# Expressividade da RTSJ para Implementar Computação Imprecisa

Patricia Plentz, Rodrigo Gonçalves, Guilherme Moreira, Rômulo de Oliveira, Carlos Montez
(plentz, rpg, gmoreira, romulo, montez)@das.ufsc.br

LCMI – Depto de Automação Sistemas – Univ. Fed. de Santa Catarina
Caixa Postal 476 – 88040-900 – Florianópolis – SC – Brasil

**Resumo**

*A RTSJ – Real-Time Specification for Java – é uma extensão da linguagem Java para a programação de aplicações com requisitos temporais. Nos sistemas de tempo real é preciso atender deadlines, os quais são impostos pelo ambiente da aplicação. A dificuldade encontrada no escalonamento tempo real levou alguns autores a propor uma técnica chamada de Computação Imprecisa, que sacrifica a qualidade dos resultados para cumprir os deadlines. Este artigo investiga a expressividade da RTSJ na implementação de Computação Imprecisa. Para isto, as três formas tradicionais de programação da computação imprecisa são implementadas utilizando recursos específicos da RTSJ.*

***Palavras-chave:*** *tempo real, computação imprecisa, Java tempo real.*

## 1. Introdução

Os sistemas em geral, que não apresentam restrições de tempo real, são caracterizados por uma abordagem do tipo "fazer o trabalho usando o tempo que for necessário". Já os sistemas tempo real possuem uma abordagem diferente, pois é preciso garantir que será possível atender prazos, impostos pelo ambiente do sistema. Logo, a preocupação é "garantir que o trabalho será concluído no tempo disponível".

A dificuldade encontrada no escalonamento tempo real levou alguns autores a proporem uma abordagem diferente, do tipo "fazer o trabalho possível dentro do tempo disponível". Isto significa sacrificar a qualidade dos resultados para poder cumprir os prazos exigidos. Essa técnica, conhecida pelo nome de Computação Imprecisa [1], flexibiliza o escalonamento.

Recentemente foi concluída a especificação de uma extensão da linguagem Java capaz de permitir a programação de aplicações com elevado determinismo temporal. Essa especificação recebeu o nome de RTSJ – *Real-Time Specification for Java*. Atualmente é disponibilizada pela empresa Timesys uma implementação de referência para a RTSJ.

Este artigo investiga a expressividade da RTSJ na implementação de Computação Imprecisa. Para isto, as três formas tradicionais de programação da Computação Imprecisa são implementadas em Java de tempo real, utilizando recursos específicos da RTSJ.

Este texto é dividido em cinco seções. Na seqüência a esta introdução, a seção 2 descreve suscintamente a RTSJ. A seção 3 caracteriza a Computação Imprecisa e descreve as formas básicas de programação. A seção 4 apresenta implementações de Computação Imprecisa em RTSJ. Finalmente, as conclusões e comentários finais aparecem na seção 5.

## 2. RTSJ (*Real-Time Specification for Java*)

A popularidade da linguagem de programação Java motivou a criação de extensões tempo real para a mesma. Em 1998 foi criado o primeiro grupo de trabalho, através do *Java Experts Group* – JEG, com objetivo de definir uma especificação tempo real para Java. O JEG liberou a versão final da RTSJ em 2000.

A RTSJ define uma API para a linguagem Java que permite a criação, verificação, análise, execução e gerenciamento de *threads* tempo real, procurando satisfazer seus requisitos temporais. Nenhuma alteração sintática foi realizada na linguagem Java. Também

foi mantida a compatibilidade entre a RTSJ e aplicações escritas para Java padrão. Assim, essas aplicações não devem sofrer restrições de execução em implementações da RTSJ.

