Upright load allocation for Cloud Computing via various Performance Options

Amit Batra
Kurukshetra University
CSE Department,
H.C.T.M Kaithal, India.

Rajender Kumar
Kurukshetra University
CSE Department,
H.C.T.M., Kaithal, India

Arvind Kumar
Kurukshetra University
CSE Department,
H.C.T.M., Kaithal, India

ABSTRACT
Cloud computing looks to deliver software as a provisioned service to end users, but the underlying infrastructure must be sufficiently scalable and robust. In our work, we focus on large-scale enterprise cloud systems and examine ho-w enterprises may use a service-oriented architecture (SOA) to provide a streamlined interface to their business processes. To scale up the business processes, each SOA tier usually deploys multiple servers for load distribution and fault tolerance, a scenario which we term horizontal load distribution. One limitation of this approach is that load cannot be distributed further when all servers in the same tier are loaded. In complex multitiered SOA systems, a single business process may actually be implemented by multiple different computation pathways among the tiers, each with different components, in order to provide resilience and scalability. Such multiple implementation options gives opportunities for vertical load distribution across tiers. In this chapter, we look at a novel request routing framework for SOA-based enterprise computing with multiple implementation options that takes into account the options of both horizontal and vertical load distribution.

General Terms
Cloud Computing, Scheduling, Genetic Algorithm.

Keywords
Service oriented architecture (SOA), chromosome, GA(Genetic algorithm), servers per service time, server completion time(α).

1. INTRODUCTION
Cloud computing looks to have computation and data storage moved away from the end user and onto servers located in data
centers, thereby relieving users of the burdens of application provisioning and management (Dikaiakos, Pallis, Katsaros, Mehra, & Vakali, 2009; Cloud Computing, 2009). Software can then be thought of as purely a service that is delivered and consumed over the Internet, offering users the flexibility to choose applications on-demand and allowing providers to scale out their capacity accordingly. As rosy as this picture seems, the underlying server-side infrastructure must be sufficiently robust, feature-rich, and scalable to facilitate cloud computing. In this chapter we focus on large-scale enterprise cloud systems and examine how issues of scalable provisioning can be met using a novel load distribution system. In enterprise cloud systems, a service-oriented architecture (SOA) can be used to provide a streamlined interface to the underlying business processes being offered through the cloud. Such an SOA may act as a programmatic front-end to a variety of building-block components distinguished as individual services and their supporting servers (e.g. (DeCandia et al., 2007)). Incoming requests to the service provided by this composite SOA must be routed to the correct components and their respective servers, and such routing must be scalable to support a large number of
requests. In order to scale up the business processes, each tier in the system usually deploys multiple servers for load distribution and fault tolerance. Such load distribution across multiple servers within the same tier can be viewed as *horizontal* load distribution, as shown in Fig. 1. One limitation of horizontal load distribution is that load cannot be further distributed when all servers in the given tier are loaded as a result of mis-configured infrastructures – where too many servers are deployed at one tier while too few servers are deployed at another tier.

2. SCHEDULING COMPOSITE SERVICES
In this section, we formally define the problem and describe how we model its complexity. We assume the following scenario elements: *Requests* for a workflow execution are submitted to a scheduling agent. The workflow can be embodied by one of several *implementations*, so each request is assigned to one of these implementations by the scheduling agent. Each implementation invokes several *service types*, such as a web application server, a DBMS, or a computational analytics server. Each service type can be embodied by one of several *instances* of the service type, where each instance can have different computing requirements. For example, one implementation may require heavy DBMS computation (such as through a stored procedure) and light computational analytics, whereas another implementation may require light DBMS querying and heavy computational analytics. We assume that these implementations are set up by administrators or engineers. Each service type is executed on a *server* within a pool of servers dedicated to that service type Each service type can be served by a pool of servers. We assume that the servers make agreements to guarantee a level of performance defined by the completion time for completing a web service invocation. Although these SLAs can be complex, in this paper we assume for simplicity that the
guarantees can take the form of a linear performance degradation under load, an approach similar to other published work on service SLAs (e.g. (DeCandia et al., 2007)). We would like to ideally perform optimal scheduling to simultaneously distribute the load both vertically (across different implementation options) and horizontally (across different servers supporting a particular service type). There are thus two stages of scheduling, as shown in Fig. 2 In the first stage, the requests are assigned to the implementations. In the second stage each implementation has a known set of instances of a service type, and each instance is assigned to servers within the pool of servers for the instance's service type. Clearly, an exhaustive search through this solution space is prohibitively costly for all but the smallest configurations. In the next section we describe how we use a genetic search algorithm to look for the optimal scheduling assignments.

