Analyse Détaillée du Code CoreMark Pro

CoreMark-PRO est un benchmark de processeur complet qui étend le benchmark CoreMark original en ajoutant la prise en charge du multicœur, des charges de travail en virgule flottante et des jeux de données pour des sous-systèmes de mémoire plus grands. L'architecture du benchmark est basée sur le **Multi-Instance Test Harness (MITH)** d'EEMBC, ce qui facilite son portage sur différentes plateformes.

Structure Générale du Projet

Le projet CoreMark-PRO est organisé en plusieurs répertoires clés :

- `benchmarks/` : Contient les différentes charges de travail (workloads) et les noyaux (kernels) du benchmark. Chaque charge de travail est un programme indépendant qui peut être compilé et exécuté.
- `docs/` : Contient la documentation, y compris le guide utilisateur PDF et un guide de portage baremetal.
- `mith/` : Contient le Multi-Instance Test Harness (MITH), qui est le cadre d'exécution du benchmark. C'est ici que se trouve la couche d'abstraction matérielle (AL).
- `util/` : Contient des utilitaires, notamment les Makefiles spécifiques aux différentes plateformes et chaînes d'outils.
- `LICENSE.md` , `Makefile` , `README.md` : Fichiers de licence, Makefile principal et fichier README.

Le Multi-Instance Test Harness (MITH)

Le MITH est le cœur de CoreMark-PRO. Il fournit un cadre pour l'exécution des benchmarks et une couche d'abstraction pour interagir avec le matériel sous-jacent. Les composants clés du MITH sont :

`mith/al/` (Abstraction Layer)

Ce répertoire contient la couche d'abstraction matérielle/système d'exploitation. C'est le seul endroit où des modifications sont généralement nécessaires pour porter le benchmark sur une nouvelle plateforme. Les fichiers importants ici sont :

- `mith/al/inc/al.h` : Fichier d'en-tête qui définit l'interface de la couche d'abstraction. Il contient les déclarations des fonctions que la plateforme cible doit implémenter (par exemple, gestion des threads, synchronisation, temporisation, entrée/sortie).
- `mith/al/src/th_al.c` : **C'est le fichier le plus important pour le portage.** Il contient les implémentations par défaut des fonctions définies dans `al.h` pour une plateforme POSIX (Linux). Pour un système embarqué comme STM32, la plupart de ces fonctions devront être réimplémentées pour utiliser les primitives du microcontrôleur (par exemple, un RTOS comme FreeRTOS ou des fonctions baremetal).
- `mith/al/src/al_single.c` : Une implémentation de l'AL pour un environnement mono-thread. Utile pour les systèmes baremetal sans support de threading.
- `mith/al/src/al_smp.c` : Une implémentation de l'AL pour les systèmes multi-cœurs (Symmetric Multi-Processing) utilisant des threads POSIX. Pour STM32, si un RTOS est utilisé avec des threads, cette implémentation pourrait servir de base.

Fonctions clés dans `th_al.c` (à adapter pour STM32) :

- `al_main()` : Le point d'entrée de l'AL. C'est ici que l'initialisation de la plateforme (horloges, périphériques, RTOS) devrait avoir lieu.
- `al_signal_init()` , `al_signal_start()` , `al_signal_stop()` , `al_signal_destroy()` : Fonctions liées à la gestion des signaux et des temporisateurs. Pour STM32, cela impliquera probablement la configuration d'un timer d'interruption pour fournir une résolution temporelle (par exemple, 1 ms).
- `al_thread_create()` , `al_thread_join()` , `al_thread_mutex_init()` , `al_thread_mutex_lock()` , `al_thread_mutex_unlock()` , `al_thread_mutex_destroy()` : Fonctions de gestion des threads et des mutex. Si un RTOS est utilisé, ces fonctions devront être mappées aux API du RTOS (par exemple, `osThreadNew` , `osMutexAcquire` , etc.). Si le système est mono-thread, ces fonctions peuvent être des stubs ou utiliser `al_single.c` .

- `al_printf()` : Fonction d'impression. Pour STM32, elle devra être redirigée vers une console de débogage (SWO, UART) ou un semihosting.
- `al_malloc()` , `al_free()` : Fonctions d'allocation mémoire. Peuvent être mappées à `malloc` / `free` de la libc ou à un gestionnaire de mémoire spécifique au RTOS.

`mith/inc/`

Contient les fichiers d'en-tête généraux du MITH, tels que `mith.h` qui définit les structures de données et les macros utilisées par le harnais de test.

`mith/src/`

Contient le code source principal du MITH, y compris `mith_main.c` qui contient la fonction `mith_main()` , le point d'entrée du harnais de test qui orchestre l'exécution des charges de travail.

Charges de Travail (Workloads) et Noyaux (Kernels)

Le répertoire `benchmarks/` contient les différentes charges de travail de CoreMark-PRO. Chaque charge de travail est un programme C qui instancie le harnais de test et exécute un ou plusieurs noyaux de benchmark avec des jeux de données spécifiques.

