鲲鹏软件迁移一本通

文档版本 01

发布日期 2021-08-23

华为技术有限公司

版权所有 © 华为技术有限公司 2021。 保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明

和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

Kunpeng Computing

目 录

1 Porting Advisor 工具概述....................1

2 源码迁移....................3

2.1 make/cmake 构建文件....................4

2.1.1 C/C++识别 x86 运行模式编译选项....................4

2.1.2 构建文件中添加-march 编译选项....................5

2.1.3 C/C++强制添加-fsigned-char 编译选项....................5

2.1.4 Fortran 源码中解除代码行部分格式限制....................6

2.1.5 Fortran 构建文件中根据是否存在 intrinsic 函数判断是否添加-fdec-math 编译选项....................6

2.1.6 依赖库....................7

2.2 C/C++....................7

2.2.1 Intrinsics 函数....................8

2.2.2 built-in 函数....................9

2.2.3 struct 结构体....................9

2.2.4 平台相关宏处理....................10

2.2.5 无嵌套的宏代码块....................11

2.2.6 两层嵌套的宏代码块....................11

2.2.7 多层嵌套的宏代码块....................12

2.3 Fortran....................13

2.3.1 Fortran 语法解析特性....................13

2.3.1.1 Fortran 字符串解析....................13

2.3.1.2 字符串与数值变量间的转化问题....................13

2.3.1.3 FORMAT 的变量表达式....................14

2.3.1.4 逻辑变量操作....................14

2.3.1.5 open binary 操作....................15

2.3.1.6 字符数组的初始化....................15

2.3.1.7 IBITS 兼容性....................15

2.3.1.8 数组长度问题....................16

2.3.1.9 data 关键字复制....................16

2.3.1.10 C 调用 Fortran 内置函数 iargc_....................17

2.3.1.11 宏定义大小写....................17

2.3.1.12 open recl 操作....................18

2.3.1.13 Fortran 行宽限制....................18

2.3.1.14 过滤注释内容....................19

Kunpeng Computing

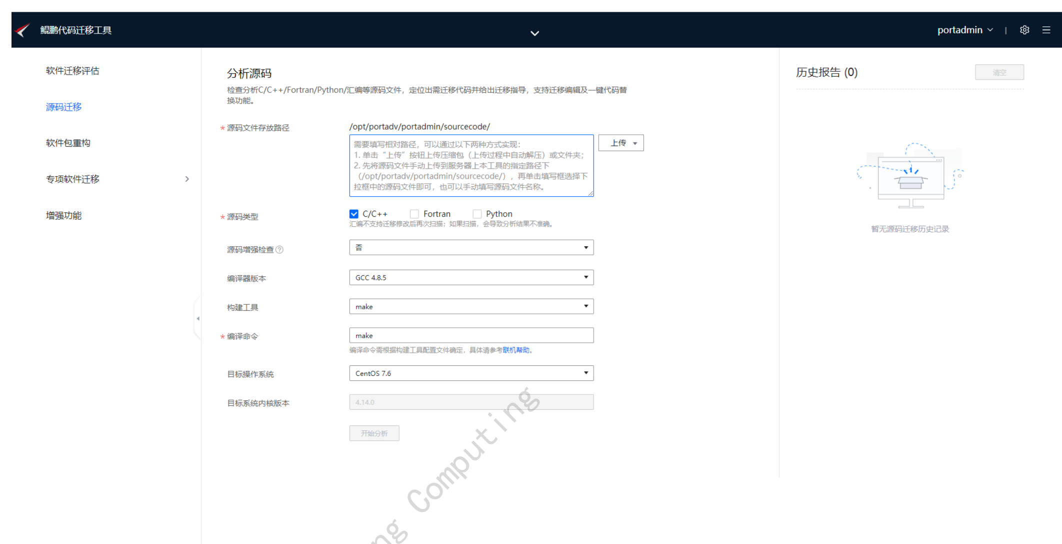
2.3.2 Fortran 特性函数 19
2.4 汇编 20
2.4.1 内嵌汇编 20
2.4.1.1 内嵌汇编代码段建议-明确替换建议 20
2.4.1.2 内嵌汇编代码段建议-增加头文件 21
2.4.1.3 内嵌汇编代码段建议-模糊提示 22
2.4.1.4 内嵌汇编代码段建议-无替换建议 23
2.4.2 全汇编 23
2.5 Python 25
2.5.1 识别 Python 代码中加载动态库函数 25
2.5.2 识别 Python 调用的依赖库 25
3 内存一致性检查 26
3.1 内存一致性静态检查 27
3.2 编译器自动修复 28
4 64 位运行模式检查 29
5 结构体字节对齐检查 35
5.1 结构变量内存空间分配不需要优化类型 35
5.2 可进行结构变量内存空间分配优化类型 36
6 软件迁移评估 38
6.1 RPM 包分析 38
6.2 Jar/War 包分析 40
6.3 已安装软件分析 41
6.4 其它软件包迁移评估分析 42
7 软件包重构 43
7.1 RPM 包重构 43
7.2 DEB 包重构 44
8 专项软件迁移 45

Kunpeng Computing

1 Porting Advisor 工具概述

为保证用户的应用程序能够在鲲鹏平台正常使用，用户需要提前检查其源代码、软件安装包是否能够兼容鲲鹏平台。鲲鹏代码迁移工具是一款可以帮助用户高效完成这种迁移工作的工具，它能够简化用户将Linux应用从x86平台向基于鲲鹏916/920的服务器及虚拟机迁移的工作，工具的界面展示如图1-1所示：

图 1-1 迁移工具界面图



Kunpeng Computing

当前工具支持的功能特性如表1-1所示：

表 1-1 工具支持的功能特性

功能	描述
源码迁移	检查用户C/C++/Fortran/汇编等软件构建工程文件，并指导用户如何迁移该文件。 检查用户C/C++/Fortran/Python/汇编等软件源码，并指导用户如何迁移该源码文件。 x86汇编指令转换，分析x86汇编指令，并转换成功能对等的鲲鹏汇编指令。

功能	描述
内存一致性检查	内存一致性检查功能可以分析、修复用户软件中存在的内存读写顺序问题。
64位运行模式检查	64位运行模式检查功能可以检查和分析原32位平台上的软件源代码，将源码迁移至64位平台时需要修改的地方识别出来。
结构体字节对齐检查	结构体字节对齐检查功能可以检查源码中结构体类型变量的字节对齐情况，用户可以根据检查结果判断是否需要进行字节对齐所需要的数据结构定义修改。
软件迁移评估	检查用户软件安装包（RPM、DEB、TAR、ZIP、GZIP文件等）中包含的SO（Shared Object）依赖库和可执行文件，并评估SO依赖库和可执行文件的可迁移性。 检查用户Java类软件包（JAR、WAR）中包含的SO依赖库和二进制文件，并评估SO依赖库和二进制文件的可迁移性。 检查指定的用户软件安装路径下的SO依赖库和可执行文件，并评估SO依赖库和可执行文件的可迁移性。
软件包重构	在鲲鹏平台上，分析用户提供的x86平台rpm格式或deb格式的软件包构成，重构并生成鲲鹏平台兼容的对应格式软件包。
专项软件迁移	在鲲鹏平台上，通过自动化方式，对部分常用的解决方案组件，进行x86源码修改、编译并构建生成鲲鹏平台兼容的软件包。

Kunpeng Computing

2 源码迁移

当用户能够提供源代码进行迁移分析时，可以使用源码迁移功能完成迁移工作。源码迁移功能主要解决了用户的代码兼容性人工排查困难、严重依赖迁移的个人经验、需要反复依赖编译调错定位导致执行效率低等痛点。按照源代码到二进制文件的翻译方式的不同，代码迁移工具将编程语言分为两类：一类是编译型，一类是解释型。

编译型语言

典型的如C/C++编译型语言，需要在运行前先翻译到底层的指令集，所以与架构相关。因此，基于x86架构编译环境编译出的C/C++语言应用程序，无法直接运行在鲲鹏服务器中，需要进行源码的移植编译。

解释型语言

基于解释型语言开发的应用程序，例如Java、Python、Scala等开发的应用程序，这些语言运行在对应语言的解释器上，由解释器完成上层代码到底层指令集的翻译，因此对于纯粹由这类语言代码构成的应用，只要鲲鹏平台的运行环境能够运行这些语言的解释器，就可以直接运行这类应用程序。另一方面，Java应用的JAR/WAR包内及Python应用的Wheel包或Egg包中可能包含基于C/C++语言开发的so（Shared Object）依赖库文件，这类so依赖库文件仍然遵循编译型语言的运行规则，需要对源代码进行移植编译，在完成so依赖库文件的移植后，才能确保重新打包生成的JAR/WAR/Wheel/Egg包可以运行在鲲鹏环境中。

说明

目前工具可以检查分析C/C++/Fortran/Python/汇编等源码文件，检出需迁移的代码并给出迁移指导。

分析文件类型如**表2-1**所示：

表 2-1 源码类型

源码类型	文件类型	动作分析
构建文件	Makefile、CMakelists.txt	对构建配置文件，主要是分析编译选项和链接库，识别出编译选项修改点和链接库依赖，给出迁移建议。

源码类型	文件类型	动作分析
C/C++	.c、.cpp、.h等	对C/C++文件进行分析，针对intrinsics函数、built-in函数、struct结构体、宏定义等给出迁移建议.
Fortran	.f、.fpp等	对Fortran语言源码，工具主要进行built-in函数、代码语法方面的检查，并给出迁移建议。
汇编	.s、.asm	对汇编语言源码，工具会进行解析和转换，提供转换得到的鲲鹏兼容版本的汇编源代码，用户可使用这些建议的汇编源代码进行替换；对嵌入式汇编，工具会解析嵌入式汇编代码块，针对不同场景给出特定的迁移建议，用户可根据提示进行修改。
Python	.py	对Python语言源码，工具主要是识别python源码中调用动态库的代码以及识别动态库文件，给出迁移建议。