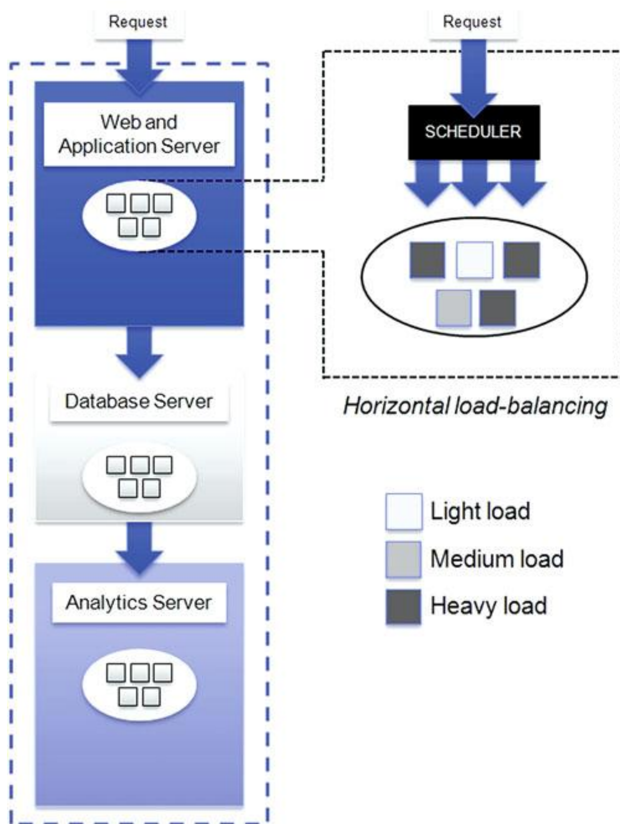

Figure: 1 Horizontal load distribution: load is distributed across a server pool within the same tier.

3. GENETIC ALGORITHM

A GA is a computer simulation of Darwinian natural selection that iterates through various generations to converge toward the best solution in the problem space. A potential solution to the problem exists as a chromosome, and in our case, a chromosome is a specific mapping of requests-to implementations and instances to- servers along with its associated workload execution time. Genetic algorithms are commonly used to find optimal exact solutions or near optimal approximations in combinatorial search problems such as the one we address. It is known that a GA provides a very good tradeoff between exploration of the solution space and exploitation of discovered maxima (Goldberg, 1989). Furthermore, a genetic algorithm does have an advantage of progressive optimization such that a solution is available at any time, and the result continues to improve as more time is given for optimization. Note that the GA is not guaranteed to find the optimal solution since the recombination and mutation steps are stochastic. Our choice of a genetic algorithm stemmed from our belief that other search heuristics (for example, simulated