Essa especificação introduz o conceito de *objeto escalonável*. Um objeto escalonável pode ser uma *thread* tempo real, ou uma *thread* tempo real não-*heap* ou ainda um tratador de eventos assíncronos. A RTSJ também acrescenta à especificação Java existente os seguintes itens [2]:

- *Threads* Tempo Real. Estas *threads* possuem atributos de escalonamento que permitem definir tarefas tempo real (período de execução, *deadline*, tratadores de perda de deadline, prioridade, etc).
- Mecanismos que ajudam a escrita de código Java sem a utilização do coletor de lixo.
- Uma classe que trata eventos assíncronos e um mecanismo que associa eventos assíncronos com acontecimentos fora da JVM. Cada evento é atendido por uma *thread*, criada especialmente para essa função. A classe *tratadores de eventos assíncronos* realiza a criação da *thread* e o atendimento do evento assíncrono, com o objetivo de que a criação dessa *thread* não penalize o desempenho do sistema.
- Um mecanismo chamado *transferência de controle assíncrona*, que permite uma *thread* mudar o fluxo de controle em outra *thread*. Tal mecanismo é uma forma bastante controlada de uma *thread* lançar uma exceção em outra *thread*.
- Mecanismos que permitem os programadores acessar memória em endereços particulares.
- Mecanismos que permitem o programador controlar onde os objetos serão alocados na memória. A RTSJ criou dois novos domínios de alocação de memória para *threads* acessarem objetos: memória permanente e memória em escopos [3].

## 3. Computação Imprecisa

Computação Imprecisa está fundamentada na idéia de que cada tarefa do sistema possui uma parte obrigatória (*mandatory*) e uma parte opcional (*optional*). A parte obrigatória da tarefa é capaz de gerar um resultado com a qualidade mínima, necessária para manter o sistema operando de maneira segura. A parte opcional refina este resultado, até que ele alcançe a qualidade máxima. O resultado da parte obrigatória é dito impreciso (*imprecise result*), enquanto o resultado das partes obrigatória+opcional é dito preciso (*precise result*). Uma tarefa é chamada de tarefa imprecisa (*imprecise task*) se for possível decompô-la em parte obrigatória e parte opcional. O modelo de tarefas associado com Computação Imprecisa não exclui a existência de tarefas somente com uma das partes. Cabe à semântica da aplicação definir quais são as partes obrigatórias e quais são as opcionais.

Em situações normais, tanto a parte obrigatória quanto a parte opcional são executadas e o sistema gera resultados com a precisão desejada. Se, por algum motivo, não for possível executar todas as tarefas do sistema, algumas partes opcionais serão deixadas de lado. Este mecanismo permite uma degradação controlada do sistema, na medida em que pode-se determinar o que não será executado em caso de sobrecarga.

Tarefas não são completamente descartadas, mas a qualidade do resultado é parcialmente sacrificada. Isto gera uma redução na demanda por recursos que permite o atendimento dos deadlines.

### 3.1 Formas de programação

Existem três formas básicas usadas na programação de tarefas imprecisas: a programação pode ser feita com funções monotônicas, funções de melhoramento ou múltiplas versões.

As funções monotônicas (*monotone functions*) são aquelas cuja qualidade do resultado aumenta (ou pelo menos não diminui) à medida que o tempo de execução da função aumenta. As computações necessárias para obter-se um nível mínimo de qualidade correspondem à parte obrigatória. Qualquer computação além desta será incluída como parte opcional. O nível

mínimo de qualidade deve garantir uma operação segura do sistema, enquanto a parte opcional refina progressivamente o resultado da tarefa. Neste tipo de função o escalonador deve decidir quanto tempo de processador cada parte opcional deve receber.

As funções de melhoramento (*sieve functions*) são aquelas cuja finalidade é produzir saídas no mínimo tão precisas quanto as correspondentes entradas. O valor de entrada é melhorado de alguma forma pela função. Se o resultado recebido como entrada por uma função de melhoramento é aceitável como saída, então a função pode ser completamente omitida (não executada). As funções de melhoramento normalmente formam partes opcionais que seguem algum cálculo obrigatório. Tipicamente, não existe benefício em executar uma função de melhoramento parcialmente. Isto significa que o escalonador deve optar, antes de iniciar a tarefa, em executá-la completamente ou não executá-la.

Uma tarefa imprecisa também pode ser implementada através de múltiplas versões (*multiple versions*). Normalmente são empregadas duas versões. A versão primária gera um resultado preciso, porém possui um tempo de execução no pior caso desconhecido ou muito grande. A versão secundária gera um resultado impreciso, porém seguro para o sistema, em um tempo de execução menor e conhecido. A cada ativação da tarefa, cabe ao escalonador escolher qual versão será executada. No mínimo, deve ser garantido tempo de processador para a execução da versão secundária, a qual corresponde a parte obrigatória. A parte opcional é definida pela diferença entre os tempos máximos de execução das versões primária e secundária. Esta técnica é a mais flexível do ponto de vista da programação.

