文章编号：2095-2163(2022)06-0065-07 中图分类号：TP302.7 文献标志码：A

基于架构的众核软件可靠性建模与分析

覃志东[1]，郝四明[1]，韦俊银[1]，肖芳雄[2]
（1 东华大学 计算机科学与技术学院，上海 201620；2 金陵科技学院 软件工程学院，南京 211169）

摘 要：众核软件被映射到众核处理器核心上并发执行，其架构不同于单核处理器软件。现有的基于单核处理器软件所建立的可靠性模型不适用于众核软件。针对这种现状，在分析众核软件映射基本流程和众核软件架构的基础上，本文建立了一个众核软件可靠性模型。该模型揭示了任务模块执行时间的不均衡性对系统可靠性的定量影响。该模型的建立，为众核软件建立一套可靠性设计与评估方法打下了基础，对于设计高可靠的众核系统具有一定的意义。

关键词：可靠性；众核软件；软件架构；可靠性测试

Many-core software reliability modeling and analysis based on the architecture

QIN Zhidong[1], HAO Siming[1], WEI Junyin[1], XIAO Fangxiong[2]
(1 College of Computer Science and Technology, Donghua University, Shanghai 201620, China;
2 School of Software Engineering, Jinling Institute of Technology, Nanjing 211169, China)

【Abstract】The architecture of many-core software is different from that of software executing on single-core processors because it is mapped on the cores and executing concurrently. The existing reliability models based on single-core processor software are not suitable for many-core software. In view of this situation, a reliability model for many-core software is formulated in this paper based on the analysis of basic mapping processes as well as the architecture of many-core software. The model reveals the quantitative impact of the imbalance of task modules execution time on system reliability. The reliability model is a basic part of establishing a series of reliability design and evaluation methods for many-core software. And it is of certain significance for the design of many-core systems with high reliability.

【Key words】reliability; many-core software; software architecture; reliability testing

0 引 言

随着芯片制程技术的不断进步，处理器设计技术已进入到众核(Many-Core)时代。如 MIT 早在 2008 年就设计出 64 核心的 Tile64 处理器[1]。

一方面，当晶体管的特征尺寸进入到纳米尺度并不断缩小时，生产缺陷、工艺偏差、温升和老化等因素将导致处理器的成品率和预期可靠性不断降低[2]。所以，为构建高可靠的众核系统，需要高可靠的众核软件系统进行支撑。

但另一方面，为充分发挥众核处理器的并行计算能力，众核软件系统一般都是由很多具有通信依赖关系的任务模块构成，按照特定的优化算法映射到处理器核心上并发执行[3]。由于任务数量众多，并发性将导致各任务及系统的执行状态更加不可预计，使得众核软件系统的可靠性预期也变差。

软件由于逻辑抽象性，其可靠性设计、测试与评估技术是滞后于硬件的。虽然软件可靠性工程已经发展了几十年了，但软件可靠性仍然是导致系统可靠性提升的瓶颈。传统单核处理器执行的软件所面临的可靠性难题，众核软件依然存在。而且，由于众核软件架构和并发执行方式的不同，原来基于单核处理器软件所建立的可靠性模型、可靠性测试方法和评估技术用于众核软件已不合适。

基于这种现状，本文在分析众核软件模型和剖析众核软件映射流程的基础上，对众核软件进行了可靠性建模。模型揭示了各任务模块和软件系统之间的可靠性关系，为后续的可靠性设计、测试与评估方法的研究打下了基础。

1 基于软件架构的典型可靠性模型介绍

为了说明不同的软件架构导致不同的软件系统

基金项目：国家自然科学基金(6126200)；2021 国家级大学生创新创业项目(112-03-0178010/001)。

作者简介：覃志东(1974-)，男，博士，副教授，主要研究方向：嵌入式系统与人工智能；郝四明(1995-)，男，硕士研究生，主要研究方向：嵌入式系统；韦俊银(1972-)，男，博士，讲师，主要研究方向：嵌入式人工智能；肖芳雄(1972-)，男，博士，教授，主要研究方向：大数据与人工智能。

通讯作者：覃志东 Email：zdqing@dhu.edu.cn

收稿日期：2021-12-11

与任务模块的可靠性关系,下面介绍 2 种典型的基于软件架构的可靠性模型。

1.1 串并行软件系统的可靠性模型

串并行模块软件系统是一种用硬件结构来模拟软件架构的模型[4]。其基本结构是由 n 个并行模块组(每个模块组 i 包括 k_i 个模块)和 m 个串行模块组成,如图 1 所示。