2.1 make/cmake构建文件

2.2 C/C++

2.3 Fortran

2.4 汇编

2.5 Python

2.1 make/cmake 构建文件

针对C/C++/Fortran的make/cmake构建文件，可识别的编译选项有如下几类：

2.1.1 C/C++识别 x86 运行模式编译选项

说明

-m64是x86平台应用程序编译选项，编译后产生的代码将运行在64位模式下。该模式下long和指针数据类型是64位。但是鲲鹏平台不支持该选项，需要根据gcc版本的不同，进行对应的修改。

处理方法

根据gcc版本的不同，分别进行处理，如在gcc4.8中，鲲鹏平台编译器没有替换该编译选项的选项，该编译选项需要进行删除操作；而在gcc6.4中工具会给出替换编译选项为“-mabi=lp64”的建议。具体是哪种情况，工具会根据用户启动源码迁移任务时设置的目标环境GCC版本信息进行判断并给出建议。

示例

原始makefile中展示代码为：

```
CFLAGS=-g3 -W -Wall -m64 -Wno-unused-but-set-variable -O0 -DDEBUG=1
```

需要修改成：

```
CFLAGS=-g3 -W -Wall -mabi=lp64 -Wno-unused-but-set-variable -O0 -DDEBUG=1
```

2.1.2 构建文件中添加-march 编译选项

说明

在编译时用户可以通过该编译选项指定处理器架构为ARMv8，使编译器按照鲲鹏处理器的微架构和指令集生成可执行程序，提升性能。

处理方法

在编译选项中添加或者修改-march=armv8-a。如果使用的GCC编译器版本在7.x以上，需要设置为-march=armv8.2-a。具体是哪种情况，工具会根据用户启动源码迁移任务时设置的目标环境GCC版本信息进行判断并给出建议。

示例

原始makefile中展示代码为：

```
CFLAGS=-g3 -W -Wall -Wno-unused-but-set-variable -O0 -DDEBUG=1 -march=i386
```

需要修改成：

```
CFLAGS=-g3 -W -Wall -Wno-unused-but-set-variable -O0 -DDEBUG=1 -march=armv8-a
```

2.1.3 C/C++强制添加-fsigned-char 编译选项

说明

char类型变量在不同CPU架构下默认行为（是否带符号）不一致，在x86架构下默认为signed char（有符号字符型），在鲲鹏平台默认为unsigned char（无符号字符型），x86架构代码移植到鲲鹏平台时，需要强制指定char类型变量默认为signed char。

处理方法

在makefile构建文件中进行修改，首先找到编译选项定义处，然后加入“-fsigned-char”选项，从而指定鲲鹏平台下的char类型变量默认为有符号数。

示例

原始makefile中展示代码为：

```
CFLAGS=-g3 -W -Wall -Wno-unused-but-set-variable -O0 -DDEBUG=1
```

需要修改成：

```
CFLAGS=-g3 -W -Wall -Wno-unused-but-set-variable -O0 -DDEBUG=1 -fsigned-char
```

2.1.4 Fortran 源码中解除代码行部分格式限制

说明

在fortran构建配置文件中，默认情况下，编译器有行宽限制功能，会检查每行代码的行宽是否满足要求，可以通过本选项禁用编译器的行宽限制功能。

处理方法

makefile构建配置文件中，添加 “-ffree-line-length-none”编译选项。

示例

原始makefile中展示代码为：

```
CFLAGS=-g3 -W -Wall -Wno-unused-but-set-variable -O0 -DDEBUG=1
```

需要修改成：

```
CFLAGS=-g3 -W -Wall -Wno-unused-but-set-variable -O0 -DDEBUG=1 -ffree-line-length-none
```

2.1.5 Fortran 构建文件中根据是否存在 intrinsic 函数判断是否添加-fdec-math 编译选项

说明：

Fortran中部分built-in（例如：CONTAN，TAND, ATAND等）在鲲鹏平台上是兼容的，当包含这些函数的程序需要迁移到鲲鹏平台时，需要在对应的makefile/CMakeLists.txt文件中加上编译选项-fdec-math。以下是需要进行适配的built-in函数中的一部分:

- ATAN2D
- ATAND
- COSD
- COTAN
- COTAND
- SIND
- TAND

Kunpeng Computing

处理方法

当Fortran源码包中有Makefile、CMakeLists.txt构建文件，且在fortran文件中存在兼容的intrinsic函数时，工具会建议添加-fdec-math编译选项。

示例

源码中存在Y = COSD(X)则需要在Makefile、CMakeLists.txt构建文件中进行添加-fdec-math。

如原始makefile中展示代码为：

```
FFLAGS = -g -fpp -openmp -standard-realloc-lhs
```

 版权所有 © 华为技术有限公司

需要修改成：

```
FFLAGS = -g -fpp -openmp -standard-realloc-lhs -fdec-math
```

2.1.6 依赖库

说明

对于构建文件构建过程中使用到的依赖库，工具会进行识别、分析、判断，并说明鲲鹏平台是否存在同名依赖库。存在的提供下载源，不存在的会给出明确的提示，不明确的会提示用户自行分析。由于工具内置的依赖文件列表与目标操作系统版本强关联，因此，在启动工具源码分析任务的时候用户需要正确选择源码迁移任务的目标操作系统。

示例

在CMakeLists.txt中有如下源码：

link_libraries(gcc)

add_library(tbb SHARED IMPORTED)

add_library(tbbmalloc SHARED IMPORTED)

link_libraries(tbb tbbmalloc)

link_libraries(${ZLIB_LIBRARIES})

源码经过工具分析后，扫描出四个依赖，具体如**图2-1**所示：

图 2-1 扫描结果

与架构相关的依赖文件

序号	依赖文件名	文件类型	待下载软件包名称	分析结果	处理建议
1	libtbb.so.2	动态库	tbb-4.1-9.20130314....	可兼容替换	下载 | 复制链接
2	libtbbmalloc.so.2	动态库	tbb-4.1-9.20130314....		
3	libz.so.1	动态库	zlib-1.2.7-18.el7.aarc...	可兼容替换	下载 | 复制链接
4	libgcc.so	动态库	--	待验证替换	请先在鲲鹏平台上验证。若不兼容，请联系供应方获取鲲鹏兼容版本，或获取源码并编译成鲲鹏兼容版本

扫出依赖文件分别是：libtbb.so.2、libtbbmalloc.so.2、libz.so.1、libgcc.so，前三个依赖文件存在于代码迁移工具内置的依赖库列表中，工具会将包含了这些依赖文件的待下载的rpm包名称和链接提供给用户，具体如下：**https://archive.kernel.org/centos-vault/altarch/7.6.1810/os/aarch64/Packages/tbb-4.1-9.20130314.el7.aarch64.rpm**、**https://archive.kernel.org/centos-vault/altarch/7.6.1810/os/aarch64/Packages/zlib-1.2.7-18.el7.aarch64.rpm**。这些依赖库是ARM服务器支持的，用户可以直接下载替换使用，对于第四个，用户需要在鲲鹏平台自行进行验证后再使用。

2.2 C/C++

C/C++语言文件（.c,.cpp,.h,.hpp等）中需要迁移的内容一般可分为： Intrinsics函数、built-in函数、struct结构体、宏定义等。

2.2.1 Intrinsics 函数

说明

Intrinsic函数是Intel为充分挖掘x86架构cpu性能而开发的、与x86 cpu架构密切相关的底层函数，这些函数与ARM架构无法兼容。为确保这些函数能够在ARM架构上使用，业界创建了开源的sse2neon项目来解决此问题，华为也提供了avx2neon开源解决方案来解决此问题。avx2neon相比sse2neon，覆盖的函数更多，性能更好，兼容性也更佳。因此，代码迁移工具在识别到intrinsic函数时，会推荐用户使用avx2neon。

处理方法

用户可通过添加头文件（avx2neon.h）来实现支持intrinsics函数在ARM系统上的使用，同时，需要下载avx2neon相关的头文件到所修改的文件所在的目录中。avx2neon的下载链接是“https://github.com/kunpengcompute/AvxToNeon”。

具体操作：增加‘#include “avx2neon.h”’到工程中，下载avx2neon相关的头文件到所修改的文件所在的目录中，并修改或者设置条件编译宏以实现对原有的x86平台头文件进行平台兼容处理。

此类intrinsic函数举例如下：

- _addcarryx_u32
- _addcarryx_u64
- _mm_aesdec_si128
- _mm_aesdeclast_si128
- _mm_aesenc_si128
- _mm_aesenclast_si128
- _mm_aesimc_si128
- _mm_aeskeygenassist_si128

示例

原始代码：

```
#include <emmintrin.h>
......
zero=_mm_set1_epi32(0);
```

根据工具建议，可以修改成：

```
#if defined(__aarch64__)
    #include "avx2neon.h"
    //Suggestion: Visit 'https://github.com/kunpengcompute/AvxToNeon' and obtain the 'avx2neon.h' source
code according to the README.md file.
#endif
#if defined(__X86_64__)
    #include <emmintrin.h>
#endif
......
zero = _mm_set1_epi32(0);
```

Kunpeng Computing

如果用户使用的intrinsic函数实现不在AvxToNeon项目提供的头文件中，用户需要访问github网站，通过README.md来了解如何获取这部分函数的函数体实现文件。在获取这部分代码头文件后，将其放置到所修改文件的目录中，同时将avx2neon.h拷贝到工程目录下。

2.2.2 built-in 函数

说明

x86编译环境下特有的内置函数，在鲲鹏平台不兼容，如果直接使用，会导致编译失败，需要用户确认并修改。

列举gcc5.3的built-in函数如下：

- __builtin_copysignq
- __builtin_cpu_init
- __builtin_cpu_is
- __builtin_cpu_supports
- __builtin_fabsq
- __builtin_huge_valq
- __builtin_ia32_addpd
- __builtin_ia32_addpd256

