Förderkennzeichen: 01IH130003
Vorhabensbezeichnung: MyThOS
Modulares Betriebssystem für Massiv Parallele Anwendungen

MyThOS D4.2.2 Developer Tools

Stefan Bonfert, Vladimir Nikolov, Robert Kuban, Randolf Rotta

28. März 2017

Zusammenfassung

Zur Unterstützung der Entwickler werden eine Reihe von Softwareentwicklungswerkzeugen zur Verfügung gestellt. Dabei wird soweit möglich auf bestehende Werkzeuge zurückgegriffen und durch Anpassung und Erweiterungen eine Umsetzung realisiert.

Das diesem Bericht zugrunde liegende Vorhaben wurde mit Mitteln des Bundesministerium für Bildung und Forschung unter dem Förderkennzeichen 01IH13003 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

Inhaltsverzeichnis

1 Deliverable D4.2.2 und das Gesamtpaket D.4.2 - *Developer Tools*

Dieses Dokument erweitert den aktuellen Stand der Softwareentwicklungswerkzeuge, welcher zuvor im Deliverable D4.2.1 erfasst wurde. Diese Werkzeuge werden bei der Entwicklung des Kernels und bei der Entwicklung bzw. Anpassung der Anwendungen an die neue Ausführungsumgebung verwendet und werden im Gesamtpaket D4.2 zusammengefasst.

Das Gesamtpaket D4.2 dient somit der Erstellung eines minimalen Werkzeugkastens, welcher für die Entwicklung von Anwendungen und deren Betrieb bzw. Evaluation mit MyThOS bereitgestellt wird. Die verwendeten Werkzeuge sind größtenteils frei verfügbar (z.B. open source). Proprietäre Produkte, wie der Intel C/C++ Compiler, die z.B. für eine Performance-Steigerung der Anwendungen auf der Evaluationshardware verwendet werden können, werden ebenfalls diskutiert, obwohl diese nicht zwingendermaßen für die Entwicklung notwendig sind (vgl. D4.2.1).

Darüber hinaus werden in D4.2 auch Vorgänge und Tools beschrieben, welche bei der Auswertung der Anwendungsperformance verwendet werden und auch solche, die beim aktuellen Betrieb der Anwendungen für das Erstellen und für die Auswertung von Simulationstests benötigt werden. Letztere werden auch für die Evaluation der Anwendungsperformance mit der neuen Betriebssystem-Umgebung zum Einsatz kommen.

In diesem Dokument sind lediglich diejenigen Werkzeuge und Vorgänge beschrieben, die seit der Veröffentlichung von Deliverable D4.2.1 verändert oder zusätzlich in den Entwicklungsprozess aufgenommen wurden. Somit ist dieses Dokument als Erweiterung von D4.2.1 zu betrachten und schließt das Gesamtpaket D4.2 ab.

2 Konfigurations- und Build-Tools

MyThOS unterstützt einen modularen und konfigurierbaren Kernel sowie entsprechende Werkzeuge, mit denen das Erstellen eines für den jeweiligen Anwendungsfall und Zweck angepassten Kernel-Abbilds automatisiert ablaufen kann. Das zentrale Werkzeug für die Verwaltung der Kernel-Konfiguration ist das python-basierte sog. `mcconf`-Tool, welches im Verzeichnis `{mythos}/3rdparty/mcconf` zu finden ist. Die Hauptaufgabe dieses Werkzeugs ist es eine Ansammlung von zu übersetzenden Quellcode-Dateien zu erstellen, die eine gewünschte bzw. benutzer-definierte Kernel-Konfiguration umsetzt, und auch Abhängigkeiten auf der Code-Ebene zu

verwalten. Damit ist es ebenfalls möglich, in einem Schritt automatisiert mehrere Kernel-Versionen für verschiedene Anwendugsfälle und Hardware-Platformen zu erzeugen.

In MyThOS werden zusammengehörende Dateien, die eine eigenständige Einheit bilden, zu Modulen zusammengefasst. Dadurch können sowohl Anforderungen von Modulen wie besipielsweise Compiler-Unterstützung, als auch Unterschiede zwischen den Kernel-Varianten für unterschiedliche Plattformen berücksichtigt werden. Zudem erleichtert die Modul-orientierte Struktur die Evaluation verschiedener Implementierungsvarianten, da einzelne Module einfach austauschbar sind.