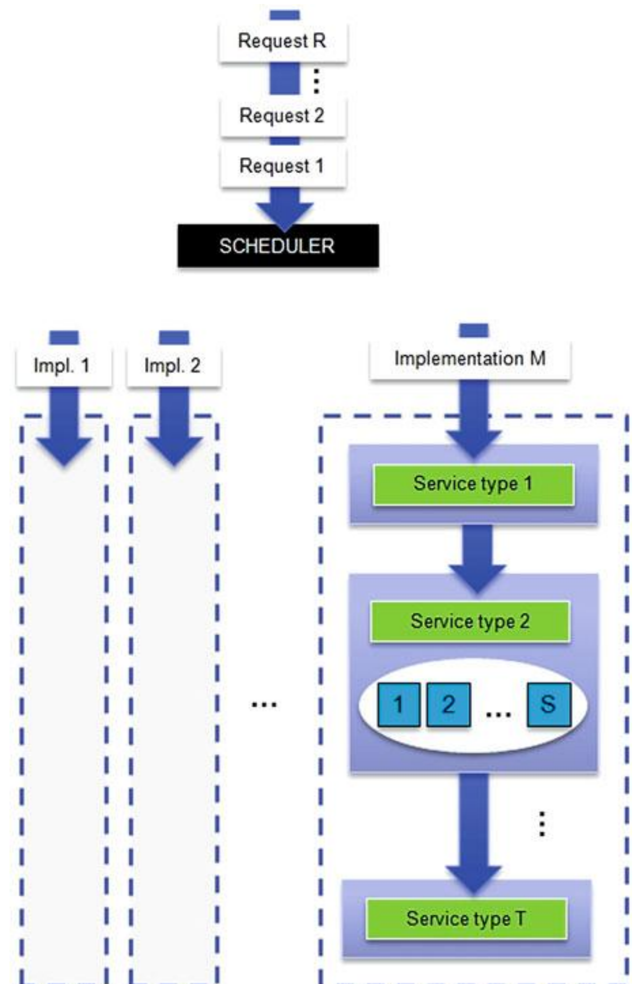

Figure: 2 The scheduling and assignment spans two stages. In the first stage, requests are assigned to implementations, and in the second stage, service type instances are assigned to servers.

annealing) are already along the same lines as a GA. These are randomized global search heuristics, and genetic algorithms are a good representative of these approaches. Prior research has shown there is no clear winner among these heuristics, with each heuristic providing better performance and more accurate results under different scenarios (Lima, Francois, Srinivasan, & Salcedo, 2004; Costa & Oliveira, 2001; Oliveira & Salcedo, 2005). Furthermore, from our own prior work, we are familiar with its operations and the factors that affect its performance and optimality convergence. Additionally, the mappings in our problem context are ideally suited to array and matrix representations, allowing us to use prior GA research that aid in chromosome recombination (Davis, 1985). There are other algorithms that we could have considered, but scheduling and assignment algorithms are a research topic unto themselves, and there is a very wide of range of approaches that we would have been forced to omit. Pseudo-code for a genetic algorithm is shown in Algorithm 1. The GA executes as follows. The GA produces an initial random population of chromosomes. The chromosomes then recombine (simulating sexual reproduction) to produce children using portions of both parents. Mutations in the children are produced with small probability to introduce traits that were not in either parent.The children with the best scores (in our case, the lowest workload execution times) are

chosen for the next generation. The steps repeat for a fixed number of iterations, allowing the GA to converge toward the best chromosome. In the end it is hoped that the GA explores a large portion of the solution space. With each recombination, the most beneficial portion of a parent chromosome is ideally retained and passed from parent to child,

Algorithm 1 Genetic Search Algorithm

```
1: FUNCTION Genetic algorithm
2: BEGIN
3: Time t
4: Population P(t) := new random Population
5:
6: while ! done do
7:     recombine and/or mutate P(t)
8:     evaluate(P(t))
9:     select the best P(t+1) from P(t)
10:    t := t + 1
11: end while
12: END
```

We used two data structures in a chromosome to represent each of the two scheduling stages. In the first stage, *R* requests are assigned to *M* implementations, so its representative structure is simply an array of size *R*, where each element of the array is in the range of [1,*M*], as shown in Fig. 3.

Figure 3 An example chromosome representing the assignment of *R* requests to *M* implementations.

The second stage where instances are assigned to servers is more complex. In Fig. 4 we show an example chromosome that encodes one scheduling assignment. The representation is a 2-dimensional matrix that maps {implementation, service type instance} to a service provider. For an implementation *i* utilizing service type instance *j*, the (*i*, *j*)*th* entry in the table is the identifier for the server to which the business process is assigned.

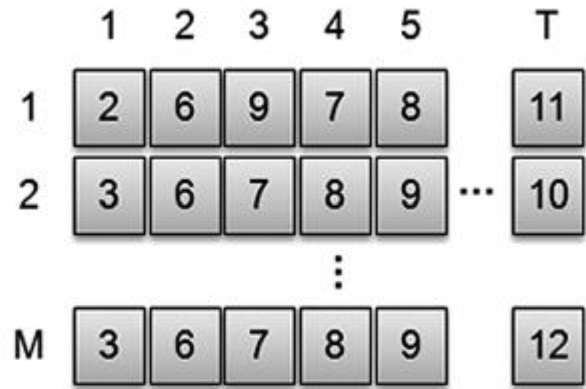

Figure 4 An example chromosome representing a scheduling assignment of (implementation, service type instance) service provider. Each row represents an implementation, and each column represents a service type instance. Here there are *M* workflows and *T* service types instances. In workflow 1, any request for service type 3 goes to server 9.