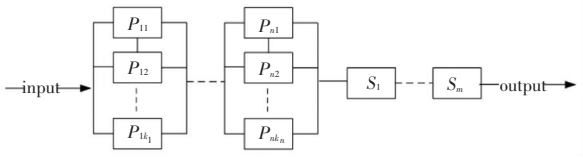


图 1 串并行模块软件系统基本架构

Fig. 1 Basic architecture of a parallel-series modular software system

假设软件系统中模块 i 的失效强度为 $\lambda_i(T_i)$,表示为:

$$\lambda_i(T_i) = a_i\, b_i\, e^{-b_i T_i} \tag{1}$$

其中,a_i、b_i 是常量,分别表示模块 i 中错误的平均值和检测到的错误率。

模块 i 持续工作 x 时长的可靠性为:

$$R(x \mid T_i) = e^{-\lambda_i(T_i)\, x} \tag{2}$$

联合式(1)、(2)可以得图 1 所示软件系统的可靠性:

$$R(x \mid T) = \prod_{l=1}^{n}\left(1 - \prod_{i=1}^{k_l}\left[1 - R_{l_i}(x \mid T_{l_i})\right]\right) \prod_{j=1}^{m} R_j(x \mid T_j) \tag{3}$$

其中,$1 - \prod_{i=1}^{k_l}\left[1 - R_{l_i}(x \mid T_{l_i})\right]$ 为 l 个并行模块组的可靠性;$\prod_{l=1}^{n}\left(1 - \prod_{i=1}^{k_l}\left[1 - R_{l_i}(x \mid T_{l_i})\right]\right)$ 为 n 个并行模块组的总可靠性;$\prod_{j=1}^{m} R_j(x \mid T_j)$ 为 m 个串行模块的总可靠性。

1.2 马尔可夫链执行状态软件可靠性模型

这里,设 $X(t)$ 为一随机过程,若 $X(t_{k+1})$ 的状态只取决于 $X(t_k)$ 的状态,而与之前的 $X(t_{k-1})$、$X(t_{k-2})$、…、$X(t_0)$ 的状态都无关,则称 $X(t)$ 为马尔可夫链。若 t 为离散时间变量,则为离散时间马尔可夫链(Discrete Time Markov Chain, DTMC)。一些软件在执行时,对 CPU 的控制权在不同的软件模块之间转移的过程便是 DTMC[5-6],如图 2 所示。

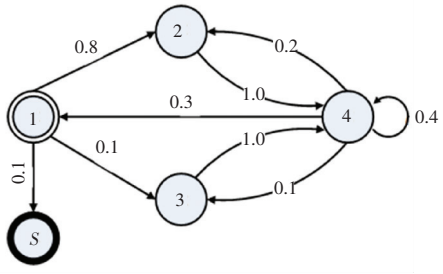


图 2 DTMC 执行状态软件基本架构

Fig. 2 Basic architecture of the DTMC executing state transition software system

用 p_{ij} 表示模块 i 到模块 j 的单步控制传递概率,$p_{ij} \in [0,1]$。因此对于 n 模块软件系统,可得到软件的控制传递矩阵:

$$\boldsymbol{P} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{bmatrix} \tag{4}$$

对于带吸收状态 DTMC,吸收状态 S 和 F 分别表示系统执行成功和失败。在软件系统的实际运行中,当软件模块存在的缺陷被触发后就会导致系统运行的失败,使得控制由模块 i 以概率 q_{iF} 转向吸收状态 F。相反地,若软件系统执行成功,则控制由模块 i 以概率 p_{iS} 转向吸收状态 S。且控制传递概率满足:

$$p_{iS} + \sum_{j=1}^{n} p_{ij} = 1 \tag{5}$$

进一步可得,在运行中的 n 模块软件系统的运行控制传递矩阵为:

$$\boldsymbol{Q} = \begin{bmatrix} \hat{\boldsymbol{P}} & \boldsymbol{C} \\ 0 & \boldsymbol{I} \end{bmatrix}_{(n+2)\times(n+2)} \tag{6}$$

其中,$\hat{\boldsymbol{P}}$ 表示实际运行中 n 个模块间的单步控制传递概率 $n \times n$ 维矩阵 $[q_{ij}]_{n\times n}$,$\boldsymbol{C}$ 表示与吸收状态 S、F 有关的控制传递概率 $n \times 2$ 维矩阵。