处理方法

设置条件编译宏以实现对原有的x86平台相关代码的兼容，并在arrch64分支里面增加适配鲲鹏平台的代码。

示例

原始代码：

```
__builtin_copysignq
```

修改成：

```
#if defined(__x86_64__)
    __builtin_copysignq
#elif defined(__aarch64__)
    // Suggestion:
    // Kunpeng does not support the builtin fucntion, please check it.
#endif
```

在aarch64分支里面，需要用户添加对应的适配代码。

Kunpeng Computing

2.2.3 struct 结构体

说明

C/C++源码中使用结构体的场景比较多，通过付出一定的内存空间为代价，可以充分利用鲲鹏920 L3 Cache的带宽优势，从而达到提高程序性能的目的。对于这种场景，工具可根据用户设置进行优化性建议的提醒，提示用户强制在为结构体变量分配地址时进行128字节对齐，通过这种操作，最高可以将程序代码性能提升一倍。

处理方法

工具会建议用户在struct结构定义处增加__attribute__((__aligned__(128)))强制进行128字节对齐。

示例

原始代码：

```
typedef struct {
    uint8_t *bits;
    uint64_t size;
    uint64_t cap;
} Bit2Vec;
```

修改后代码：

```
typedef struct {
    uint8_t *bits;
    uint64_t size;
    uint64_t cap;
} __attribute__((__aligned__(128))) Bit2Vec;
```

2.2.4 平台相关宏处理

说明

从平台兼容性角度看，系统中的宏可分类为x86宏、ARM宏、x86和ARM都支持的宏、以及其它平台相关的宏，其中WIN32相关宏作为x86特殊宏进行处理，如果不属于这几种情况，则归属于未定义宏。

处理方法

工具会根据宏代码块中分支的逻辑关系，判断各分支属于哪些平台，总的代码块逻辑结果中如果没有ARM平台相关宏，工具会提示用户进行修改。具体相关宏示例可查看如下表2-2。

表 2-2 平台宏举例

ARM平台宏	x86平台宏	ARM和x86都支持	Other
__arm__	__amd64__	__STDC_HOSTED__	__alpha__
__thumb__	i386	__INT64_TYPE__	__370__
__ARM_ARCH_4T__	__k8__	_LP64	__mips
__aarch64__	__SSE2__	__WCHAR_MAX__	__MIPS__

Kunpeng Computing

示例

原始代码：

```
#if (__i386 || __amd64 || __powerpc__) && __GNUC__
    #define GNUC_VERSION (__GNUC__ * 10000 + __GNUC_MINOR__ * 100 + __GNUC_PATCHLEVEL__)
    #if defined(__clang__)
        #define HAVE_ATOMIC
    #endif
    #if (defined(__GLIBC__) && defined(__GLIBC_PREREQ))
        #if (GNUC_VERSION >= 40100 && __GLIBC_PREREQ(2, 6))
            #define HAVE_ATOMIC
        #endif
    #endif
#endif
```

修改后是：

```
#if (__i386 || __amd64 || __powerpc__) && __GNUC__
    #define GNUC_VERSION (__GNUC__ * 10000 + __GNUC_MINOR__ * 100 + __GNUC_PATCHLEVEL__)
    #if defined(__clang__)
        #define HAVE_ATOMIC
    #endif
    #if (defined(__GLIBC__) && defined(__GLIBC_PREREQ))
        #if (GNUC_VERSION >= 40100 && __GLIBC_PREREQ(2, 6))
            #define HAVE_ATOMIC
        #endif
    #endif
#elif defined(__aarch64__)
// Suggestion:    Add code that adapts to the Kunpeng platform.
#endif
```

2.2.5 无嵌套的宏代码块

说明

C/C++源码文件中的宏代码块如果未定义aarch64分支，工具会检出这种情况并提示用户增加aarch64代码分支。

处理方法

如果宏块内没有aarch64分支，工具会在宏块末尾提示加aarch64分支，且提示用户根据实际代码场景在新加的aarch64分支内补充上在鲲鹏平台上可用的代码。

示例

C/C++源码文件中有如下代码：

```
#if defined(__x86_64__) || defined(__x86_64__) || defined (__i386__)
    #define UNALIGNED_LE_CPU
#endif
```

需要修改为：

```
#if defined(__x86_64__) || defined(__x86_64__) || defined (__i386__)
    #define UNALIGNED_LE_CPU
#elif defined(__aarch64__)
// Suggestion: dd code that adapts to the Kunpeng platform.
#endif
```

Kunpeng Computing

2.2.6 两层嵌套的宏代码块

说明

C/C++源码文件中的宏代码块如果还嵌套了宏代码块，且整体上没有aarch64分支，工具会检出这种情况并提示用户增加aarch64分支。

处理方法

如果嵌套宏代码块内没有aarch64分支，工具会在宏块最外层末尾提示加aarch64分支，且提示用户根据实际代码场景在新加的aarch64分支内补充上在鲲鹏平台上可用的代码。

示例

C/C++源码文件中：

```
#if defined(__x86__)
    _mm_stream_si64(dest + i, src[i]);
    #ifdef __amd64__
        _mm_stream_si64(dest + i, src[i]);
    #endif
#endif
```

需要修改为：

```
#if defined(__x86__)
    _mm_stream_si64(dest + i, src[i]);
    #ifdef __amd64__
        _mm_stream_si64(dest + i, src[i]);
    #endif
#elif defined(__aarch64__)
// Suggestion:  Add code that adapts to the Kunpeng platform.
#endif
```

2.2.7 多层嵌套的宏代码块

说明

C/C++源码文件中的宏代码块如果还嵌套了多个宏代码块，且整体上没有aarch64分支，工具会检出这种情况并提示用户增加aarch64分支。

处理方法

如果多层嵌套宏块内没有aarch64分支，工具会在宏块末尾提示加aarch64分支，且提示用户根据实际代码场景在新加的aarch64分支内补充上在鲲鹏平台上可用的代码。

示例

C/C++源码文件中：

```
#if defined(__x86__)
    _mm_stream_si64(dest + i, src[i]);
    #ifdef __amd64__
        _mm_stream_si64(dest + i, src[i]);
        #ifdef __alpha
            _mm_stream_si64(dest + i, src[i]);
        #endif
    #endif
#endif
```

Kunpeng Computing

需要修改为：

```
#if defined(__x86__)
    _mm_stream_si64(dest + i, src[i]);
    #ifdef __amd64__
        _mm_stream_si64(dest + i, src[i]);
        #ifdef __alpha
            _mm_stream_si64(dest + i, src[i]);
        #endif
    #endif
#elif defined(__aarch64__)
// Suggestion:  Add code that adapts to the Kunpeng platform.
#endif
```

2.3 Fortran

代码迁移工具支持将基于Intel Fortran编译器编写的Fortran源码迁移修改为基于鲲鹏版本GCC Fortran编译器编写的程序，并在迁移过程中检查编译器支持的特性及语法扩展特性。

工具支持的Fortran语言文件类型有：有.f、.for、.fpp, .ftn、.f90、.f95、.f03、.f08（不区分大小写）。

工具支持的GCC Fortran编译器版本有7.x, 8.x和9.x。

2.3.1 Fortran 语法解析特性

2.3.1.1 Fortran 字符串解析

说明

Fortran函数FORMAT后跟随的括号中，如果出现多个成对的单引号，GCC Fortran编译器会将源码中FORMAT函数代码解析为多个字符串。

处理方法

对于语法格式如：FORMAT('notice , failure number is [',(I),']')，工具会提示将FORMAT括号中的所有字符用双引号框住以确保GCC Fortran解析的结果与用户原始意图一致。

示例

原始代码为：

```
FORMAT('notice , failure  number is [',(I),']')
```

需要修改成：

```
FORMAT("notice , failure   number is [',(I),']")
```

Kunpeng Computing

2.3.1.2 字符串与数值变量间的转化问题

说明

鲲鹏平台不支持'(I)'这种方式，可以将'(I)'改成'*'。

处理方法

在遇到Fortran源代码，如：read(xx1, '(I)') xx2；write(xx2, '(I)') xx1，工具会提示用户将代码中的'(I)'改成*。

示例

原始代码为：

```
read( xx1, '(I)' ) xx2
```

需要修改成：

```
read( xx1, '*' ) xx2
```

2.3.1.3 FORMAT 的变量表达式

说明

在GCC Fortran源码编译的时候，如果FORMAT中有<xx>这种格式，并且<xx>不是字符串，迁移到鲲鹏平台上时工具会识别并提示用户需要适配修改。

示例

原始代码为：

```
WRITE(6,20) INT1
 20 FORMAT(I<N+1>)
```

需要修改成

```
CHARACTER(LEN=20) FMT
WRITE(FMT,'("(I", I0,   ")")') N+1
WRITE(6,FMT) INT1
```

或者

```
CHARACTER(LEN=20) FMT
WRITE(FMT,*) N+1
WRITE(6,"(I" //   ADJUSTL(FMT) // ")") INT1
```

2.3.1.4 逻辑变量操作

说明

逻辑变量 和.TRUE. .FALSE.的比较，迁移到鲲鹏平台上，逻辑变量比较符不能为“==”、“.EQ.”、“!=”、“.NE.”。

处理方法

工具将识别逻辑变量比较符“==”、“.EQ.”并提示用户改为“.EQV.”，将“!=”、“.NE.”改为“.NEQV.”。

Kunpeng Computing

示例

原始代码为：

```
IF(BACKWARD_ADVECTION==.TRUE.)THEN 或 IF(BACKWARD_ADVECTION .EQ. .TRUE.)THEN
```

需修改为：

```
IF(BACKWARD_ADVECTION .EQV. .TRUE.)THEN
```

原始代码为：

```
IF(BACKWARD_ADVECTION!=.TRUE.)THEN 或 IF(BACKWARD_ADVECTION .NE. .TRUE.)THEN
```

需修改为：

```
IF(BACKWARD_ADVECTION .NEQV. .TRUE.)THEN
```

2.3.1.5 open binary 操作

说明

open函数中如果form参数值为'binary'，迁移到鲲鹏平台上时，需要修改为'stream'。

处理方法

open函数中如果form参数为'binary'，则工具会提示用户修改为'stream'。

示例

原始代码为：

```
open(unit=xxx, file=xxx,   form='binary', form='unformatted')
```

需修改为：

```
open(unit=xxx, file=xxx, access='stream', form='unformatted')
```

2.3.1.6 字符数组的初始化

说明

鲲鹏平台上在对character类型的字符数组初始化时，Fortran字符串数组要求每个元素长度都一样，所以需要在数组元素之前加上长度限制。

处理方法

工具会识别代码中对character类型的字符数组的初始化操作并提示用户在数组元素之前加上长度限制。

示例

原始代码为：

```
character(len=80), dimension(5) :: bed_snames = (/"bed_thick","bed_age","bed_por","bed_diff","bed_btcr"/)
```