Jedes Modul stellt eine Menge von Dateien bereit und kann von anderen Dateien und Modulen abhängen. Solche Abhängigkeiten können beispielsweise Header-Dateien oder spezielle Implementierungsvarianten einzelner Objekte darstellen. Die meisten Abhängigkeiten müssen nicht manuell spezifiziert werden, sondern werden durch das `mcconf` Tool automatisiert aus den `#include` Anweisungen im Quellcode extrahiert. Die Definition eines jeden Kernel-Moduls liegt als Datei (`mcconf.module`) im entsprechenden Verzeichnis im Kernel-Quellcode vor. Diese spezifiziert die vom Modul bereitgestellten Objekte und Dateien, benötigte Objekte und auch spezielle Konfigurationen und Anweisungen für den Build-Prozess (Compiler und Linker). Das `mcconf` Tool traversiert automatisch die Verzeichnisstruktur des Kernel-Quellcodes und registriert alle auffindbaren Moduldefinitionen.

Um automatisiert alle für eine bestimmte Konfiguration benötigten Dateien zusammenzustellen wird eine Beschreibung der gewünschten Kernel-Konfiguration benötigt. Solche Konfigurationen können als Dateien mit definierter Struktur und Syntax bereitgestellt und abgespeichert werden. Diese tragen die Endung `.config`. Beispiele bereits vordefinierter Konfigurationen für `amd64` (`kernel-amd64.config`) und `KNC` (`kernel-knc.config`) Architekturen liegen im MyThOS-Hauptverzeichnis vor. Die entsprechenden Kernel-Konfigurationen werden durch das dort ebenfalls bereitgestellte Makefile automatisch erzeugt und liegen dann als Unterverzeichnisse und Dateien vor. Jede Spezifikation legt fest, welche Module für eine bestimmte Ziel-Konfiguration benötigt werden, um ein lauffähiges Kernel-Image zu erstellen. Zur Auflösung der hier beschriebenen Abhängigkeiten erzeugt das `mcconf` Tool eine Liste aller benötigten Dateien und Module. Anschließend wird für jedes benötigte Objekt ein Modul ausgewählt und zum Kernel-Abbild hinzugefügt, falls dieses eines der benötigten Objekte bereitstellt.

Falls zwei Module die selbe Datei bereitstellen entsteht ein Konflikt, der vom Entwickler manuell aufgelöst werden muss, indem er eines der möglichen Module

```
[module.boot-memory-multiboot]
requires = ["platform:multiboot"]
incfiles = ["boot/memory/Stage3Setup.h"]
kernelfiles = ["boot/memory/Stage3Setup.cc", "boot/memory/Stage3Setup.cc",
"boot/memory/Stage3Setup-multiboot.cc"]

[module.boot-memory-gem5]
requires = ["platform:gem5"]
incfiles = ["boot/memory/Stage3Setup.h"]
kernelfiles = ["boot/memory/Stage3Setup.cc", "boot/memory/Stage3Setup.cc",
"boot/memory/Stage3Setup-e820.cc"]

[module.boot-memory-knc]
requires = ["platform:knc"]
incfiles = ["boot/memory/Stage3Setup.h"]
kernelfiles = ["boot/memory/Stage3Setup.cc", "boot/memory/Stage3Setup.cc",
"boot/memory/Stage3Setup-sfi.cc", "boot/memory/Stage3Setup-knc.cc"]
```

Listing 1: Ein Beispiel einer Modulbeschreibung (`mcconf.module`).

auswählt. Zur einfacheren Auswahl größerer Mengen von Modulen können Tags verwendet werden, die Teil der Modulbeschreibung sind. So sind beispielsweise alle Module, die auf die Xeon Phi Karte zugeschnitten sind, mit dem Tag `platform:knc` markiert.

Im Folgenden wird nochmals speziell auf die Moduldefinitionen eingegangen, da diese insbesondere bei der Entwicklung von Anwendungen und Erweiterung des Kernels zum Einsatz kommen.

Jedes Modul befindet sich mit seinen Quelldateien und der Spezifikation in einem eigenen Ordner. Die Spezifikation wird dabei in einer oder mehreren `*.module`-Dateien festgehalten. Die Pfade zu den einzelnen Quelldateien sind darin relativ zum Pfad der Spezifikation (`*.module`-Datei) angegeben (siehe Listing 1). Der Pfad zum Verzeichnis, das alle vorhandenen Module beinhaltet, ist Teil der Konfigurations-Spezifikation. Listing 1 zeigt die Spezifikation eines Moduls, das die Datei `Stage3Setup.h` für unterschiedliche Plattformen bereitstellt. Anhand des platform-Tags wird die korrekte Implementierungs-Variante ausgewählt.