因此,可得到该系统的可靠性 R_S 为[6]:

$$R_S = \sum_{i=1}^{n} (\boldsymbol{I}_n - \hat{\boldsymbol{P}})^{-1}_{1i}\, r_i\, p_{iS} \tag{7}$$

2 众核软件映射流程剖析

众核软件往往基于吞吐率、通信量、功耗等优化目标和限制条件,映射到各处理核心上并发执行。任务映射就是离线任务分配、调度和绑定。处于并发执行状态的众核软件的架构是不同于单处理器上

经操作系统动态调度的序贯执行架构。现有的基于架构的可靠性模型不适用于众核软件。

本节首先介绍众核软件映射流程和平衡调度算法，为可靠性建模提供知识基础。

2.1　众核软件映射流程剖析

众核软件映射的基本流程如下[7]。

(1)逻辑处理器抽象。图 3(a)是一个具有 4 核心的物理处理器。根据各核心的个数、互联方式、缓存的设置方式等，可以把物理处理器抽象为逻辑处理器，如图 3(g)所示。其中，物理核心 Core1 抽象为逻辑核心 P_1，物理核心 Core3 被抽象为逻辑核心 P_2，而物理核心 Core2 被抽象为逻辑核心 P_3，物理核心 Core4 被抽象为逻辑核心 P_4。处理器核心很多、甚至有些坏掉了，可以根据具体情况选择一些核心使用，并将其抽象成逻辑处理器。

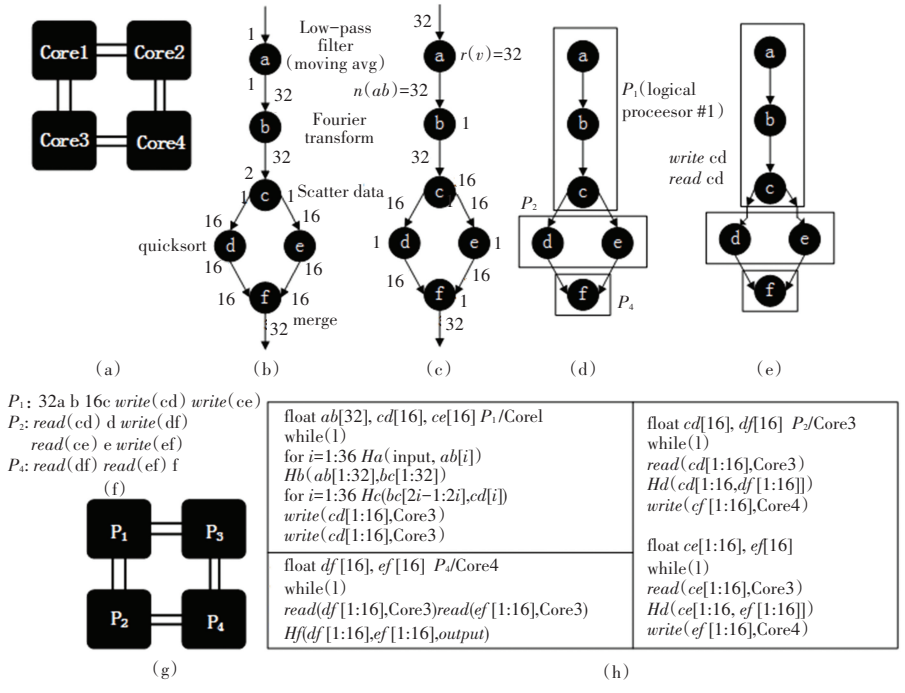


图 3　众核软件映射流程

Fig. 3　Mapping process of the many-core software system

(2)数据流图分析与转换。同步数据流图(Synchronous Data Flow Graph，SDFG)常用来表示一个众核软件，如图 3(b)所示。该软件由 6 个任务 a、b、c、d、e 和 f 组成，分别实现低通滤波、傅里叶变换、数据分发、数据快插和数据融合。其中，a 运行一次要消耗 1 单元的数据，并产生 1 单元数据。b 运行一次要消耗 32 单元的数据，并产生 32 单元数据。所以，为满足软件系统的正常运行，a 需要连续运行 32 次，b 才运行 1 次，而 c 需 16 次，d、e 和 f 各需 1 次；此时，通信量为 32。这种调度平衡后的负载与通信常用有向无循环图(Directed Acyclic Graph，DAG)表示，见图 3(c)。已有成熟算法解决 SDFG 的平衡调度[8]，平衡调度好的 SDFG 可以直接用 DAG 表示，供划分映射。