需修改为：

```
character(len=80), dimension(5) :: bed_snames = (/
character(80)::"bed_thick","bed_age","bed_por","bed_diff","bed_btcr"/)
```

Kunpeng Computing

2.3.1.7 IBITS 兼容性

说明

在Fortran中，如IBITS(-1,0,52)，0+52不能大于-1的bitsize，但是Fortran默认是32位整形，迁移到鲲鹏平台时，需要将整形默认设置成64位。

处理方法

工具会检查Fortran内置函数IBITS，并判断第二、三参数字和不能大于第一个参数的数字，如果大于则将第一个参数强制转换成int8类型，不满足则给出修改提示。

示例

原始fortran中展示代码为：

```
IBITS(-1,0,52)
```

需要修改为：

```
IBITS(int(-1),0,52)
```

2.3.1.8 数组长度问题

说明

当Fortran中函数的形参是数组的时候，该数组是包含维度和长度信息的。如果Fortran函数的形参是一维数组，迁移到鲲鹏平台上时，建议用户检查修改。

处理方法

工具会检查作为函数形参的数组，数组类型有：integer, real, complex, logical，如果长度大于1，工具可支持quick fix修改；如果长度为变量，则提示用户检查修改。

示例

原始代码为：

```
integer function Add(x,y)
implicit none
integer ::x,y
real x(3)
real z(1)
Add=x+y
end function add
```

real x(3)该行会有提示建议：

需要修改为：

```
integer function Add(x,y)
implicit none
integer ::x,y
real x(*)
real z(1)
Add=x+y
end function add
```

Kunpeng Computing

2.3.1.9 data 关键字复制

说明

Intel Fortran允许使用字符串赋初值，通常就是空字符串赋初值0，但GCC Fortran编译器不允许整形或浮点用字符串方式赋值，如DATA JBLANK/' '/，需要把空字符串换成数字0。

处理方法

工具识别到DATA XXX/''/格式时，会提示用户将/''/修改成/0/。

示例

原始代码为：

```
DATA test/' '/
```

需修改为：

```
DATA test/0/
```

2.3.1.10 C 调用 Fortran 内置函数 iargc_

说明

每一种Fortran编译器，内置函数在API层面的名字和Fortran函数原始名可能存在差异，ifort （Intel Fortran编译器）中iargc在底层的API是iargc_，而在GCC Fortran中底层的API是_gfortran_iargc，迁移到鲲鹏平台上时，需要将C源码中调用iargc的地方将iargc_替换成_gfortran_iargc。

处理方法

当工具检查发现工程中包括C/C++和Fortran源码，且C/C++源文件中有iargc_字符串时，在iargc_处提示修改，该建议支持quick fix进行修复。

示例

原始代码为：

```
xargc = iargc_()+1;
```

需修改为：

```
#ifdef __GFORTRAN__
   xargc = _gfortran_iargc()+1;
#else
   xargc = iargc_()+1;
#endif
```

2.3.1.11 宏定义大小写

说明

ifort的fpp预处理器可接受大写的#IFDEF #ENDIF,但是GCC Fortran调用的cpp不接受大写，只能用小写，迁移到鲲鹏平台上时需要进行修改。

处理方法

检测到大写的关键字时，工具在Fortran源码文件对应位置，提示用户将#IFDEF修改为#ifdef，将#ENDIF修改为#endif.。

示例

原始代码为：

```
#IFDEF XXX
```

需要修改为：

```
#ifdef xxx
```

原始代码为：

```
#ENDIF
```

#ENDIF

需要修改为：

```
#endif
```

2.3.1.12 open recl 操作

说明

Fortran读取二进制文件，open函数中参数recl代表每次读的记录长度，根据编译器的不同，记录长度的单位不同，有可能以1字节为单位，也有可能以4字节为单位，在迁移到鲲鹏平台的时候，建议用户检查修改。

处理方法

对于open函数，当open函数中的参数含有access='direct’，form='unformatted'且recl参数未乘以4，工具会给出下面3条建议：

1. 鲲鹏使用GNU编译器，大概率需要乘以4；
2. 用户需要查看自己使用的编译器参考手册，才能有明确的结论；
3. 分别测试乘以4与不乘4的情况，排查一下，只有其中一种才可以正常编译运行。

示例

原始代码为：

```
open(1,file='Emis_inv/'//trim(filename), recl=nx*ny, access='direct', form='unformatted')
```

可能需要修改为：

```
open(1,file='Emis_inv/'//trim(filename), recl=nx*ny*4, access='direct', form='unformatted')
```

Kunpeng Computing

2.3.1.13 Fortran 行宽限制

说明

Fortran分为固定格式和自由格式。固定格式中，每行字符长度不能超过72个字符长度，第73个字符之后不使用。自由格式中，每行可以有132个字符。两种格式中任意一种超过其字符长度限制都会被忽略，GCC Fortran编译器对这种情况会报错。

处理方法

固定格式：后缀名是.f, .for, .fpp, .ftn, .F, .FOR, .FPP和.FTN的文件。

自由格式：后缀名是.f90, .f95, .f03, .f08, .F90, .F95,F08，.F03的文件。

工具会根据文件后缀名对文件格式进行判断，如果超过其格式对应的最大长度，会给出修改建议。

示例

如果xxx.f文件中源码的某一行长度超过72个，则工具给出建议如下：

"The line width exceeds the maximum."。

2.3.1.14 过滤注释内容

说明

fortran文件中，迁移到鲲鹏平台时，注释内的内容不会被检查。

注释符号说明如下：

固定格式：

1. 第1个字符，如果是字母C、c或是*，这一行文字 会被当成注释；
2. 惊叹号“！”后面的文字都是注释；

自由格式：

惊叹号“！”后面的文字都是注释。

处理方法

工具会将注释内的内容过滤掉，不作检查，也不进行迁移处理。

示例

test.f文件中有源码如下：

```
C      IBITS(-1,0,52)
```

此行就为注释内容，不会被扫描出来

test.f95文件中有源码如下：

```
! IBITS(-1,0,52)
```

此行就为注释内容，不会被扫描出来

Kunpeng Computing

2.3.2 Fortran 特性函数

描述

Intel Fortran编译器部分built-in函数在鲲鹏平台上是不兼容的，当需要迁移到鲲鹏平台上时，需要对这些函数进行修改。

处理方法

工具能识别这些built-in函数在fortran文件中所在行并给出模糊的提示建议。

示例

AIMAX0() ，在鲲鹏平台不支持，需要用户自行确认修改。

提示建议为："The GCC Fortran compiler does not support this function. Find and replace the function."。

2.4 汇编

工具支持对x86平台高频汇编代码的自动转换功能，可以将x86汇编代码转换为鲲鹏汇编代码，当前支持70%常用场景下的汇编代码转换，随着版本更新完善，后续将达到x86汇编指令的全覆盖。

2.4.1 内嵌汇编

ARM的汇编语言与x86完全不同，需要重写，涉及使用嵌入式汇编的代码，都需要针对ARM进行配套修改。

工具会识别C/C++源码文件中的嵌入式汇编代码段，如果该汇编代码段属于x86汇编代码，对于工具支持的转换场景，就会给出迁移建议。

给出的迁移建议场景大致分为四类：**有明确替换建议**、**提示增加头文件建议**、**模糊提示建议**、**无替换建议**。

2.4.1.1 内嵌汇编代码段建议-明确替换建议

说明

如果C/C++源码中存在内嵌汇编指令，则需要进行适配鲲鹏平台的兼容修改，才能保证代码正常工作在鲲鹏平台上。

处理方法

这种场景下，工具会提示用户在源代码中加上aarch64分支，将适配aarch64的等效的汇编代码添加到aarch64分支内。

示例

原始代码如下：

```
longintmulti_inst(longintdata1,longintdata2)
{
    longintresult=0;
    __asm____volatile("nop\n\t"
    "movq%[data1],%%rax\n\t"
    "movl$4,%%ecx\n\t"
    "addq%%rax,%%rax\n\t"
    "orq%[data2],%%rax\n"
    "movq%%rax,%0\n"
    :"=r"(result)
    :[data1]"r"(data1),[data2]"r"(data2)
    :"rax","ecx","cc","memory");
    returnresult;
}
```

需要修改为：

```
longintmulti_inst(longintdata1,longintdata2)
{
    longintresult=0;
    #ifdefined__x86_64__
        __asm____volatile("nop\n\t"
```

Kunpeng Computing

```
        "movq%[data1],%%rax\n\t"
        "movl$4,%%ecx\n\t"
        "addq%%rax,%%rax\n\t"
        "orq%[data2],%%rax\n"
        "movq%%rax,%0\n"
        :"=r"(result)
        :[data1]"r"(data1),[data2]"r"(data2)
        :"rax","ecx","cc","memory");
    #elif defined(__aarch64__)
        __asm__(
        "movx0,xzr\n\t"
        "ldrx1,%[ARG1_64]\n\t"
        "ldrx2,%[ARG2_64]\n\t"
        "movx3,%[ARG0_64_output]\n\t"
        "orrx8,x2,x1,lsl#1\n\t"
        "strx8,[x3]\n\t"
        "b.Lfunc_end_kpt_1\n\t"
        ".Lfunc_end_kpt_1:\n\t"
        :
        :[ARG1_64]"m"(data1),[ARG2_64]"m"(data2),[ARG0_64_output]"r"(&result)
        :"x0","x1","x2","x3");
    #endif
returnresult;
}
```