4. CHROMOSOME RECOMBINATION

Two parent chromosomes recombine to produce a new child chromosome. The hope is that the child contains the best contiguous chromosome regions from its parents. Recombining the chromosome from the first scheduling stage is simple since the chromsomes are simple 1-dimensional arrays. Two cut points are chosen randomly and applied to both the parents. The array elements between the cut points in the first parent are given to the child, and the array elements outside the cut points from the second parent are appended to the array elements in the child. This is known as a 2-point crossover and is shown in Fig. 5. For the 2-dimensional matrix, chromosome recombination was implemented by performing a one-point crossover scheme twice (once along each dimension). The crossover is best explained by analogy to Cartesian space as follows. A random location is chosen in the matrix to be coordinate (0, 0).

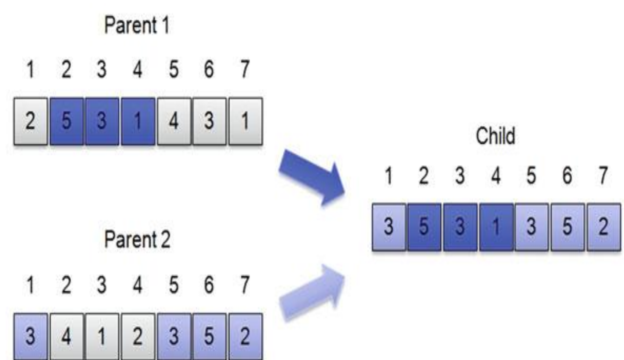

Figure :5 An example recombination between two parents to produce a child for the first stage assignments. This recombination uses a 2-point crossover recombination of two one-dimensional arrays. Contiguous subsections of both parents are used to create the new child.

5. GA EVALUATION FUNCTION

The evaluation function returns the resulting workload execution time given a chromosome. Note the function can be mplemented to evaluate the workload in any way so long as it is consistently applied to all chromosomes across all generations. Our evaluation function is shown in Algorithm 2. In lines 6–8, it initialises the execution times for all the servers in the chromosome. In lines 11–17, it assigns requests to implementations and service type instances to servers using the two mappings in the chromosome. The end result of this phase is that the instances are accordingly enqueued the servers. In lines 19–21 the running times of the servers are calculated. In lines 24–26, the results of the servers are used to compute the results of the implementations. The function returns the maximum execution time among the implementations.

6. HANDLING ONLINE ARRIVING REQUESTS

As mentioned earlier, the problem domain we consider is that of batch-arrival request routing. We take full advantage of such a scenario through the use of the GA, which has knowledge of the request population. We can further extend this approach to online arriving requests, a lengthy discussion which we omit here due to space limits. A typical approach is to aggregate the incoming requests into a queue, and when a designated timer expires, all requests in the queue at that time are scheduled. There may still be uncompleted requests from the previous execution, so the requests may be mingled together to produce a larger schedule. An alternative approach is to use online stochastic optimization techniques commonly found in online decision-making systems (Van Hentenryck & Bent, 2006). First, we can continue to use the GA, but instead of having the complete collection of requests available to us, we can allow requests to aggregate into a queue first. When a periodic timer expires, we can run the GA on those requests while aggregating any more incoming requests into another queue. Once the GA is finished with the first queue, it will process the next queue when the periodic timer expires again. If the request arrival rate is faster than the GA's processing rate, we can take advantage of the fact that the GA can be run as an incomplete, near-optimal search heuristic: we can go ahead and let the timer interrupt the GA, and the GA will have ∗some∗ solutions that, although sub-optimal, is probabilistically better than a greedy solution. This typical methodology is also shown in (Dewri, Ray, Ray, &Whitley, 2008), where requests for broadcast messages are queued, and the messages are optimally distributed through the use of an evolutionary strategies algorithm (a close cousin of a genetic algorithm). Second (and unrelated to genetic algorithms), we can use online stochastic optimization techniques to serve online arrivals. This approach approximates the offline problem by sampling historical arrival data in order to make the best online decision. A good overview is provided in (Bent & Van Hentenryck, 2004). In this technique, the online optimizer receives an incoming sequence of requests, gets historical data over some period of time from a sampling function that creates a statistical distribution model, and then calculates and returns an optimized allocation of requests to available resources. This optimization can be done on a periodic or continuous basis.