(3)软件向逻辑处理器分配与调度。此阶段由映射算法把图 3(c)中的 DAG 向逻辑处理器进行分配与调度。由图 3(d)可知，假定：a、b、c 分配到 P_1，d、e 分配到 P_2，f 分配到 P_4，便构成了一个 3 段流水线。系统可以基于最大化该条流水线吞吐率的优化目标进行分配。单个处理核心上的任务集分配结束后，可以基于某种优化方式离线安排好相应的执行顺序，即静态调度[9]。

(4)软件任务向处理器核心绑定。由图 3 中(e)至(h)可知，分配好的任务集需要绑定到逻辑核心所对应的物理核心。如 d、e 对应的是逻辑核心 P_2，要绑定到物理核心 Core3。此时，需要根据任务间的相对位置，建立通信。若 2 个任务在一个核心上，那么定义数组变量，完成数据通信，如 $ab[32]$ 便是 a 对 b 的通信。若两者不在一个核心上，那么要定义数据发送/接收函数；如 $write(cd[1:16]$, Core3) 便是 c 向 d 所在的 Core3 的缓存写数据，而 $read(cd[1:16]$, Core3) 便是 d 把 c 写在本地缓存的

数据读出来。显然 *write* 函数是跨越处理器核心的，其通信时间是受具体通信链路决定的。为了让任务利用以及产生通信数据，还要对任务加上封装函数，如 $Hb(ab[1:32],bc[1:32])$。

(5)众核软件的流水执行模式。假定映射到 P_1 的 a、b、c 及其封装函数为 G_1，映射到 P_2 的 d、e 及其封装函数为 G_2，映射到 P_4 的 f 及其封装函数为 G_3。那么 G_1、G_2、G_3 所绑定的物理核心 Core1、Core3 和 Core4 就构成处理器核心执行流水线，处理器上的 G_1、G_2、G_3 是并行执行的。其中，流水周期 T 由 G_1、G_2、G_3 三者的最大执行时间决定，即：$T = \max\{G_1, G_2, G_3\}$；当流水线充满后，每隔一个周期 T 便有一个系统输出。该流水线的时空图如图 4 所示。

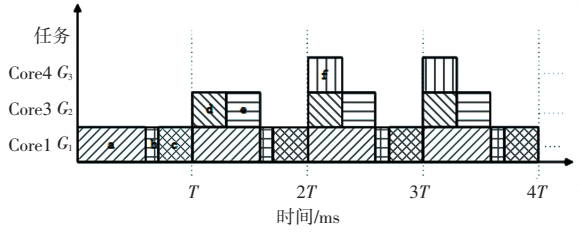


图 4 众核软件流水线执行的时空图

Fig. 4 Executed spatio－temporal diagram of the pipeline for many-core software

2.2 SDFG 的周期平衡调度

SDFG 描述的众核软件如图 5 所示。图 5 中，节点代表任务，任务按字母顺序编号；箭头代表弧(任务之间的输出输入关系)，弧的箭尾上的数字，代表前一个任务执行一次产生的数据资源，箭尖的数字代表下一个任务执行一次所需要消耗的数据资源。弧上方框里的数字代表了弧的编号。

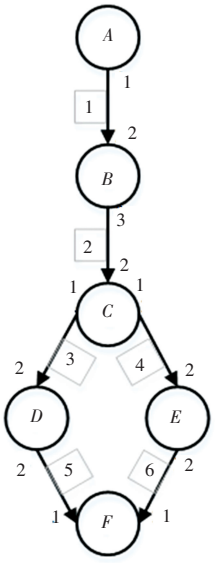


图 5 多链结构 SDFG

Fig. 5 Structure of the multi-chains SDFG

显然，当满足产生与消耗的数据量平衡时，这个 SDFG 表示的众核软件才能正常执行下去。所以，当众核软件执行时，每个任务需要按照一定的次数重复执行；而单个任务重复执行的次数越多，代表分配到处理核心的执行负载越大。则在任务向处理核心分配前，需要对 SDFG 进行平衡调度，从而得到满足通信量平衡时每个任务模块具体执行次数，并间接得到负载量。