2.4.1.2 内嵌汇编代码段建议-增加头文件

说明

对于内嵌汇编内的汇编指令，如果包含一些特殊指令，例如“cpuid”指令，如果迁移到鲲鹏平台上，需要加上头文件"KunpengTrans.h"才能保证工具提供的转换后的内嵌汇编代码可以工作在鲲鹏平台上。

处理方法

工具会在代码文件开头的部分提示用户加上如下语句块：

```
#if defined(__aarch64__)
    #include"KunpengTrans.h"
#endif
```

示例

```
voidCPUID_CASE_PASS_1(unsignedintV4,unsignedintV5)
{
    unsignedintV0,V1,V2,V3;
    asmvolatile("cpuid\n\t":"=a"(V0),"=b"(V1),"=c"(V2),"=d"(V3):"0"(V4),"c"(V5):"cc");
}
```

Kunpeng Computing

需要修改为：

在源码文件最上方加上：

```
#if defined(__aarch64__)
    #include"KunpengTrans.h"
#endif
```

源码对应位置代码修改为：

```
voidCPUID_CASE_PASS_1(unsignedintV4,unsignedintV5)
{
    unsignedintV0,V1,V2,V3;
    #if defined（__x86_64__）
    asmvolatile("cpuid\n\t":"=a"(V0),"=b"(V1),"=c"(V2),"=d"(V3):"0"(V4),"c"(V5):"cc");
    #elif defined(__aarch64__)
```

```
    {
        V0=V4;
        V2=V5;
        GetSupportedCPUID(&V0,&V1,&V2,&V3);
    }
    #endif
}
```

2.4.1.3 内嵌汇编代码段建议-模糊提示

说明

C/C++代码内嵌的汇编指令，在迁移到鲲鹏平台时，需要适配aarch64进行修改，工具对其中部分汇编指令是不能明确准确的替换代码的，需要用户根据工具识别的代码块，结合上下文和用户自己的汇编知识背景进行人工介入，判断是否可以直接使用工具给出的建议源代码。

处理方法

工具会建议用户加上aarch64分支，并将适配aarch64的汇编代码以注释的形式添加到aarch64分支内。需要用户根据迁移建议中的提示，结合上下文检查汇编代码，判断建议中的汇编代码是否可以直接在鲲鹏平台上使用。若用户判断可以直接使用，可将注释符号删除。在不确定的情况下，用户也可以先按照建议修改，手工编译，编译后再根据编译情况及功能测试情况进一步修改。

示例

```
unsigned int swap_big_endian(unsigned int data)
{
    unsigned int result = 0;
    /* 汇编倒序指令 */
    __asm__("bswap %0" : "=r" (result) : "0" (data));
    return result;
}
```

需要修改为：

```
unsigned int swap_big_endian(unsigned int data)
{
    unsigned int result = 0;
    /* 汇编倒序指令 */
    #if defined  __x86_64__
        __asm__("bswap %0" : "=r" (result) : "0" (data));
    #elif defined(__aarch64__)
    // Description: Replace with the converted code block suggested. Note:
    // note 1: Check whether the C/C++ variable expressions associated with the embedded assembly have
side   effect. The side effect expressions need to be modified in the new code to be used.
    // Suggestion:
    //  {
    // result = data;
    // __asm__ __volatile__ ("rev %w0, %w1"
    //  :"=r"(result)
    //  :"0"(result)
    //  );
    // }
    #endif
    return result;
}
```

2.4.1.4 内嵌汇编代码段建议-无替换建议

说明

C/C++代码内嵌的汇编指令，在迁移到鲲鹏平台时，可能会有部分汇编指令是工具暂时无法提供替换建议的，此时工具会提示用户手动处理。

处理方法

对于暂时无法提供可直接替换的代码的场景，工具会提示用户此处存在内嵌汇编需要修改，需要用户自行判断和修改。

示例

```
void case017(void)
{
    int data1 = 8;
    int result = 2;
    __asm__ __volatile__(
    "nop\n\t"
    "pushf\n\t"               // eflags ----> 栈.
    "sal %[data1]\n\t"
    "adc %[data1], %eax\n\t"
    "popf\n\t"            // 栈顶 ----> eflags
    :"+r"(result)
    :[data1]"r"(data1));
    cout <<"result = " <<result << endl;
}
```

提示修改如下：note 1: The assembly code contains a global variable or global symbol, and cannot be automatically converted. Manually convert the code or modify the variable or symbol, and then convert the code again. See https://support.huaweicloud.com/ug-pa-kunpengdevps/kunpengpt_06_0126.html#section4 for reference.

2.4.2 全汇编

说明

Kunpeng Computing

工具会将X86汇编文件转换为鲲鹏平台支持的汇编文件，用户可以利用工具转换的结果对整个文件进行替换。

示例

x86源代码文件代码如下：

```
.text
.globltest001
.typetest001,@function
test001:
mov(%rdi),%ebx
movl$1,%eax
movl$.Overhere,%edx
jmp*%rdx
.L1:
mov%ebx,(%rsi)
ret
.Overhere:
movl$20,%ebx
jmp.L1
```

对应到ARM平台整个文件需要改写成：

```
.text
.file"llvm-link"
.p2align4//--Beginfunctionsub_0_test001_wrapper
.typesub_0_test001_wrapper,@function
sub_0_test001_wrapper://@sub_0_test001_wrapper
//%bb.0:
movx0,x7
movw8,#20
strq0,[sp,#-16]!
ldpx9,x10,[sp],#16
adrpx11,test001
addx11,x11,:lo12:test001
strw8,[x1]
movw8,#1
stpx9,x10,[x6,#-32]
andx11,x11,#0xffffffff
stpx11,x8,[x6,#-16]
ret
.Lfunc_end0:
.sizesub_0_test001_wrapper,.Lfunc_end0-sub_0_test001_wrapper
//--Endfunction
.globltest001//--Beginfunctiontest001
.p2align2
.typetest001,@function
test001://@test001
//%bb.0:
stpx22,x21,[sp,#-32]!//16-byteFoldedSpill
stpx20,x19,[sp,#16]//16-byteFoldedSpill
mrsx8,TPIDR_EL0
addx10,x8,:tprel_hi12:simulation_stack
movw9,#8388480
addx10,x10,:tprel_lo12_nc:simulation_stack
addx8,x8,:tprel_hi12:simulation_stack_pointer
addx9,x10,x9
adrpx10,__unnamed_1
addx8,x8,:tprel_lo12_nc:simulation_stack_pointer
addx10,x10,:lo12:__unnamed_1
//APP
sturx30,[sp,#-16]
subsp,sp,#16//=16
movx21,x8
movx22,x9
ldrx20,[x21]
subx20,x22,x20
movx6,sp
strx6,[x20]
movsp,x20
ldrx19,[x10]
blrx19
movx20,sp
subx20,x22,x20
strx20,[x21]
ldrx19,[sp]
movsp,x19
ldrx30,[sp]
ldurx0,[sp,#-8]
ldurx1,[sp,#-16]
ldurq0,[sp,#-32]
addsp,sp,#16//=16
//NO_APP
ldpx20,x19,[sp,#16]//16-byteFoldedReload
ldpx22,x21,[sp],#32//16-byteFoldedReload
ret
.Lfunc_end1:
.sizetest001,.Lfunc_end1-test001
//--Endfunction
.type__unnamed_1,@object//@0
.section.rodata,"a",@progbits
```

Kunpeng Computing

```
.p2align3
__unnamed_1:
.xwordsub_0_test001_wrapper
.size__unnamed_1,8
.typesimulation_stack,@object//@simulation_stack
.section.tbss.simulation_stack,"aGwT",@nobits,simulation_stack,comdat
.globlsimulation_stack
.p2align4
simulation_stack:
.zero8388608
.sizesimulation_stack,8388608
.typesimulation_stack_pointer,@object//@simulation_stack_pointer
.section.tbss.simulation_stack_pointer,"aGwT",@nobits,simulation_stack_pointer,comdat
.globlsimulation_stack_pointer
.p2align3
simulation_stack_pointer:
.xword0//0x0
.sizesimulation_stack_pointer,8
.ident"clangversion10.0.1"
.ident"clangversion10.0.1"
.section".note.GNU-stack","",@progbits
```

2.5 Python

工具目前支持Python语言的扫描，Python源码文件中如果存在调用动态链接库相关的内容，则需要考虑对鲲鹏平台的兼容性。

2.5.1 识别 Python 代码中加载动态库函数

说明

python在调用动态库的时候，需要使用LoadLibrary函数进行加载。对于识别出的函数源码，如果对应加载的so文件不在工具内置的依赖文件列表中，工具会提示用户对程序中所用到的so是否在鲲鹏平台支持进行确认。用户如果有这些so的源代码，也可以利用工具提供的C/C++/汇编源码分析能力对这些源码进行分析。

2.5.2 识别 Python 调用的依赖库

说明

工具能识别Python代码中的依赖库文件，对依赖库文件进行兼容性检查，对兼容鲲鹏平台的so库文件提供可供用户下载的地址；否则，在分析报告中告知用户“该so库文件暂不兼容鲲鹏平台，需要用户检查、迁移并重新编译”。

示例

python源码：

```
ctypes.cdll.LoadLibrary("/path/to/the/so/name.so")
```

中会识别出name.so依赖库。

如果name.so在工具内置的依赖文件列表中，则直接提供下载链接供用户下载依赖替换。如果不在，则提醒用户需要迁移到鲲鹏平台，并提醒用户需要自行验证替换。

3 内存一致性检查

说明

不同的CPU架构使用的内存模型是不同的。内存模型定义了CPU读写内存的规则及指令执行序列的规则。例如：内存读写原子操作的规则，CPU执行指令顺序是否与程序指令顺序一致等。在多线程程序中，同样的代码，因内存模型不同会出现不同的程序行为。

