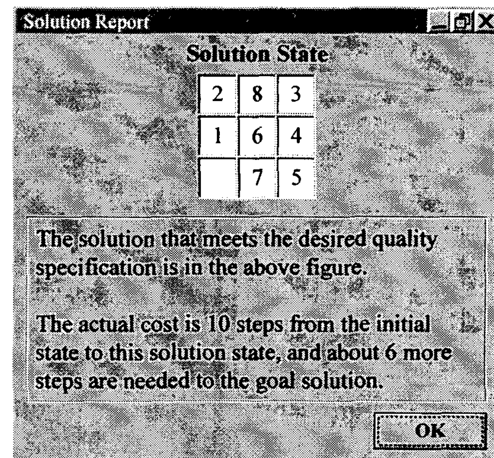

Fig. 10 Output of 8-puzzle solver

We randomly chose 649 8-puzzle instances and gave it to the 8-puzzle solver. Table 1 shows the test results on three quality levels. We define the success rate to be the percentage of instances that have returned output with quality the same as or better than the user-designated quality level. The table shows that the quality-driven feature of the system is quite successful.

Table 1 Experiment result

Quality Specification	Testing Instances	Success Instances	Success Ratio
better	649	600	92.44%
medium	649	572	88.13%
lowest	649	634	97.68%

4. Related works

Research projects related to our quality-driven system architecture are briefly discussed in this section. First, the EPIQ project [11][15] provides an end-to-end QoS management to facilitate the determination of change on the local task requirement corresponding to the QoS change from the remote task. The Cactus project [6] [7] introduces a micro-protocol to software architecture in order to provide the ability of system reconfiguration, and to explore the survivability of software under a dynamic environment. It provides some customizable, dynamic fine-grained attributes of quality of service for distributed systems, including dependability, real time, and security.

Self-adaptive software [8][14] evaluates and changes its own behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible. Progressive processing, proposed by Zilberstein [12][13][19], employs resource-bounded reasoning techniques and Markov decision theory to determine how to satisfy a set of requests under time pressure. It trades computational resources for quality of results. Value-driven information gathering [4][5] plans an information retrieval schedule according to the value of information about computational resource and cost. Finally, fault-tolerance systems [2][9][16] explore how to detect, repair or reconfigure, and recover from computational errors caused by either hardware or software fault.

5. Conclusion

We have described a new software architecture that can do quality-driven operations. The key concept of the architecture is that it contains multiple configurations so that it can adapt itself to fit various user requirements and constraints. The adaptation involves the tradeoff of a set of QoS factors in order to provide the best feasible performance. Such a quality-driven architecture is applicable in many domains such as multimedia (in video and audio quality control), searching, information gathering, decision-making (based on different quality of information source), image processing systems, etc. This quality-driving concept can be introduced to more domains; in fact, any resource-constrained or pay-per-use applications can benefit from the concept in providing a range of quality of services.

Two issues need to be further explored to make this architecture a popular one. First, the compilation of QoS profiles. The general compilation problem is NP-complete [20]. Our current application constructs a QoS profile for a system configuration off-line. An on-line QoS profile compiler is a necessity for a QDS that can dynamically construct a new configuration in order to cope with some unforeseeable quality requirement at run time. Second, the resource allocation control during reconfiguration. Our current application only involves the control of execution time. In a real application, both time and space controls are very important.

6. References

[1] C. Aurrecoechea, A. T. Campbell, and L. Hauw, "A Survey of QoS Architectures," *ACM/Springer Verlag Multimedia Systems Journal, Special Issue on QoS Architecture, vol. 6, no. 3*, pp. 138~151, May 1998.

[2] R. D. (Shawn) Blanton, S. C. Goldstein, and H. Schmit, "Tunable Fault Tolerance via Test and Reconfiguration," *Proc. Fault Tolerance Computing Symposium*, 1998. (available at http://www.ece.cmu.edu/research/piperench/pubs.html)

[3] M. Boddy, and T. L. Dean, "Solving Time-Dependent Planning Problems," *Proc. 7th International Joint Conference on Artificial Intelligence (IJCAI-89)*, pp. 979~984, Menlo Park, Calif., 1989.