Um ein komplettes Kernel-Image zu erstellen liest das `mcconf`-Tool alle vorhandenen Modulbeschreibungen ein, wählt anhand einer Konfigurations-Spezifikation Module aus um vorhandene Abhängigkeiten aufzulösen und kopiert die ausgewählten Quell-Dateien in einen Zielordner. Außerdem generiert das Tool ein `Makefile`, das

Regeln für die Übersetzung der Quelldateien und das Linken des fertigen Kernel-Images enthält. Diese Regeln können, wie bereits beschrieben, innerhalb der einzelnen Modul-Spezifikationen spezialisiert werden, die die entsprechenden Teile des `Makefile` enthalten.

Sobald das angepasste `Makefile` für die gesamte Kernel-Konfiguration im entsprechenden Unterverzeichnis erzeugt wurde, kann diese übersetzt und ausgeführt werden. Dafür enthält das `Makefile` automatisch generierte Targets wie `make micrun` – zum Starten auf einem Knights Corner Prozessor – `make qemu` – zur Emulation mit QEMU –, oder zur Aktivierung im GDB Debug-Modus `make qemudgb`. Die Übersetzung und das Erstellen des Kernel-Abbilds geschieht durch `make`.

Für die Konfiguration des Build-Prozesses können separate Module verwendet werden, die keine Quelldateien enthalten. Damit können beispielsweise je nach Zielplattform unterschiedliche Compiler verwendet werden. Diese Module enthalten lediglich Verweise auf unterschiedliche Submodule und deren Versionen oder Code, der direkt in das `Makefile` übernommen wird und damit die Übersetzung von MyThOS an die Zielplattform anpasst.

Weitere Informationen und Beispiele für die Funktionsweise des `mcconf` Tools sowie für die Definition und Umsetzung von Modul-Konfigurationen befinden sich im Projekt-Verzeichnis `{mythos}/3rdparty/mcconf`.

3 Debugging

Für die Fehlersuche im Kernel und in Anwendungsprogrammen kamen zunächst hauptsächlich klassische Debugger in Kombination mit Emulatoren wie Qemu und Bochs zum Einsatz. Wie bereits erklärt, enthält das `Makefile` einer jeden Kernel-Konfiguration entsprechende Targets (wie z.B. `qemugdb`, `qemu-ddd`). Der Emulator und der Debugger können über die Remote-Debugging Werkzeuge beliebiger IDEs, wie z.B. Eclipse, KDevelop etc., angesteuert werden, was das Debugging des Kernels im Emulator erleichtert. Für diese Fälle ist die Kernel-Konfiguration `kernel-amd64.config` vorgesehen.

Um jedoch die Ausführung von parallelem Code auf realer Hardware wie dem Xeon Phi Coprozessor nachvollziehen zu können, sind zusätzliche Werkzeuge notwendig, da die dortige Ausführung des Codes nicht beliebig unterbrochen werden kann.

Zu diesem Zweck wurde ein Tracing Werkzeug entwickelt, das verteilt über alle Hardwarethreads Ausführungs- und Laufzeit-Informationen (Traces) sammelt und

diese aggregiert ausgeben kann. Hierfür wird auf jedem Hardwarethread separater Speicher verwendet, um Synchronisation beim Sammeln einzelner Traces zu vermeiden, da dadurch die Ausführung selbst beeinflusst werden würde. Das Tracing Werkzeug wird momentan erweitert und an die aktuelle Kernel-Architektur angepasst. Entsprechende Dokumentationen und Anleitungen werden auf dem öffentlichen Repository (https://github.com/ManyThreads/mythos) bereitgestellt und auf der Projektseite (https://manythreads.github.io/mythos/) bekannt gegeben.

Zusätzlich wurde ein neues Werkzeug entwickelt, mit dem kausale Zusammenhänge zwischen Ereignissen auf verschiedenen Hardwarethreads extrahiert und anschließend für die Programmanalyse verwendet werden können. Solche Ereignisse können beispielsweise durch Nachrichtenaustausch hervorgerufene Funktionsaufrufe sein. Durch dieses Werkzeug ist es möglich, die Ausführung von parallelem Code im Nachhinein genau zu untersuchen und wichtige Informationen über Parallelität und Abhängigkeiten zu extrahieren. Dadurch können Deadlocks oder Kommunikationsmuster deutlich einfacher untersucht werden. Der Code und die entsprechenden Dokumentationen des Tools werden gerade ebenfalls portiert und erweitert und nach einer entsprechenden Evaluation und Testphase öffentlich zur Verfügung gestellt.