顺序性最严格的是“顺序一致性“（SC），通常被认为只是一种理想模型，也是程序员最自然的工作模式，但是SC模型会导致CPU执行效率的低下，因此现在没有SC模型的处理器。SC模型允许程序对应的指令按照它们在源码中的顺序执行，并且在多线程场景下，每个单个线程的执行只是以某种顺序交替。

比SC宽松的内存模型是x86的TSO（Total Store Order）内存模型。在该模型下，所有处理器单元都只能保证指令写入到共享内存的顺序，以此来保证指令执行的总体顺序。但TSO模型运行store-load指令仍然存在乱序的可能性。

最宽松的内存模型是RMM （Relax Memory Model），ARM（含鲲鹏）和Power都采用RMM内存模型。在该模型下，各个处理器从自己本地存储中读取数据，并将每个写入，各自广播到其它处理器，并且允许内存写入操作重新排序。

不同的内存模型对内存读写指令顺序交错的容忍度不同，列表1可直观看出。

类型	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	z/Architecture
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	

Kunpeng Computing

在非SC模型下，可以通过内存屏障指令避免内存读写指令的乱序，例如：x86架构是mfence指令，ARM架构是dmb指令。

处理方法

x86平台采用了TSO内存模型，经过长时间的软件生态积累，用户软件都是基于x86的TSO内存模型开发的，导致这些软件，特别是多线程软件在移植到鲲鹏平台后出现程序运行异常。这时，用户可以重点检查全局共享变量的读写操作是否通过锁、信号量进行了保护；或者在lock-free的多线程程序中对全局变量的读写通过内存屏障指令进行保序处理。

[3.1 内存一致性静态检查](#)

[3.2 编译器自动修复](#)

3.1 内存一致性静态检查

鲲鹏开发套件中的代码迁移工具Porting Advisor具备内存一致性检查能力，通过clang编译器编译用户源码产生中间代码（IR）文件（即bytecode或者说.bc文件），对该IR文件进行内存一致性静态检查，并生成报告文件，给出用户迁移提示。用户也可以自行使用Clang/LLVM手动编译产生的单个IR文件，上传到Porting Advisor进行内存一致性检查，并生成报告文件。

示例

对test-gllvm软件包(test-gllvm.zip)进行分析。

执行编译后，使用给gllvm生成BC文件，利用工具对BC文件进行检查，检查成功后，查看报告，报告中需要修改的代码行如**表3-1**所示：

表 3-1 结果展示

<table>
<tr><th>序号</th><th>文件</th><th>代码行</th><th>处理建议</th><th>修改后的代码</th></tr>
<tr><td>1</td><td>test007.c</td><td>my_city->sync_flag = true; // 52行</td><td rowspan="4">Suggestion: add"__asm__ volatile("dmb sy")"in the position indicated by the below items.</td><td>__asm__ volatile("dmb sy");
my_city->sync_flag = true;</td></tr>
<tr><td>2</td><td>test007.c</td><td>printf("The population of BeiJing is %d.\n",my_city->Population);//62行</td><td>__asm__ volatile("dmb sy");
printf("The population of BeiJing is %d.\n",my_city->Population);</td></tr>
<tr><td>3</td><td>test008.c</td><td>my_city->sync_flag = true; 75行</td><td>__asm__ volatile("dmb sy");
my_city->sync_flag = true;</td></tr>
<tr><td>4</td><td>test008.c</td><td>printf("The population of BeiJing is %d.\n",my_city->Population);85行</td><td>__asm__ volatile("dmb sy");
printf("The population of BeiJing is %d.\n",my_city->Population);</td></tr>
</table>

在工具检测出存在内存一致性问题的地方，会给出添加内存屏障指令的建议，用户可以根据建议、结合个人经验和知识背景进行处理，从而解决软件中存在的内存一致性问题。

 版权所有 © 华为技术有限公司

3.2 编译器自动修复

通过编译器自动修复工具，可以确保编译出的程序中修复了用户程序中存在的弱内存序bug。用户使用前，先安装编译器自动修复工具，该工具会以补丁形式对用户使用的gcc编译器组件进行优化。安装完成后，用户在编译自己的程序时通过附加的指令参数激活编译器自动修复工具功能，从而保证在编译过程中，编译器自动修复功能自动发挥作用。

Kunpeng Computing

4 64 位运行模式检查

64位运行模式检查功能可以在用户需要将原32位平台上的软件迁移到64位平台时，对用户源代码进行迁移预检并给出修改建议。以下列表是32位和64的部分数据模型对比，如表4-1所示：

表 4-1 32 位和 64 位数据字节对比

Data type	Data length（32bit）	Date length（64bit）	Signed
char	8	8	Y
Unsigned char	8	8	N
short	16	16	Y
unsigned short	16	16	N
char*	32	64	N
short int	16	16	Y
int	32	32	Y
unsigned int	32	32	N
float	32	32	Y
double	64	64	Y
long	32	64	Y
unsigned long	32	64	N
long long	64	64	Y
unsigned long	32	64	N

Kunpeng Computing

由上表我们可以看出，32位到64位的迁移工作，主要就是处理长度变化所引发的各种问题。在32位平台上很多正确的操作，在64位平台上都不再成立。例如：long->int等，会出现截断问题等，而这些问题最终会导致程序运行异常，甚至崩溃。

示例

扫描源码包，包中有如下文件：func001.c（图4-1）、func002.c（图4-2）、func003.c（图4-3）、func004.c（图4-4）、Makefile（图4-5）。

图 4-1 func001.c 源码

```c
#include <stdio.h>

// -Wconversion
void func001(void)
{
    char c = 0;
    int i = 10;

    int m = 20;
    char a = m;

    c = c+i;
}

int main(int argc, char *argv[])
{
    func001();

    return 0;
}

```

图 4-2 func002.c 源码

```c
#include <stdio.h>

//-Wsign-conversion

void func002(void)
{
    char *c = NULL;
    int *p = NULL;
    char *s = p;

    c = p;
}

int main(int argc, char *argv[])
{
    func002();

    return 0;
}

```

Kunpeng Computing

图 **4-3** func003.c 源码

```
#include <stdio.h>

//-Wpointer-sign
void func003(void)
{
    char *c = NULL;
    unsigned char *s  = c;
}


int main(int argc, char *argv[])
{

    func003();

    return 0;
}

```

图 **4-4** func004.c 源码

```
#include <stdio.h>

//-Wint-conversion
void func004(void)
 {
     int test = 10;
     char s[] = "hello";
     int a = 10;
     int q = &a; /* 此处应有提示 */
     int p;

     p = &a; /* 此处应有提示 */

     test = s;
 }

int main(int argc, char *argv[])
{
    func004();
    return 0;
}

```

Kunpeng Computing

图 4-5 Makefile 源码

```

prog=func001 func002 func003 func004
cleanfile=hello *.o *.out
CC=gcc
CFLAGS = -Wconversion -Wint-to-pointer-cast

all:${prog}

func001: func001.o
	${CC} ${CFLAGS} -o $@ func001.c

func002: func002.o
	${CC} ${CFLAGS} -o $@ func002.c

func003: func003.o
	${CC} ${CFLAGS} -o $@ func003.c

func004: func004.o
	${CC} ${CFLAGS} -o $@ func004.c

.PHONY: clean
clean:
	rm -f ${cleanfile} ${prog}

```

执行make命令后，得到如图4-6warning提示信息，

Kunpeng Computing

图 **4-6** make 执行结果

```
[root@localhost demo]# make
gcc -Wconversion -Wint-to-pointer-cast   -c -o func001.o func001.c
func001.c: In function ‘func001’:
func001.c:10:5: warning: conversion to ‘char’ from ‘int’ may alter its value [-Wconversion]
     char a = m;
     ^
func001.c:12:10: warning: conversion to ‘char’ from ‘int’ may alter its value [-Wconversion]
     c = c+i;
          ^
gcc -Wconversion -Wint-to-pointer-cast -o func001 func001.c
func001.c: In function ‘func001’:
func001.c:10:5: warning: conversion to ‘char’ from ‘int’ may alter its value [-Wconversion]
     char a = m;
     ^
func001.c:12:10: warning: conversion to ‘char’ from ‘int’ may alter its value [-Wconversion]
     c = c+i;
          ^
gcc -Wconversion -Wint-to-pointer-cast   -c -o func002.o func002.c
func002.c: In function ‘func002’:
func002.c:11:7: warning: assignment from incompatible pointer type [enabled by default]
     c = p;
       ^
gcc -Wconversion -Wint-to-pointer-cast -o func002 func002.c
func002.c: In function ‘func002’:
func002.c:11:7: warning: assignment from incompatible pointer type [enabled by default]
     c = p;
       ^
gcc -Wconversion -Wint-to-pointer-cast   -c -o func003.o func003.c
gcc -Wconversion -Wint-to-pointer-cast -o func003 func003.c
gcc -Wconversion -Wint-to-pointer-cast   -c -o func004.o func004.c
func004.c: In function ‘func004’:
func004.c:9:14: warning: initialization makes integer from pointer without a cast [enabled by default]
      int q = &a; /* 此处应有提示 */
              ^
func004.c:12:8: warning: assignment makes integer from pointer without a cast [enabled by default]
      p = &a; /* 此处应有提示 */
        ^
func004.c:14:11: warning: assignment makes integer from pointer without a cast [enabled by default]
      test = s;
           ^
gcc -Wconversion -Wint-to-pointer-cast -o func004 func004.c
func004.c: In function ‘func004’:
func004.c:9:14: warning: initialization makes integer from pointer without a cast [enabled by default]
      int q = &a; /* 此处应有提示 */
              ^
func004.c:12:8: warning: assignment makes integer from pointer without a cast [enabled by default]
      p = &a; /* 此处应有提示 */
        ^
func004.c:14:11: warning: assignment makes integer from pointer without a cast [enabled by default]
      test = s;
           ^
```