[4] J. Grass and S. Zilberstein, A Value-Driven System for Scheduling Information Gathering. Technical Report 98-9, Computer Science Department, University of Massachusetts, Feb. 1998.

[5] J. Grass and S. Zilberstein, "Value-Driven Information Gathering," *Proc. AAAI Workshop on Building Resource-Bounded Reasoning Systems*, Providence, Rhode Island, 1997.

[6] M. Hiltunen, "Configuration Management for Highly-Customizable Software," *Proc. 4th International Conference on Configurable Distributed Systems*, pp. 197~205, May 1998.

[7] M. Hiltunen, R. Schlichting, and C. Ugarte, "Survivability Issues in Cactus," *Proc. 1998 Information Survivability Workshop*, pp. 85~88, Oct. 1998.

[8] G. Karsai, and J. Sztipanovits, "A Model-Based Approcah to Self-Adaptive Software," *IEEE Intelligent Systems, vol. 14, no. 3*, pp. 46~53, May/June 1999.

[9] R. J. Kreutzfeld, and R. E. Neese, "Methodology for Cost-Effective Software Fault Tolerance for Mission-Critical Systems," *IEEE Aerospace and Electronics Systems Magazine, vol. 12-9*, pp. 25~30, Sep. 1997.

[10] W. C. Lin, A User-Oriented Quality-Driven System Architecture. Master Thesis, Department of Electronic Engineering, National Taiwan University of Science and Technology, 2000.

[11] J. W. S. Liu, K. Nahrstedt, D. Hull, S. Chen, and B. Li, "EPIQ QoS Characterization (Draft Version)," July 1997. (available at http://pertsserver.cs.uiuc.edu/ epiq/files/qos-970722.ps)

[12] A. I. Mouaddib, and S. Zilberstein, "Optimal Scheduling of Dynamic Progressive Processing," *Proc. 13th Biennial European Conference on Artificial Intelligence (ECAI-98)*, Brighton, UK, 23~28 Aug. 1998. (available at http://anytime. cs.umass.edu/shlomo/papers/ecai98.html)

[13] A. I. Mouaddib, S. Zilberstein, and V. Danilchenko, "New Directions in Modeling and Control of Progressive Processing," *Proc. 13th Biennial European Conference on Artificial Intelligence (ECAI-98)*, Brighton, UK, 23~28 Aug. 1998. (available at http://anytime.cs.umass.edu/shlomo/ papers/ecai98ws.html)

[14] P. Robertson, and J. M. Brady, "Adaptive Image Analysis for Aerial Surveillance," *IEEE Intelligent Systems, vol. 14, no. 3,* pp. 30~36, May/June 1999.

[15] M. Shankar, M. DeMiguel, and J. W. S. Liu, "An End-to-End QoS Management Architecture," *Proc. Real-Time Applications Symposium*, June 1999. (available at http://pertsserver.cs.uiuc.edu/papers/ ShDeLi99.ps)

[16] X. Sun, and M. Rao, "Intelligent Multivariable Fault Tolerant Control Systems," *Proc. IEEE International Conference on Syst., Man & Cybe., vol. 2,* pp. 1406~1410, 1998.

[17] S. Zilberstein, "Using Anytime Algorithms in Intelligent Systems," *AI Magazine, vol. 17, no. 3,* pp. 73~83, 1996.

[18] S. Zilberstein, "Resource-Bounded Sensing and Planning in Autonomous Systems," *Autonomous Robots, vol. 3,* pp. 31~48, 1996. (available at http://anytime.cs.umass.edu/shlomo/papers/ar95.html)

[19] S. Zilberstein, and A. I. Mouaddib, "Reactive Control of Dynamic Progressive Processing," *Proc. 16th International Joint Conference on Artificial Intelligence (IJCAI-99),* Stockholm, Sweden, 1999. (available at http://anytime.cs.umass.edu/shlomo/ papers/ijcai99b.html)

[20] S. Zilberstein, and S. Russel, "Optimal Composition of Real-Time Systems," *Artificial Intelligence, vol. 82,* pp. 181~213, 1996.