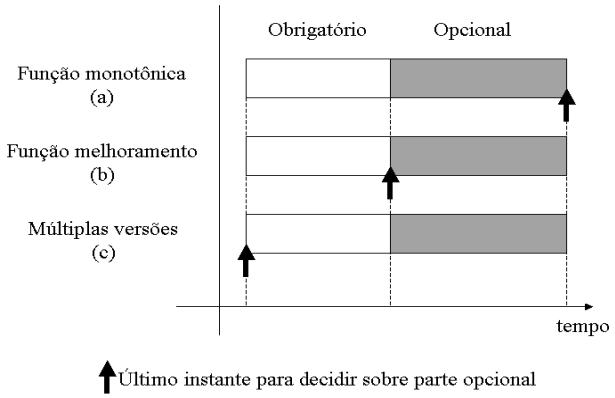


**Figura 1. Escalonamento de tarefas imprecisas.**

A forma de programação empregada interfere no comportamento do escalonador. Quando uma função monotônica é empregada, o tempo de processador alocado à parte opcional pode ser qualquer valor entre zero e o tempo máximo de execução da parte opcional. Qualquer tempo de processador fornecido ajuda a melhorar a qualidade do resultado. Além disto, a decisão de interromper a execução da parte opcional pode ser tomada a qualquer instante, mesmo quando esta já estiver executando (Figura 1-a).

Quando a parte opcional executa uma função de melhoramento, não existe benefício em executá-la parcialmente. O escalonador deve decidir se executa a função de melhoramento completamente ou a descarta. Esta característica é chamada na bibliografia de restrição 0/1 (*0/1 constraint*). Além disto, a decisão deverá ser tomada, no mais tardar, quando a parte obrigatória é concluida e a parte opcional deve (ou não) ser iniciada. Uma vez iniciada a execução da parte opcional, ela é executada até o final (Figura 1-b).

Múltiplas versões também criam uma restrição 0/1. O escalonador é obrigado a executar completamente a parte opcional (escolhendo a versão primária) ou descartá-la completamente (escolhendo a versão secundária). Esta decisão deve ser tomada antes de iniciar a parte

obrigatória da tarefa pois, uma vez escolhida a versão, a execução ou não da parte opcional já estará automaticamente definida (Figura 1-c).

## 4. Programação de Computação Imprecisa usando RTSJ

As três formas de tarefa imprecisa, descritas na seção anterior, podem ser implementadas com relativa facilidade em RTSJ através de simples disciplina de programação. No programa exemplo descrito a seguir serão feitas referências a uma classe Escalonador. Esta classe é responsável por embutir o algoritmo de escalonamento que decide qual a precisão de cada ativação de cada tarefa. Em função da limitação de espaço, não será discutida a seleção do algoritmo a ser usado, entre os diversos presentes na literatura ([4], [5], etc).

### 4.1 Parte inicial do código

Conforme pode ser visto na Listagem 1, no programa exemplo serão criadas três *threads* de tempo real, e cada uma empregará uma das formas de programação da CI.

```
01: import javax.realtime.*;
02: public class CI {
03:   public static void main(String[] args) {
04:     ThreadMultVersoes  tmv = new ThreadMultVersoes();
05:     ThreadMelhoramento tmo = new ThreadMelhoramento();
06:     ThreadMonotonica   tmn = new ThreadMonotonica();

07:     tmv.start(); tmo.start(); tmn.start();
08:     try {
09:           tmv.join(); tmo.join(); tmn.join();
10:     } catch (Exception e) {
11: }}}
```

**Listagem 1. Cria *threads* que representam formas de tarefa imprecisa.**

A Listagem 2 mostra a classe que trata o evento assíncrono associado com a perda de um deadline. Cada thread da aplicação usa sua própria instância desta classe. Este tratador avisa a classe Escalonador da perda e arma a próxima ocorrência periódica da *thread* da aplicação.

```
01: public class MissHandler extends AsyncEventHandler {
02:   String thread;
03:   RealtimeThread me;

04:   public MissHandler(RealtimeThread th, String t) {
05:     me = th; thread = t;
06:   }

07:   public void handleAsyncEvent() {
08:     RealtimeThread rtt = RealtimeThread.currentRealtimeThread();
09:     Tela.println("Thread: " + me.getClass().toString() + "perd dead.");
10:     Escalonador.perdeDeadline(me, thread);
11:     me.schedulePeriodic();
12:   } };
```