SDFG 的周期平衡调度已有多种成熟算法。本文介绍文献[8]中方法。首先，SDFG 用与有向图相关联的拓扑矩阵来表示：行 i 对应弧；j 对应任务节点。第 (i, j) 项表示第 j 个任务节点调用一次，其在弧 i 上产生的数据流。若是消耗，数值为负；若没有在弧上，则为 0。图 5 的拓扑矩阵如式(8)所示：

$$\boldsymbol{\Gamma} = \begin{bmatrix} 1 & -2 & 0 & 0 & 0 & 0 \\ 0 & 3 & -2 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 0 & 0 \\ 0 & 0 & 1 & 0 & -2 & 0 \\ 0 & 0 & 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 0 & 2 & -1 \end{bmatrix} \tag{8}$$

若拓扑矩阵 $\boldsymbol{\Gamma}$ 的秩 $r = s - 1$(s 为节点的数量)，则表示该 SDFG 上能构造周期性可允许的顺序调度。对图 5 的拓扑矩阵矩进行初等行变换，得：

$$\boldsymbol{\Gamma} \rightarrow \begin{bmatrix} 1 & -2 & 0 & 0 & 0 & 0 \\ 0 & 3 & -2 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 0 & 0 \\ 0 & 0 & 0 & 2 & -2 & 0 \\ 0 & 0 & 0 & 0 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{9}$$

矩阵 $\boldsymbol{\Gamma}$ 的秩 $r = 5$。等于 $s - 1$，则可以构造周期性均衡循环调度。具体构造方法是求解满足 $\boldsymbol{\Gamma q} = 0$ 的向量 $\boldsymbol{q}$。例如，拓扑矩阵(9)对应的向量为：

$$\boldsymbol{q} = [8\ 4\ 6\ 3\ 3\ 6]^{\mathrm{T}} \tag{10}$$

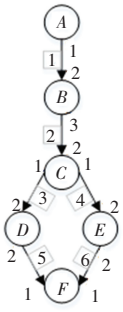
向量 $\boldsymbol{q}$ 中对应的元素为相应任务调度在一个循环周期中被激发执行的次数。

3 众核软件可靠性建模

3.1 SDFG 向 DAG 转换

显然，只有对 SDFG 进行均衡调度后，才能知道一个具体的任务在一个循环周期中对处理核心产生的负载量是多少。可以把调度好的软件转换为由 DAG 来刻画。例如图 5 给出的 SDFG 向 DAG 转化

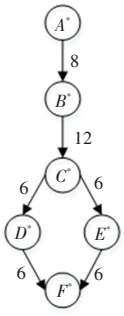
结果如图 6 所示。其中，任务 A 在一个周期循环中，需要激发执行 8 次，每次占用处理核心 5 单位时间，总共占用 40 单位时间；那么用任务 A^* 来代表整体执行 8 次的 A，于是就形成了右边的 DAG 图。



(a)

节点	执行次数	执行时间
A	8	5
B	4	5
C	6	4
D	3	6
E	3	8
F	6	5

(b)



(c)

节点	执行次数	执行时间
A^*	1	40
B^*	1	20
C^*	1	24
D^*	1	18
E^*	1	24
F^*	1	30

(d)

图 6　SDFG 转化为 DAG

Fig. 6　Conversion of SDFG to DAG

3.2　众核软件可靠性建模

假定众核处理器有 n 个处理核心，可能映射到每个处理核心上的最大任务模块数为 m。那么第 i 个处理核心上所映射的任务 A^*_{ij} 调度执行一次的时间(任务负载)可以编号为：T^*_{ij}。其中，$i=1,2,\cdots,n$；$j=1,2,\cdots,m$。例如，核心 i 上映射的任务数为 q，则 $T^*_{iq+1}\cdots T^*_{im}$ 都为 0。则核心 i 上任务总运行时间为：$\sum_{j=1}^{m} T^*_{ij}$。显然，流水线周期 T 取决于 n 个核心中任务集周期性执行一次的最长的时间。即：

$$T=\max\sum_{j=1}^{m} T^*_{ij} \qquad i=1,2,\cdots,n \tag{11}$$

当流水充满后，每过 T 时间便会有个输出。定义核心 i 上任务集运行一次的时间与周期 T 的比：

$$w_i=\frac{\sum_{j=1}^{m} T^*_{ij}}{T} \qquad i=1,2,\cdots,n \tag{12}$$

由于核心 i 上任务集运行时间是不大于 T 的，即：$\sum_{j=1}^{m} T^*_{ij}\leqslant T$，所以 $w_i\leqslant 1$。这些任务集是独立并行执行，处理器核心很多，一般来说满足 $\sum_{i=1}^{n} w_i>>1$，而不是 $\sum_{i=1}^{n} w_i=1$。