Exemples de charges de travail :

- `benchmarks/consumer_v2/cjpeg/` : Implémente la compression JPEG. Contient `cjpeg.c` et des fichiers de données BMP.
- `benchmarks/core/` : Une version plus gourmande en mémoire du CoreMark original.
- `benchmarks/fp/linpack/` : Implémente l'élimination de Gauss pour les calculs en virgule flottante.
- `benchmarks/fp/loops/` : Contient les boucles de Livermore.

Important : Selon le `README.md` , il n'est **PAS autorisé** de modifier les fichiers source dans les répertoires `benchmarks/` ou `workloads/` . Toute modification doit se faire dans la couche d'abstraction (`mith/al/src/th_al.c`).

Fichiers de Données

Les jeux de données pour les benchmarks sont inclus directement dans le code source sous forme de structures C (par exemple, `benchmarks/consumer_v2/cjpeg/data/*.c`). Cela signifie qu'il n'y a pas de dépendance à l'E/S de fichiers, ce qui est idéal pour les systèmes embarqués.

Makefiles

Le projet utilise des Makefiles pour la compilation. Le `Makefile` principal dans le répertoire racine gère le processus de construction. Il inclut des fichiers spécifiques à la cible dans `util/make/`.

- `util/make/linux64.mak` : Exemple de fichier de configuration pour Linux 64 bits.
- `util/make/gcc64.mak` : Fichier de configuration pour la chaîne d'outils GCC 64 bits.

Pour STM32 avec IAR, il faudra créer un nouveau fichier `.mak` dans `util/make/` ou adapter un existant, ou plus probablement, créer un projet IAR Workbench et y importer les fichiers source pertinents.

Processus d'Exécution

CoreMark-PRO exécute chaque charge de travail individuellement. Pour les systèmes embarqués, cela signifie que chaque binaire de charge de travail doit être téléchargé manuellement sur le matériel. Les résultats (itérations par seconde) doivent être collectés via une console de débogage (en redirigeant `al_printf`). Le score final CoreMark-PRO est ensuite calculé manuellement (ou via un script Perl sur l'hôte) en utilisant les scores de référence et les facteurs d'échelle fournis dans le `README.md` et la documentation PDF.

Prochaines Étapes pour STM32/IAR

1. **Configuration de l'environnement IAR** : Créer un nouveau projet IAR pour un microcontrôleur STM32 spécifique.

2. **Importation des sources** : Importer les fichiers source pertinents de CoreMark-PRO dans le projet IAR, en excluant les Makefiles et les fichiers spécifiques à d'autres plateformes.

3. **Adaptation de `th_al.c`** : C'est l'étape la plus critique. Implémenter les fonctions de l'AL (`al_main`, `al_signal_*`, `al_thread_*`, `al_printf`, `al_malloc` / `al_free`) en utilisant les API STM32 (HAL/LL) et/ou un RTOS (si utilisé).

4. **Gestion de `argc` / `argv`** : Adapter la fonction `main()` de chaque charge de travail pour qu'elle puisse être appelée sans arguments ou en utilisant une chaîne d'arguments fixe, si le semihosting n'est pas disponible.

5. **Compilation et Débogage** : Compiler chaque charge de travail, la télécharger sur le STM32 et déboguer pour s'assurer que les résultats sont correctement affichés via `al_printf`.

6. **Collecte des Résultats** : Exécuter chaque charge de travail, collecter les scores et calculer le score CoreMark-PRO final.

Cette analyse servira de base pour la création du guide détaillé d'implémentation pour STM32 avec IAR Workbench.

Fonctionnalité Détaillée du Code

Pour comprendre comment CoreMark-PRO fonctionne, il est essentiel d'examiner de plus près les fichiers clés et leurs interactions. Nous allons nous concentrer sur les fichiers pertinents pour l'exécution du benchmark sur une plateforme embarquée.

`mith/al/src/th_al.c` : La Couche d'Abstraction Matérielle

Ce fichier est le pont entre le code du benchmark et le matériel. Voici une explication plus détaillée des fonctions que vous devrez implémenter pour STM32 :

- **`al_main(int argc, char *argv[])`** : C'est le point d'entrée de la couche d'abstraction. Dans un environnement embarqué, cette fonction sera appelée après l'initialisation du système (horloges, etc.). C'est ici que vous devriez initialiser les périphériques

nécessaires au benchmark, comme un timer pour la mesure du temps et une interface de communication (UART, SWO) pour l'affichage des résultats.

- **`al_signal_init()`**, **`al_signal_start()`**, **`al_signal_stop()`**, **`al_signal_destroy()`** : Ces fonctions gèrent la temporisation. Pour STM32, vous pouvez utiliser un des timers matériels (par exemple, TIM2) pour générer une interruption à une fréquence connue (par exemple, 1 kHz pour une résolution de 1 ms). L'interruption incrémentera un compteur global qui sera utilisé pour mesurer le temps d'exécution des benchmarks.