**Listagem 2. Tratador de evento assíncrono.**

### 4.2 Função monotônica

A função monotônica pode ser implementada de várias formas. Por exemplo, é possível permitir a classe Escalonador usar o método *fire()* para interromper as iterações quando o tempo disponível tiver acabado. Neste caso, a classe que contém o código iterativo deve implementar a interface *Interruptible*. Um objeto *AsynchronouslyInterruptedException* é usado para, via método *doInterruptible*, encapsular a execução monotônica (Listagem 3).

```
01: public class ThreadMonotonica extends RealtimeThread {
02:   private AsynchronouslyInterruptedException aie;
03:   private ExemploFuncaoMonotonica monotonica;
04:   private boolean dead = true;
05:   public ThreadMonotonica() { super();
06:     // Declara ReleaseParameter c/ parametros (periodica, periodo,deadl., ...)
07:     ReleaseParameters release = new PeriodicParameters(
08:       new RelativeTime(1000, 0),      // inicio
09:       new RelativeTime(5000, 0),      // periodo de 5 segundos
10:       new RelativeTime(1000, 0),      // custo
11:       new RelativeTime(4000, 0),      // deadline
12:       null,                           // gerenciador de excesso de execucao
13:       new MissHandler(this, "ThreadMonotonica")); //gerenc. perda de deadline
14:     // Declara o SchedulingParameter com a prioridade
15:     SchedulingParameters sched  = new
                        PriorityParameters(PriorityScheduler.MIN_PRIORITY + 12);
16:     // Atribui os parametros da thread
17:     setSchedulingParameters(sched); setReleaseParameters(release);
18:   }
19:   public AsynchronouslyInterruptedException getAIE() {
20:     return aie;
21:   }
22:   public void run() {
23:     aie        = new AsynchronouslyInterruptedException();
24:     monotonica = new ExemploFuncaoMonotonica();
25:     while(true) {
26:       do {
27:           if(dead)  // Avisa o escalonador que cumpriu o deadline
28:             Escalonador.cumpreDeadline(this, "ThreadMonotonica");
29:           aie.doInterruptible(monotonica);
30:           Tela.println("Resultado: " + monotonica.getMelhorValor());
31:       } while(dead = waitForNextPeriod());
32:     }}};
```

**Listagem 3. Implementação da função monotônica.**

A Listagem 4 mostra a classe que contém a computação a ser feita. Por implementar a interface *Interruptible*, ela oferece um método *run()* para execução e um método *interruptAction()* para quando o *fire()* for disparado pela classe Escalonador. O melhor valor é posteriormente obtido via *getMelhorValor()*.

```
01: public class ExemploFuncaoMonotonica implements Interruptible {
02:   private long melhorValor = -1;
03:   public void run(AsynchronouslyInterruptedException a)
                                    throws AsynchronouslyInterruptedException {
04:     melhorValor = 0;
05:     Escalonador.inicioMonotonica(RealtimeThread.currentRealtimeThread());
06:     while(true)
07:         melhorValor = funcao_que_refina_valor(melhorValor);
08:   }
09:   public void interruptAction(AsynchronouslyInterruptedException a) {
10:      // Deixa o estado da classe consistente
11:   }
12:   public long getMelhorValor() {
13:      return melhorValor; // Retorna o melhor valor calculado pela função
14:   }};
```