核心 i 上任务是分时独占处理核心资源的，第 j 个任务占用比例为：

$$k_{ij}=\frac{T^*_{ij}}{\sum_{j=1}^{m} T^*_{ij}} \qquad j=1,2,\cdots,m \tag{13}$$

且对 $\forall i$，满足 $\sum_{j=1}^{m} k_{ij}=1$。

当处理器稳定工作一段时间 t 以后，处理核心 i 上任务集运行的总时间为：

$$t_i=w_i t \tag{14}$$

此时，处理核心 i 上单个任务 A^*_{ij} 的实际运行时间为：

$$t_{ij}=k_{ij}\,t_i=k_{ij}\,w_i t \tag{15}$$

整个软件系统在 t 时间段内不失效，则必须满足所有核心上工作的任务集在其执行期间不失效。则系统可靠性可以表示为：

$$R_S(t)=P\{t'\geqslant t\}=P(\bigcap_{i=1}^{n} t'_i\geqslant t_i)=\prod_{i=1}^{n} R_i(t_i) \tag{16}$$

设 λ^*_{ij} 为任务 A^*_{ij} 的失效率，则核心 i 上任务集 $\sum_{j=1}^{m} A^*_{ij}$ 的可靠性 $R_i(t_i)$ 可以表示为：

$$R_i(t_i)=P(t'_i\geqslant t_i)=\prod_{j=1}^{m} P(t'_{ij}\geqslant t_{ij})=e^{\sum_{j=1}^{m}-\lambda^*_{ij}t_{ij}} \tag{17}$$

将式(15)代入式(17)，可得：

$$R_i(t_i)=e^{\sum_{j=1}^{m}-\lambda^*_{ij}k_{ij}w_i t} \tag{18}$$

将式(18)代入式(16)，可得：

$$R_S(t)=\prod_{i=1}^{n} e^{\sum_{j=1}^{m}-\lambda^*_{ij}k_{ij}w_i t}=e^{(\sum_{i=1}^{n}\sum_{j=1}^{m}-\lambda^*_{ij}k_{ij}w_i)\,t} \tag{19}$$

又因为，$R_S(t)=e^{-\lambda_S t}$，则有：

$$\lambda_S=\sum_{i=1}^{n}\sum_{j=1}^{m} w_i\,k_{ij}\,\lambda^*_{ij} \tag{20}$$

式(20)便是众核软件系统的失效率 λ_S 与转换后的 A^*_{ij} 的失效率 λ^*_{ij} 的关系。

而 A^*_{ij} 是由 A_{ij} 经平衡调度转换而来的。假设 λ_{ij} 是 A_{ij} 的失效率，A^*_{ij} 的执行就是 A_{ij} 在执行，显然，让二者执行相同的时间，其可靠性是一样的。则有：

$$R_{A^*_{ij}}(t) = R_{A_{ij}}(t) \tag{21}$$

又因为，$R_{A^*_{ij}}(t) = e^{-\lambda^*_{ij} t}$，$R_{A_{ij}}(t) = e^{-\lambda_{ij} t}$，所以：

$$\lambda^*_{ij} = \lambda_{ij} \tag{22}$$

将式(14)代入式(12)得到：

$$\lambda_S = \sum_{i=1}^{n} \sum_{j=1}^{m} w_i \, k_{ij} \, \lambda_{ij} \tag{23}$$

式(23)便是 SDFG 所示软件系统的可靠性模型。

设任务 A_{ij} 在一个周期内被调度执行的次数为 d_{ij}（可由 q 得到），运行一次的时间为 T_{ij}，则有：

$$T^*_{ij} = d_{ij} \, T_{ij} \tag{24}$$

将式(24)代入式(13)，可得 k_{ij}：

$$k_{ij} = \frac{d_{ij} \, T_{ij}}{\sum_{j=1}^{m} d_{ij} \, T_{ij}} \qquad j = 1, 2, \cdots, m \tag{25}$$