Algorithm 2 GA evaluation function

```
1: FUNCTION evaluate
2: IN: CHROMOSOME, a representation of the assignments of requests to implementation and
   service type instances to servers
3: OUT: runningtime, the running time of this workload
4: BEGIN
5:
6: for (each server ∈ CHROMOSOME) do
7:    set server's running time to 0
8: end for
9:
10: {Loop over each request and its implementations}
11: for (each request ∈ CHROMOSOME) do
12:    implementation := requests's implementation
13:    for (each instance ∈ implementation) do
14:       server := implementation's server
15:       Enqueue this job at server
16:    end for
17: end for
18:
19: for (each server) do
20:    Compute the running time of server
21: end for
22:
23: {Now compute the running time of the implementations}
24: for (each implementation ∈ CHROMOSOME) do
25:    Aggregate the running time of this implementation across its instances
26: end for
27:
28: runningtime := maximum running time of each implementation
29: return runningtime
30: END
```

7. EXPERIMENTS AND RESULTS

We ran experiments to show how our system compared to other well-known algorithms with respect to our goal of providing request routing with horizontal and vertical distribution. Since one of our intentions was to demonstrate how our system scales well up to 1000 requests, we used a synthetic workload that allowed us to precisely control experimental parameters, including the number of available implementations, the number of published service types, the number of service type instances per implementation, and the number of servers per service type instance. The scheduling and execution of this workload was simulated using a program we implemented in standard C++. The simulation ran

on an off-the-shelf Red Hat Linux desktop with a 3.0 GHz Pentinum IV and 2 GB of RAM. In these experiments we compared our algorithm against the following alternatives: _A *round-robin* algorithm that assigns requests to an implementation and service type instances to a server in circular fashion. This well-known approach provides a fast and simple scheme for load-balancing. A *random-proportional* algorithm that proportionally assigns instances to the servers. For a given service type, the servers are ranked by their guaranteed completion time, and instances are assigned proportionally to the servers based on the servers' completion time. (We also tried a proportionality scheme based on both the completion times and maximum concurrency but attained the same results, so only the former scheme's results are shown here.) To isolate the behavior of this proportionality scheme in the second phase of the scheduling, we always assigned the requests to the implementations in the first phase using a round-robin scheme. A *purely random* algorithm that randomly assigns requests to an implementation and service type instances to a server in random fashion. Each choice was made with a uniform random distribution. A *greedy* algorithm that always assigns business processes to the service provider that has the fastest guaranteed completion time. This algorithm represents a naïve approach based on greedy, local observations of each workflow without taking into consideration all workflows. In the experiments that follow, all results were averaged across 20 trials, and to help normalize the effects of any randomization used during any of the algorithms, each trial started by reading in pre-initialized data from disk. In Table 1 list our experimental parameters for our baseline experiments.We vary these parameters in other experiments, as we discuss later.

8. CONCLUSIONS

Cloud computing aims to do the dirty work for the user: by moving issues of management and provisioning away from the end consumer and into the server-side data centers, users are given more freedom to pick and choose the applications that suit their needs. However, computing in the clouds depends heavily on the scalability and robustness of underlying cloud architecture. We discussed enterprise cloud computing where enterprises may use a service oriented architecture to publish a streamlined interface to their business processes.

Table 1. Experiment Parameters

Experimental parameter	**Comment**
Requests	1 to 1000
Implementations	5, 10, 20
Service types used per Implementation	uniform random: 1 – 10
Instances per service type	uniform random: 1 – 10
Servers per service type	uniform random: 1 – 10
Server completion time (α)	uniform random: 1 – 12 seconds
Server maximum concurrency (β)	uniform random: 1 – 12
Server degradation coefficient (γ)	uniform random: 0.1 – 0.9
GA: population size	100
GA: number of generations	200