**Listagem 4.Exemplo de função monotônica.**

### 4.3 Função melhoramento

Uma função melhoramento pode ser implementada facilmente através de uma consulta a classe Escalonador e chamada, ou não, da função que refina o resultado obtido. O código da Listagem 5 mostra construção da thread periódica.

```
01: public class ThreadMelhoramento extends RealtimeThread {
02:   private ExemploFuncaoMelhoramento melhoramento;
03:   private boolean dead = false;
04:   public ThreadMelhoramento() {  super();
05:     // Declara ReleaseParameter c/ parametros (periodica, periodo, deadl.,...)
06:     ReleaseParameters release = new PeriodicParameters(
07:       new RelativeTime(1000, 0),         // inicio
08:       new RelativeTime(5000, 0),         // periodo de 5 segundos
09:       new RelativeTime(1000, 0),         // custo
10:       new RelativeTime(4000, 0),         // deadline
11:       null,                              // gerenciador de excesso de execucao
12:       new MissHandler(this, "ThreadMelhoramento")); //gerenc. de perda deadline
13:     // Declara SchedulingParameter com a prioridade
14:     SchedulingParameters sched  = new
                        PriorityParameters(PriorityScheduler.MIN_PRIORITY + 11);
15:     // Atribui os parametros da thread
16:     setSchedulingParameters(sched); setReleaseParameters(release);
17:   }
18:   public void run() {
19:     melhoramento = new ExemploFuncaoMelhoramento();
20:     while(true) {
21:       do {
22:         if(dead) // Avisa ao escalonador
23:             Escalonador.cumpreDeadline(this, "ThreadMelhoramento");
24:         Tela.println("Resultado: " + melhoramento.calculaValor(10));
25:       }while(dead = waitForNextPeriod());
26:     }}};
```

**Listagem 5. Função melhoramento.**

A Listagem 6 mostra a classe que contém o mecanismo de melhoramento.

```
01: public class ExemploFuncaoMelhoramento {
02:   private long refinaCalculoValor(long x) {
03:     return executa_mais_um_processamento(x);
04:   }
05:   private long calculaValorAproximado(long x) {
06:     return executa_algum_processamento(x);
07   }
08:   public long calculaValor(long x) {
09:     long valorImpreciso = calculaValorAproximado(x);
10:     if(Escalonador.fazPreciso(RealtimeThread.currentRealtimeThread()))
11:         return refinaCalculoValor(valorImpreciso);
12:     else
13:         return valorImpreciso;
14:    }};
```

**Listagem 6. Exemplo de função melhoramento.**

### 4.4 Múltiplas versões

Múltiplas versões podem ser implementadas através de um método público que, ao invés de conter a lógica da aplicação, apenas consulta a classe Escalonador e então desvia a execução para a versão apropriada. O código Java da Listagem 7 mostra a construção da thread.

```
01: public class ThreadMultVersoes extends RealtimeThread {
02:   private ExemploFuncaoMultVersoes multiplasVersoes;
03:   private boolean dead = false;
04:   public ThreadMultVersoes() { super();
05:     // Declara ReleaseParameter c/ parametros (periodica, periodo, ...)
06:     ReleaseParameters release = new PeriodicParameters(
```

```
07:     new RelativeTime(1000, 0),      // inicio
08:     new RelativeTime(5000, 0),      // periodo de 5 segundos
09:     new RelativeTime(1000, 0),      // custo
10:     new RelativeTime(4000, 0),      // deadline
11:     null,                           // gerenciador de excesso de execucao
12:     new MissHandler(this, "ThreadMultiplasVersoes")); // gerenc. perda deadline
13:     // Declara o SchedulingParameter com a prioridade
14:     SchedulingParameters sched = new
                        PriorityParameters(PriorityScheduler.MIN_PRIORITY + 10);
15:     // Atribui os parametros da thread
16:     setSchedulingParameters(sched); setReleaseParameters(release);
17:   }
18:   public void run(){
19:     multiplasVersoes = new ExemploFuncaoMultVersoes();
20:     while(true) {
21:       do {
22:         if(dead)  // Avisa ao escalonador
23:           Escalonador.cumpreDeadline(this, "ThreadMultiplasVersoes");
24:         Tela.println("Resultado: " + multiplasVersoes.calculaValor(10));
25:       }while(dead = waitForNextPeriod());
26:     } } };
```

**Listagem 7. Múltiplas versões.**

A Listagem 8 apresenta a classe que implementa as múltiplas versões.

```
01: public class ExemploFuncaoMultVersoes {
02:   private long calculaValorPreciso(long x) {
03:     return executa_longo_processamento(x);
04:   }
05:   private long calculaValorImpreciso(long x) {
06:     return executa_rapido_processamento(x);
07:   }
08:   public long calculaValor(long x) {
09:     if(Escalonador.fazPreciso(RealtimeThread.currentRealtimeThread()))
10:         return calculaValorPreciso(x);
11:     else
12:         return calculaValorImpreciso(x);
13:   }};
```