对warning信息进行处理，得到预检报告。具体操作如下：

“func001.c:10:5: warning: conversion to ‘char’ from ‘int’ may alter its value [-Wconversion]

char a = m;”

上段源码对应的提示信息：在第10行源码处提示：Suggestion: // This line needs to be adapted for the 64-bit environment

表4-2是所有的分析结果。

表 **4-2** 代码修改处理

序号	文件	代码行	处理建议	修改后的代码
1	func001.c	第10行char a = m;	Suggestion: // This line needs to be adapted for the 64-bit environment.	char a = (char)m;
2	func001.c	第12行c = c+i;		c = (char)((int)c +i);
3	func002.c	第11行c = p;		c = (char *)p;
4	func003.c	第7行unsigned char *s = c;		unsigned char *s = (unsigned char *)c;

序号	文件	代码行	处理建议	修改后的代码
5	func004.c	第9行int q = &a;		long int q = (long)&a;
6	func004.c	第12行p = &a;		long int p; p = (long)&a;
7	func004.c	第14行test = s;		long int test = 10; test = (long)s

按照要求修改后，重新执行make，显示结果图4-7：

图 4-7 make 执行结果

```
[root@localhost demo]# make
gcc -Wconversion -Wint-to-pointer-cast   -c -o func001.o func001.c
gcc -Wconversion -Wint-to-pointer-cast -o func001 func001.c
gcc -Wconversion -Wint-to-pointer-cast   -c -o func002.o func002.c
gcc -Wconversion -Wint-to-pointer-cast -o func002 func002.c
gcc -Wconversion -Wint-to-pointer-cast   -c -o func003.o func003.c
gcc -Wconversion -Wint-to-pointer-cast -o func003 func003.c
gcc -Wconversion -Wint-to-pointer-cast   -c -o func004.o func004.c
gcc -Wconversion -Wint-to-pointer-cast -o func004 func004.c
```

从中可以看出和32-64迁移相关的warning信息已经全部清除。

Kunpeng Computing

5 结构体字节对齐检查

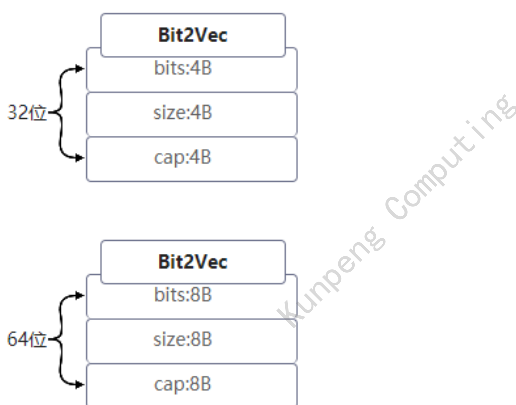
结构体字节对齐检查功能可以检查源码中结构体类型变量的字节对齐情况，方便用户在将源码从32位迁移到64位机器的时候，进行结构体定义优化，从而节约系统内存资源。

用户使用该功能对自己的源码分析后，将得到一份结构体字节对齐检查报告，从报告中可以看到每个结构体定义的位置及分别在32位和64位两种运行模式下的内存占用情况，以便进行代码优化。

5.1 结构变量内存空间分配不需要优化类型
5.2 可进行结构变量内存空间分配优化类型

5.1 结构变量内存空间分配不需要优化类型

图 5-1 32 位与 64 位结构体比对



Kunpeng Computing

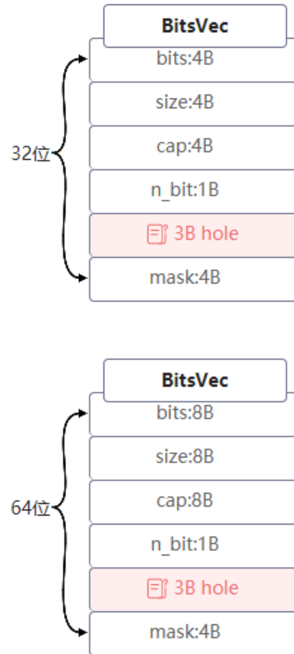
BitVec结构体源码：

```
typedef struct {
    uint8_t *bits;
    uint64_t size;
    uint64_t cap;
} Bit2Vec;
```

从**图5-1**可以看出Bit2Vec结构体从32位机器到64位机器，内存分配合理，不需要进行优化。

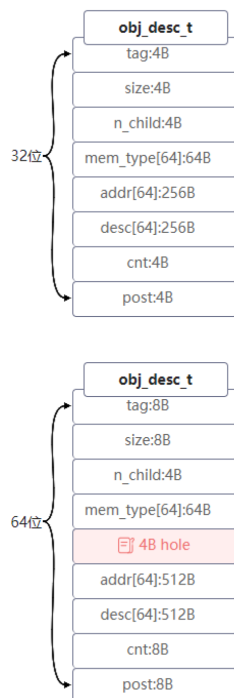
5.2 可进行结构变量内存空间分配优化类型

图 5-2 32 位和 64 位结构体比对



从图5-2可以看出BitsVec结构体无论是32位还是64位，都有可以优化的点。

图 5-3 32 位和 64 位结构体比对



Kunpeng Computing

图5-3中， obj_desc_t从32迁移到64位，可进行优化。

Kunpeng Computing

6 软件迁移评估

描述

rpm及deb等格式的Linux应用软件二进制安装包，是与操作系统版本及CPU结构相关的，因此基于x86服务器的二进制安装包，是无法直接在基于鲲鹏处理器制造的服务器上使用的。

软件迁移评估是鲲鹏代码迁移工具的一项重要功能，它可以检查x86平台软件安装包或已安装软件中使用的动态链接库、静态链接库和可执行文件，并将检查出的文件与工具内置的依赖文件列表进行比较匹配，从而为用户提供迁移建议。

处理方法

如果用户软件中的依赖文件是存在于工具内置的依赖文件列表中的，则提示用户这些文件可兼容、可直接替换；否则，工具会提示用户自行分析这些文件的鲲鹏平台兼容性。

目前软件迁移评估功能支持分析的软件安装包类型有："．rpm"，“.deb”,“.zip”,“.tar”，“.tar.gz”，“.tar.xz”，“.tar.bz”,“.bz2”,“.tgz”，“.tbz”,“.jar”,“.war”，“.egg”，“.whl”。

可检出的x86平台依赖文件类型有JAR包、so文件、.a文件及其它类型的二进制文件。

下文将分章节分别描述对rpm包、jar/war包、已安装软件、其它类型安装包的分析。

6.1 RPM包分析

6.2 Jar/War包分析

6.3 已安装软件分析

6.4 其它软件包迁移评估分析

Kunpeng Computing

6.1 RPM 包分析

软件迁移评估功能可以分析用户提供的rpm包。分析完，工具会输出软件迁移评估报告。

示例

以分析hadoop_2_6_3_0_235-2.7.3.2.6.3.0-235.x86_64.rpm为例（下载地址：**https://public-repo-1.hortonworks.com/HDP/centos7/2.x/updates/2.6.3.0/HDP-2.6.3.0-centos7-rpm.tar.gz**），

对软件包进行解压，部分结果如下：

```
/usr/hdp/2.6.3.0-235/etc/bash_completion.d/hadoop
/usr/hdp/2.6.3.0-235/etc/default/hadoop
/usr/hdp/2.6.3.0-235/etc/hadoop/conf.empty/capacity-scheduler.xml
/usr/hdp/2.6.3.0-235/etc/hadoop/conf.empty/configuration.xsl
/usr/hdp/2.6.3.0-235/etc/hadoop/conf.empty/container-executor.cfg
/usr/hdp/2.6.3.0-235/etc/hadoop/conf.empty/core-site.xml
/usr/hdp/2.6.3.0-235/etc/hadoop/conf.empty/hadoop-env.cmd
/usr/hdp/2.6.3.0-235/etc/hadoop/conf.empty/hadoop-env.sh
.........
```

解压出的文件中，有JAR包、动态库、静态库、可执行文件等，工具会对这些数据进行分析，**表6-1**包含该四类文件：

表 6-1 工具扫描结果

依赖文件名	文件类型	路径	待下载软件包名称	分析结果	处理建议
libsnappy.so.1.1.4	动态库	hadoop_2_6_3_0_235-2.7.3.2.6.3.0-235.x86_64.rpm/usr/hdp/2.6.3.0-235/hadoop/lib/native/libsnappy.so.1.1.4	snappy-1.1.0-3.el7.aarch64.rpm	可兼容替换	扫描报告中会给出待下载软件包的下载链接，待下载rpm包内包含aarch64架构的libsnappy.so.1.1.4动态库
snappy-java-1.0.4.1.jar	Jar包	hadoop_2_6_3_0_235-2.7.3.2.6.3.0-235.x86_64.rpm/usr/hdp/2.6.3.0-235/hadoop/lib/snappy-java-1.0.4.1.jar	snappy-java-1.0.4.1.jar	可兼容替换	扫描报告中会给出待下载软件包的下载链接，待下载Jar包可直接替换snappy-java-1.0.4.1.jar
libhadoop.a	静态库	hadoop_2_6_3_0_235-2.7.3.2.6.3.0-235.x86_64.rpm/usr/hdp/2.6.3.0-235/hadoop/lib/native/libhadoop.a	--	待验证替换	请先在鲲鹏平台上验证。若不兼容，请联系供应方获取鲲鹏兼容版本，或获取源码并编译成鲲鹏兼容版本
container-executor	可执行文件	hadoop_2_6_3_0_235-2.7.3.2.6.3.0-235.x86_64.rpm/usr/hdp/2.6.3.0-235/hadoop/mapreduce.tar.gz/hadoop/bin/container-executor			

依赖文件名	文件类型	路径	待下载软件包名称	分析结果	处理建议
spark2_2_6_3_0_235-yarn-shuffle	可执行文件	无，该可执行文件是从当前软件包中获取的依赖信息			