In order to scale up the number of business processes, each tier in the provider's architecture usually deploys multiple servers for load distribution and fault tolerance. Such load distribution across multiple servers within the same tier can be viewed as *horizontal* load distribution. One limitation of this approach is that load cannot be distributed further when all servers in the same tier are fullyloaded. Another approach for providing resiliency and scalabilty is to have *multiple implementation options* that give opportunities for *vertical* load distribution across tiers. We described in detail a request routing framework for SOAbased enterprise cloud computing that takes into account both these options for *horizontal* and *vertical* load distribution. Experiments showed that our algorithm and methodology can scale well up to a large-scale system configuration comprising up to 1000 workflow requests directed to a complex composite service with multiple implementation options available. The experimental results also demonstrate that our framework is more agile in the sense that it is effective in dealing with mis-configured infrastructures in which there are too many or too few servers in one tier. As a result, our framework can effectively utilize available multiple implementations to distribute loads across tiers.

9. ACKNOWLEDGMENTS

We would like to express our gratitude to all those who gave us the possibility to complete this paper. Furthermore we would like to thank Department of Computer Science, Punjabi University Patiala for giving this esteemed opportunity for publishing this paper. We would also like to thank Director, H.C.T.M, Kaithal, as well as Head of Department(CSE), H.C.T.M., Kaithal for their collaborat-ion and helping us to make resources availbility.

10. REFERRENCES

[1] Bent, R., & Van Hentenryck, P. (2004). Regrets Only! Online stochastic optimization under time constraints. *Nineteenth National Conference on Artificial Intelligence*, *SanJose, CA*.

[2] Buco, M. J., Chang, R. N., Luan, L. Z., Ward, C., Wolf, J.L., Yu, P. S., et al. (2003). Managing ebusiness on demand slacontracts in business terms using the cross-sla executionmanager sam. *ISADS, Washington, DC*, 157–164.

[3] *Business Process Execution Language for Web Services* (Version 1.1), (2005).

[4] Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., & Shan,M.-C. (2000). Adaptive and Dynamic Service

Composition ineFlow. *Proceedings of CAISE, Stockholm, Sweden*, 13–31.

[5] Costa, L., & Oliveira, P. (2001). Evolutionary algorithms approach to the solution of mixed integer nonlinearprogramming problems. *Computers and ChemicalEngineering, 25*(2–3), 257–266.

[6] Davis, L. (1985). Job shop scheduling with geneticalgorithms. *Proceedings of the International Conference on Genetic Algorithms*, *Pittsburgh, PA*.

[7] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P.& Vogels, W. (2007). Dynamo: Amazon’s highly available key-value store. *Proceedings of SOSP, Washington D.C.,*205–220.

[8] Dewri, R., Ray, I., Ray, I., & Whitley, D. (2008). Optimizing on-demand data broadcast scheduling in pervasiveenvironments. *Proceedings of EDBT, Nantes, France*, 559–569.

[9] Dikaiakos, M., Pallis, G., Katsaros, D., Mehra, P., & Vakali, A. (2009). Cloud computing: Distributed internet computing for it and scientific research. *IEEE Internet Computing, 13*(5), 10–13.

[10] Goldberg, D. (1989). Genetic algorithms in searth, optimization, and machine learning. Dordrecht: Kluwer.

[11] Holland, J. (1992). *Adaptation in natural and artificial systems*. Cambridge, MA: MIT Press.

[12] Lima, R., Francois, G., Srinivasan, B., & Salcedo, R. (2004). Dynamic optimization of batch emulsion polymerization using MSIMPSA, a simulated-annealingbasedalgorithm. *Industrial and Engineering ChemistryResearch, 43*(24), 7796–7806.

[13] Oliveira, R., & Salcedo, R. (2005). Benchmark testing of simulated annealing, adaptive random search and genetic algorithms for the global optimization of bioprocesses. *International Conference on Adaptive and Natural Computing Algorithms, Coimbra, Portugal*.

[14] Phan, T., & Li, W.-S. (2008a). Dynamic materialization of query views for data warehouse workloads. *Proceedings ofthe International Conference on Data Engineering, Long Beach, CA*.

[15] Ponnekanti, S., & Fox, A. (2004). Interoperability among Independently Evolving Web Services. *Proceedings ofMiddleware, Toronto, Canada*.

[16] Shankar, M., De Miguel, M., & Liu, J. W.-S. (2004). An end-to-end qos management architecture *Proceedings of theFifth IEEE Real Time Technology and Applications* Symposium, Vancouver, British Columbia, Canada, p. 176.