- **`al_thread_create(th_thread_t *thread, void *(*start_routine)(void *), void *arg)`** : Si vous utilisez un RTOS comme FreeRTOS, cette fonction créera une nouvelle tâche. `start_routine` sera la fonction principale de la tâche et `arg` sera son argument. Si vous n'utilisez pas de RTOS (baremetal), vous pouvez utiliser l'implémentation de `al_single.c` qui simule un environnement mono-thread.

- **`al_thread_join(th_thread_t thread, void **retval)`** : Attend la fin d'une tâche. Dans un environnement RTOS, cela correspondra à une fonction comme `vTaskDelete` ou à un mécanisme de synchronisation pour attendre la fin de la tâche.

- **`al_thread_mutex_init()`**, **`al_thread_mutex_lock()`**, **`al_thread_mutex_unlock()`**, **`al_thread_mutex_destroy()`** : Fonctions de gestion des mutex pour la synchronisation entre les threads. Si vous utilisez un RTOS, vous les mapperez aux fonctions de mutex du RTOS (par exemple, `xSemaphoreCreateMutex`, `xSemaphoreTake`, `xSemaphoreGive`). En baremetal, ces fonctions peuvent être vides si vous n'avez qu'un seul thread.

- **`al_printf(const char *fmt, ...)`** : C'est la fonction d'affichage. Pour STM32, vous pouvez la rediriger vers une sortie UART en utilisant les fonctions HAL/LL, ou vers la console de débogage via SWO (Serial Wire Output) ou le semihosting d'IAR.

- **`al_malloc(size_t size)`**, **`al_free(void *ptr)`** : Fonctions d'allocation dynamique de mémoire. Vous pouvez utiliser les fonctions `malloc` et `free` de la bibliothèque C standard, mais assurez-vous que la taille du tas (heap) est correctement configurée dans les options du projet IAR.

`mith/src/mith_main.c` : Le Cœur du Harnais de Test

Ce fichier contient la logique principale du MITH. La fonction `mith_main()` est le point d'entrée du harnais de test. Elle prend en charge les tâches suivantes :

1. **Initialisation** : Initialise le harnais de test, y compris la couche d'abstraction matérielle.
2. **Création des Threads** : Crée les threads de travail qui exécuteront les benchmarks.
3. **Exécution des Benchmarks** : Lance les benchmarks et mesure leur temps d'exécution.
4. **Validation** : Vérifie que les résultats des benchmarks sont corrects en les comparant à des valeurs de référence.
5. **Rapport des Résultats** : Affiche les résultats (itérations par seconde) via `al_printf()` .

`workloads/` : Les Applications de Benchmark

Chaque sous-répertoire de `workloads/` correspond à un benchmark spécifique. Par exemple, `workloads/cjpeg-rose7-preset/` contient le benchmark de compression JPEG. Chaque répertoire de charge de travail contient un fichier `main.c` qui est le point d'entrée de ce benchmark particulier. Ce fichier `main.c` appelle `mith_main()` pour exécuter le benchmark.

Exemple : **`workloads/cjpeg-rose7-preset/bmark_lite.c`**

Ce fichier est le point d'entrée du benchmark JPEG. Il définit la fonction `bmark_lite_main()` qui est appelée par `mith_main()` . Cette fonction effectue les opérations suivantes :

1. **Initialisation** : Initialise les structures de données nécessaires à la compression JPEG.
2. **Exécution** : Appelle les fonctions de compression JPEG du noyau du benchmark (situé dans `benchmarks/consumer_v2/cjpeg/`).
3. **Validation** : Compare le résultat de la compression à une valeur de référence pour s'assurer que le benchmark s'est exécuté correctement.

`benchmarks/` : Les Noyaux de Benchmark

Ce répertoire contient le code source des algorithmes de benchmark eux-mêmes. Par exemple, `benchmarks/consumer_v2/cjpeg/cjpeg.c` contient l'implémentation de l'algorithme

de compression JPEG. Ces fichiers sont appelés par les charges de travail dans le répertoire `workloads/`.

Fichiers Pertinents pour l'Exécution

Pour exécuter CoreMark-PRO sur STM32, vous devrez vous concentrer sur les fichiers suivants :

- **`mith/al/src/th_al.c`** : Le fichier à modifier pour le portage.
- **Les fichiers `main.c` dans chaque répertoire de `workloads/`** : Pour comprendre comment chaque benchmark est lancé et pour éventuellement adapter la gestion des arguments `argc` / `argv`.
- **Les fichiers source dans `mith/src/`** : Pour comprendre le fonctionnement interne du harnais de test.
- **Les fichiers source dans `benchmarks/`** : Pour comprendre les algorithmes de benchmark eux-mêmes (facultatif, car vous n'êtes pas censé les modifier).

En résumé, le processus d'exécution de CoreMark-PRO sur STM32 consiste à prendre le code du benchmark, à le compiler avec la chaîne d'outils IAR, et à fournir une implémentation de la couche d'abstraction matérielle (`th_al.c`) qui permet au benchmark de s'exécuter sur le matériel STM32. Le reste du code du benchmark reste inchangé.