**Listagem 8. Exemplo de múltiplas versões.**

### 4.5 Classes Escalonador e ThreadFire

A classe Escalonador é mostrada na Listagem 9. Ela é responsável por decidir qual deve ser o nível de precisão de cada ativação de tarefa imprecisa. O Escalonador embute portanto um algoritmo de escalonamento adaptativo, que executa sobre o escalonador nativo do RTSJ, o qual emprega prioridades fixas preemptivas. Algoritmos adaptativos apropriados para Computação Imprecisa não serão discutidos neste artigo.

```
01: public class Escalonador {
02:   private static RealtimeThread rtt;
03:   private static ThreadFire tf;
04:   public synchronized static void inicioMonotonica(RealtimeThread t) {
05:     rtt = t;
06:     // Instancia local realtimeThread
07:     tf = new ThreadFire();
08:     tf.start();
09:   }
10:   public synchronized static void fimMonotonica() {
11:     if(((ThreadMonotonica)rtt).getAIE().fire())
12:       Tela.println("Método interrompido.");
13:     else
```

```
14:        Tela.println("Método não está sendo executado.");
15:     }
16:     public synchronized static boolean fazPreciso(RealtimeThread th) {
17:       return true;
18:     }
19:     public synchronized static void perdeDeadline(RealtimeThread th, String t) {
20:       Tela.println("PERDE THREAD: " + t);
21:     }
22:     public synchronized static void cumpreDeadline(RealtimeThread th, String t){
23:       Tela.println("CUMPRE THREAD: " + t);
24:     }};
```

**Listagem 9. Classe Escalonador.**

A classe Escalonador necessita de uma thread auxiliar, a qual é responsável por ativar o método *fire()* da função monotônica no momento apropriado. A Listagem 10 mostra um exemplo de código para esta thread, que sempre deixa a função monotônica executar por 1s.

```
01: public class ThreadFire extends RealtimeThread {
02:   public ThreadFire() { super();
03:     // Declara o SchedulingParameter com a prioridade
04:     SchedulingParameters sched  = new
                          PriorityParameters(PriorityScheduler.MAX_PRIORITY - 10);
05:     setSchedulingParameters(sched); // Atribui os parametros da thread
06:   }
07:   public void run() {
08:     try {
09:       sleep(1000);
10:     } catch(InterruptedException e) {}
11:     Escalonador.fimMonotonica();
12:   }};
```

**Listagem 10. Classe ThreadFire.**

## 5. Conclusões

Este artigo abordou a expressividade da *Real-Time Specification for Java* para a programação de aplicações que utilizam técnicas de Computação Imprecisa. Foi investigado como programar as três formas tradicionais da Computação Imprecisa: função monotônica, função melhoramento e múltiplas versões. Apesar da RTSJ não ser ideal para Computação Imprecisa, isso se torna possível através de disciplina de programação. Com isso, as três formas de tarefa imprecisa foram implementadas com relativa facilidade usando a RTSJ. Todas elas utilizaram o mecanismo tratador de perda de deadline fornecido pelo RTSJ. Por outro lado, a função monotônica usou um recurso novo introduzido na RTSJ, o *AsynchronouslyInterruptedException* e seu método *fire()*.

O programa exemplo usou uma classe Escalonador que recebe informações sobre o comportamento das tarefas e decide sobre o nível de precisão de cada ativação. Como estudo futuro, será necessário avaliar a compatibilidade dos algoritmos adaptativos descritos na literatura com o escalonador baseado em prioridades fixas subjacente à RTSJ.

## Referências


[1] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, J.-Y. Chung. *Imprecise Computations.* Proceedings of the IEEE, Vol. 82, No. 1, pp. 83-94, January 1994.

[2] P. C. Dibble. *Real-Time Java Platform Programming*, Prentice-Hall, 2002.

[3] G.Bollella, B.Brosgol, P.Dibble, *et al. The Real-Time Specification for Java.* Addison-Wesley, 2000.

[4] C. Montez, J. da S. Fraga, R. S. de Oliveira. *Escalonamento Adaptativo (p+i,k)-firm,* III Workshop de Tempo Real - WTR'2001, Florianópolis - SC, 22 de maio de 2001.

[5] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, J.-Y. Chung, W. Zhao. *Algorithms for Scheduling Imprecise Computations.* IEEE Computer, May 1991.