4 实验及结果分析

图 7 即为一 SDFG 表示的众核软件系统，及其被映射到的逻辑处理器核心的情况。表 1 是其各个任务模块的参数。

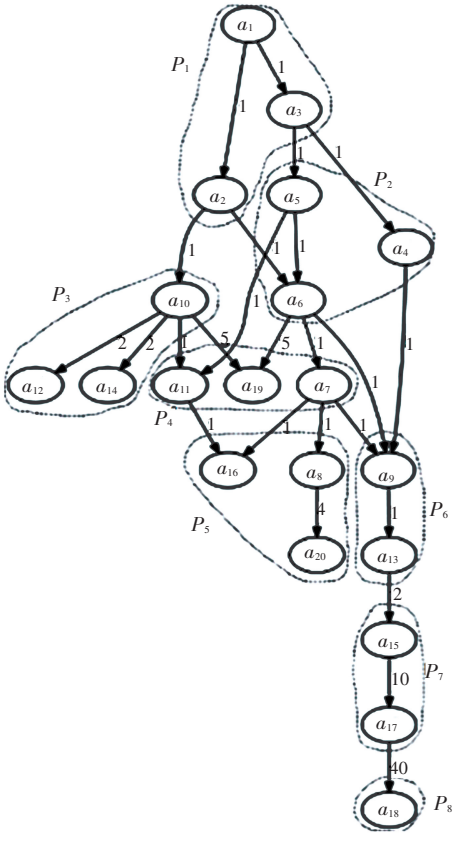


图 7 一个众核软件系统例子

Fig. 7 An example of the many-core software system

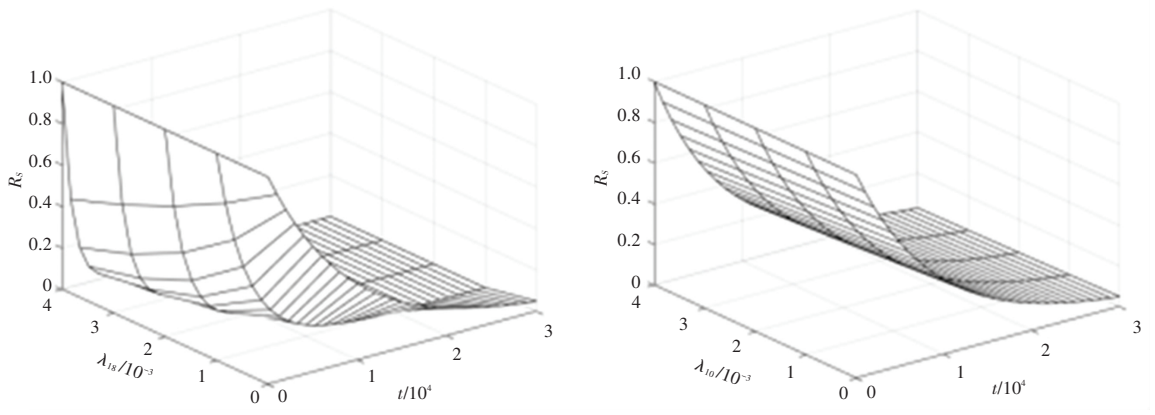
表 1 众核软件的任务模块参数

Tab. 1 Parameters of task modules in the many-core software system

任务模块 (A_i)	执行次数 (q_i)	执行时间 (T_i)	总执行时间 (T_i^*)	所属核心
A_1	1	20	20	P_1
A_2	1	15	15	P_1
A_3	1	8	8	P_1
A_4	1	10	10	P_2
A_5	1	12	12	P_2
A_6	1	16	16	P_2
A_7	1	9	9	P_4
A_8	1	18	18	P_5
A_9	1	11	11	P_6
A_{10}	1	2	2	P_3
A_{11}	1	10	10	P_4
A_{12}	2	12	24	P_3
A_{13}	1	15	15	P_6
A_{14}	1	8	8	P_3
A_{15}	2	7	14	P_7
A_{16}	1	13	13	P_5
A_{17}	5	9	45	P_7
A_{18}	8	10	80	P_8
A_{19}	5	10	50	P_4
A_{20}	4	10	40	P_5

由表 1 可知，模块 A_{18} 的执行时间比例是最高的，而模块 A_{10} 的执行时间比例是最低的。将其它模块的总失效率之和设为 0.000 1 个/h，利用 $R_S(t) = e^{-\lambda_S t}$ 和式(23)，分别对模块 A_{18} 和 A_{10} 的失效率变化所导致对系统可靠性的影响进行了数值模拟，结果如图 8 所示。