其中libsnappy.so.1.1.4动态库在鲲鹏平台不支持运行，但工具发现snappy-1.1.0-3.el7.aarch64.rpm这个包中存在鲲鹏兼容版本的同名so文件，用户可以直接下载，并替换原有包中文件。同时这个依赖库不仅是在/package/hadoop_2_6_3_0_235-2.7.3.2.6.3.0-235.x86_64.rpm/usr/hdp/2.6.3.0-235/hadoop/lib/native/libsnappy.so.1.1.4路径下，也在/package/hadoop_2_6_3_0_235-2.7.3.2.6.3.0-235.x86_64.rpm/usr/hdp/2.6.3.0-235/hadoop/mapreduce.tar.gz/hadoop/lib/native/libsnappy.so.1.1.4路径下，所以这两处都需要对依赖库进行替换。而libhadoop.so.1.0.0动态库，在工具内置的依赖文件中查找不到，对于它在鲲鹏平台是否兼容，需要用户进一步验证。

对于DEB包，分析过程和RPM一致，差别在于最后的扫描报告展示，对于需要替换的依赖可能会有DEB包替换，如**图6-1**所示：

图 6-1 DEB 包扫描报告

序号	依赖文件名	文件类型	路径	待下载软件包名称	分析结果	处理建议
1	rpm2cpio	可执行文件	/package/deb_all.deb/bin/rpm2cpio	file-roller_3.28.0-1ubuntu1_arm64.deb	可兼容替换	下载 | 复制链接
2	libleveldb.so.1.13	动态库	/package/deb_all.deb/libleveldb.so.1.13	libleveldb1v5_1.20-2_arm64.deb	可兼容替换	下载 | 复制链接
3	libssl3.so	动态库	/package/deb_all.deb/libssl3.so	libnss3_3.35-2ubuntu2_arm64.deb	可兼容替换	下载 | 复制链接
4	libleveldb.so.1.13	动态库	/package/deb_all.deb/test.tar/usr/libleveldb....	libleveldb1v5_1.20-2_arm64.deb	可兼容替换	下载 | 复制链接
5	libssl3.so	动态库	/package/deb_all.deb/test.tar/usr/libssl3.so	libnss3_3.35-2ubuntu2_arm64.deb	可兼容替换	下载 | 复制链接

6.2 Jar/War 包分析

软件迁移评估在处理软件包为Jar/War包的时候，会对Jar/War包包含的so或者Jar/War包中嵌套的Jar/War包进行分析，并给出依赖分析报告。

示例

以分析package.jar包为例（package.zip解压即可获取），将该软件包进行解压，得到119个文件，部分结果如下：

```
META-INF/MANIFEST.MF
package/jline-2.14.6.jar
package/netty-4.1.zip
package/1.jar
package/META-INF/MANIFEST.MF
package/META-INF/maven/jline/jline/pom.xml
package/META-INF/maven/jline/jline/pom.properties
package/META-INF/native/linux32/libjansi.so
package/META-INF/native/linux64/libjansi.so
……..
```

对解压后的文件进行分析处理，如果是jar包，可进行再次解压，直到统计完所有的文件。工具会对JAR包中的JAR包、动态库、静态库、可执行文件，与工具内置的依赖文件列表进行比较匹配，如果Jar包未匹配到，工具会对嵌套的JAR包内部的文件进行同

样的匹配操作。如果Jar包匹配到了则可直接替换整个Jar包（War同理）。详细内容可查看表6-2：

表 6-2 扫描结果示例

依赖文件名	文件类型	路径	待下载软件包名称	分析结果	处理建议
package.jar	Jar包	/package.jar	--	待验证替换	请先在鲲鹏平台上验证。若不兼容，请联系供应方获取鲲鹏兼容版本，或获取源码并编译成鲲鹏兼容版本
libjansi.so(此依赖文件为package.jar内部so文件。)		/package/META-INF/native/linux64/libjansi.so	jansi-native-1.4-11.el7.aarch64.rpm	可兼容替换	扫描报告中会给出待下载软件包的下载链接，待下载rpm包内包含aarch64架构的libjansi.so文件
jline-2.14.6.jar	Jar包	package.jar/package/jline-2.14.6.jar	jline-2.14.6.jar	可兼容替换	扫描报告中会给出待下载软件包的下载链接，待下载Jar包可直接替换jline-2.14.6.jar

6.3 已安装软件分析

工具分析已安装软件时，首先会对安装路径下的所有内容进行扫描，将所有的文件找出，然后对文件进行分析，找出so依赖库和可执行文件，并评估so依赖库和可执行文件的可迁移性。

如分析/usr/bin目录下的已安装软件时，得到图6-2：

Kunpeng Computing

图 6-2 扫描结果

与架构相关的依赖文件

序号	依赖文件名	文件类型	路径	待下载软件包名称	分析结果	处理建议
1	a2p	可执行文件	/usr/bin/a2p	perl-5.16.3-293.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
2	agentxtrap	可执行文件	/usr/bin/agentxtrap	net-snmp-5.7.2-37.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
3	bdftopcf	可执行文件	/usr/bin/bdftopcf	xorg-x11-font-utils-7.5-21.el7.aarch64....	可兼容替换	下载 | 复制链接
4	bdftruncate	可执行文件	/usr/bin/bdftruncate			
5	bzip2	可执行文件	/usr/bin/bzip2	bzip2-1.0.6-13.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
6	bzip2recover	可执行文件	/usr/bin/bzip2recover			
7	c++	可执行文件	/usr/bin/c++	gcc-c%2B%2B-4.8.5-36.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
8	cairo-sphinx	可执行文件	/usr/bin/cairo-sphinx	cairo-1.15.12-3.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
9	ccmake	可执行文件	/usr/bin/ccmake	cmake-2.8.12.2-2.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
10	checksctp	可执行文件	/usr/bin/checksctp	lksctp-tools-1.0.17-2.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
11	cmake	可执行文件	/usr/bin/cmake	cmake-2.8.12.2-2.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
12	consolehelper	可执行文件	/usr/bin/consolehelper	usermode-1.111-5.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
13	cpack	可执行文件	/usr/bin/cpack	cmake-2.8.12.2-2.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
14	cpp	可执行文件	/usr/bin/cpp	cpp-4.8.5-36.el7.aarch64.rpm	可兼容替换	下载 | 复制链接
15	ctest	可执行文件	/usr/bin/ctest	cmake-2.8.12.2-2.el7.aarch64.rpm	可兼容替换	下载 | 复制链接

从上图看出，部分可执行文件有兼容替换的软件包，用户可以从工具提供的下载链接中直接下载使用对应的鲲鹏兼容版本软件。

6.4 其它软件包迁移评估分析

其它软件包类似“.zip”，“.tar”，“.tar.gz”，“.tar.xz”，“.tar.bz”,“.bz2”,“.tgz”，“.tbz”,“.egg”，“.whl”的迁移流程和RPM包扫描流程基本一致，按照文件格式进行对应格式的解压方式进行解压，循环遍历得到的文件，工具对文件进行分析。用户可以根据实际项目需要，上传软件包进行分析，得到相关依赖库的扫描结果，在迁移到鲲鹏平台之前替换依赖文件。

Kunpeng Computing

7 软件包重构

软件包重构功能是代码迁移工具提供的一项功能，该功能可以将用户提供的x86版本rpm包或deb包重构为鲲鹏版本的同类型包，供用户在鲲鹏平台安装使用。

说明

在重构过程中，工具主要对包内包含的.so文件，.a文件和.jar/.war文件进行分析，会对其中需要适配鲲鹏平台的依赖文件进行替换，然后执行软件包的重构，最终输出鲲鹏兼容版本的软件安装包。无论是否重构成功，工具都会输出一份重构报告，用户可以查看报告以了解重构结果。

7.1 RPM包重构
7.2 DEB包重构

7.1 RPM 包重构

软件包重构功能可以分析用户提供的rpm包。

示例

以对parquet-1.5.0+cdh5.12.1+187-1.cdh5.12.1.p0.3.el7.noarch.rpm（下载rebuildrpm.zip，进行解压获取到该rpm包以及依赖文件）包进行重构为例。

用户上传需要重构的软件包和依赖配置文件，然后执行重构。重构完成后，用户可以直接下载重构后的rpm格式软件包，也可以通过查看报告了解具体更新替换的依赖文件。rpm包的重构报告样例如图7-1所示：

图 7-1 重构报告

< parquet-1.5.0+cdh5.12.1+187-1.cdh5.12.1.p0.3.el7.aarch64.rpm

重构信息

软件包存放路径	/opt/portadv/portadmin/packagerebuild/parquet-1.5...
重构报告生成时间	2021-07-26 14:35:34
重构后软件包存放路径	/opt/portadv/portadmin/report/packagerebuild/202...
重构结果	重构成功

已更新依赖文件	缺少依赖文件	依赖文件总数
3	0	3

已更新依赖文件

序号	依赖文件名	路径	文件来源
1	leveldbjni-all-1.8.jar	/usr/lib/parquet/lib/leveldbjni-all-1.8.jar	用户上传
2	snappy-java-1.0.4.1.jar	/usr/lib/parquet/lib/snappy-java-1.0.4.1.jar	用户上传
3	netty-all-4.0.23.Final.jar	/usr/lib/parquet/lib/netty-all-4.0.23.Final.jar	用户上传

下载重构包　下载报告 (.html)

7.2 DEB 包重构

当待重构处理的软件包为deb包时，工具会分析包内包含的依赖库和二进制文件，并对deb包进行依赖文件替换，完成deb包的重构。

Kunpeng Computing

8 专项软件迁移

专项软件迁移功能中提供了大数据、数据库、高性能计算、web等几个领域中部分典型软件迁移的经验工具化集成，用户可以通过工具中提供的操作直接生成相应的软件包，也可以参考对应软件的操作步骤进行手工操作，编译生成需要的软件包。

Kunpeng Computing