由图 8 可知，随着任务模块 A_{18} 失效率的增加，系统可靠性 R_S 降低得更快，表明 R_S 对 A_{18} 的失效率变化非常敏感。但是，随着任务模块 A_{10} 的失效率的增加，系统可靠性 R_S 基本没有变化，说明 R_S 对 A_{10} 的失效率变化不敏感。由于 A_{18} 执行时间比例是远远大于 A_{10} 的，也就表明了执行时间比例高的任务模块对系统可靠性影响大。所以，可以通过本文所建立的众核软件可靠性模型，识别出对系统可靠性影响大的模块，并采取措施切实提高这些任务模块的可靠性，如此就可以有效确保整个软件系统的可靠性。



（a）系统可靠性对任务模块 A_{18} 的敏感性　　（b）系统可靠性对任务模块 A_{10} 的敏感性

图 8　系统对任务模块可靠性变化的敏感性仿真

Fig. 8　Sensitivity simulation of the system to reliability change of a task module

5　结束语

本文在剖析众核软件映射过程的基础上，建立了一个众核软件可靠性模型；该模型建立了软件系统与任务模块之间的可靠性关系。利用该模型，识别出对系统可靠性影响较大的任务模块，可以提高可靠性保障措施的实施针对性。

参考文献

[1] BELL S, EDWARDS B, AMANN J, et al. Tile64- processor: A 64-core SoC with mesh interconnect [C]// IEEE International Solid-State Circuits Conference. San Francisco, CA, USA: IEEE, 2008: 88-98.

[2] 李晓维，胡瑜，张磊，等. 数字集成电路容错设计[M]. 北京：科学出版社，2011.

[3] FOROOZANNEJAD M H. Advances in streaming software synthesis for manycore processor platforms [D]. Berkeley: University of California, 2015.

[4] WANG Zai, TANG Ke, YAO Xin. Multi-objective approaches to optimal testing resource allocation in modular software systems[J]. IEEE Transactions on Reliability, 2010, 59(3): 563-575.

[5] YANG Bo, HU Yanmei, HUANG C Y. An architecture-based multi-objective optimization approach to testing resource allocation [J]. IEEE Transactions on Reliability, 2015, 64(1): 497-515.

[6] 齐蓓. 基于软件体系结构的软件可靠性验证测评方法研究[D]. 上海：东华大学，2013.

[7] HASHEMI M. Automated software synthesis for streaming applications on embedded manycore processors[D]. Berkeley : University of California, Davis, 2011.

[8] BATTACHARYYA S S, LEE E A, MURTHY P K. Software synthesis from dataflow graphs [M]. The Kingdom of the Netherlands: Kluwer Academic Publishers, 1996.

[9] GROOTE D, SMIT R, GERARD J, et al. Incremental analysis of cyclo-static synchronous dataflow graphs[J]. ACM Transactions on Embedded Computing Systems, 2015, 14(4): 1-21.

（上接第 64 页）

[3] 沈冬东，汪海涛，姜瑛，等. 一种融合知识图谱与长短期偏好的下一项推荐算法[J]. 小型微型计算机系统，2020，41(04)：849-854.

[4] 王静，周莹莹. 基于知识图谱的深度学习研究现状及趋势计量可视化分析[J]. 信息与电脑(理论版)，2020，32(01)：131-132.

[5] 陶睿，吴继春，谢胜强，等. 深度学习和知识图谱在智能监管中的应用研究[J]. 金融纵横，2019(08)：56-66.

[6] 张颖怡，章成志. 基于学术论文全文的研究方法句自动抽取研究[J]. 情报学报，2020，39(06)：640-650.

[7] 蒲姗姗，何燕. 个性化学术资源推荐研究：现状、特点及展望[J]. 图书馆学研究，2019(16)：9-17.

[8] 莫怡丹. 小数据思维驱动下图书馆学术资源的个性化推荐服务探讨[J]. 中国中医药图书情报杂志，2020，44(02)：22-25.

[9] 田野. 关联数据驱动的学术资源语义检索推荐系统框架[J]. 图书馆理论与实践，2019(02)：49-54.

[10] 吕敏. 基于小数据的图书馆学术资源个性化推荐[J]. 中文科技期刊数据库(全文版)图书情报，2020(03)：200-201.

[11] 赵庆来. 学术期刊精准传播平台构建与内容推荐[J]. 中国出版，2020(05)：23-27.

[12] 陈长华，李小涛，邹小筑，等. 融合 Word2vec 与时间因素的馆藏学术论文推荐算法[J]. 图书馆论坛，2019，39(05)：110-117.